



SELECTIVE PRIORITISATION OF REAL-TIME IP PACKETS TO IMPROVE QUALITY OF SERVICE IN 5G WIRELESS SENSOR NETWORKS

ORLANDO NAVAL DOS SANTOS (BEng Hons)

A Thesis submitted in partial fulfilment of the requirements of the London
Metropolitan University for the Degree of Doctor of Philosophy

JULY 2025

DECLARATION

I certify that this thesis has not been accepted in substance for any other degree and is not being concurrently submitted for any other degree or qualification, except for the Doctor of Philosophy at London Metropolitan University. I further declare that this work is the result of my own independent research, except where otherwise indicated by proper referencing, and that it does not contain any material plagiarised from other sources.

X-----

Orlando Naval Dos Santos

X-----

Professor Bal Virdee

(First supervisor)

X-----

Dr Shahram Salek Zamankhani

(Second supervisor)

ACKNOWLEDGEMENT

First and foremost, I would like to express my deepest gratitude to my supervisors, Professor Bal Virdee and Dr Shahram Salek Zamankhani, for their expert guidance, insightful feedback, and unwavering patience throughout this research journey. Their encouragement and academic support were invaluable in shaping this work, ensuring that it was not only successful but also a significant contribution to an area that is very much topical in all its ramifications.

I would also like to express my gratitude to my wife, Isabel, for her unwavering support, understanding, and encouragement throughout this project. Her reassurance, especially during challenging times, was always timely and exactly what I needed to carry on. Her support made this work not only possible but also a joy to pursue.

My heartfelt appreciation goes to my father, Antonio, and my mother, Ana, for their lifelong belief in me, constant encouragement, and practical support. Their faith in my abilities gave me the strength to keep going.

My appreciation also extends to my sons, Telmo and Braúlio, and my daughters, Patricia and Ayumi, for their patience, joy, and unwavering support. I dedicate this work to my loving family for their encouragement, understanding, and steadfast support, which motivated me to persevere during challenging times and achieve success in this study.

I would also like to express my gratitude to Mr John Hucker for his valuable support in resolving the issues with the remote PC at the University, and to Mr Kevin Willis for the job opportunity that provided me with the necessary peace of mind to undertake this research project.

ABSTRACT

The successful deployment of 5G and future-generation networks depends critically on the provision of Ultra-Reliable Low-Latency Communication (URLLC), which is vital to meeting the stringent requirements of emerging real-time applications. However, the static, rule-based scheduling algorithms traditionally used for Quality of Service (QoS) management are ill-equipped to handle the dynamic, stochastic nature of modern wireless environments. This thesis addresses this critical gap by designing, implementing, and empirically validating a novel, multi-stage Artificial Intelligence (AI) framework that enables a transition from passive network analysis to active, autonomous control. The central research question investigates whether a hierarchical suite of Machine Learning (ML), Deep Learning (DL), and Deep Reinforcement Learning (DRL) techniques can dynamically manage real-time traffic to significantly enhance QoS in 5G Wireless Sensor Networks (WSNs).

The research methodology unfolds in three logical stages, validated within a live network testbed. First, supervised learning techniques – including regression and classification models – are used to quantitatively evaluate and “fingerprint” traditional schedulers, establishing a robust performance baseline and confirming that predictability is a critical QoS metric. Second, the investigation proceeds to predictive forecasting, where a comparative analysis reveals that a tuned RF model leveraging rich instantaneous metrics outperforms complex sequential DL architectures for predicting next-step network delays.

The culmination of this research work is the development of an autonomous scheduling agent. By framing the network scheduling as a Markov Decision Process (MDP), a novel Transformer-based RL agent is trained online. The findings demonstrate that this agent can learn an optimal policy in real time, achieving 100% scheduling success with ultra-low latency and closing the loop from analysis to control.

Collectively, this research contributes a comprehensive, validated AI-driven framework for intelligent QoS management. It provides a practical roadmap for evolving from static network configurations to adaptive, self-learning systems. The results confirm that the proposed hierarchy of AI techniques can successfully navigate the complex trade-offs in 5G WSNs, significantly outperforming traditional methods and paving the way for truly autonomous network control.

TABLE OF CONTENTS

TABLE OF CONTENTS	v
Chapter 1: Introduction.....	1
1.1 Introduction.....	1
1.2 Context and Background	1
1.2.1 5G WSN Overview and Key Characteristics	1
1.2.2 5G QoS Architecture and WSN Integration.....	2
1.2.3 Resource Allocation and Prioritisation in 5G	3
1.3 Motivation and Relevance	4
1.3.1 Ultra-Low Latency Requirements	6
1.3.2 Reliability and Wireless Channel Challenges	6
1.3.3 Core Resource Management Constraints	7
1.3.4 Application Domain and Use Cases.....	8
1.3.5 Research Gaps and Future Directions	14
1.4 Problem Statement.....	16
1.5 Research Aims and Objectives	18
1.6 Methodological Overview of Proposed Approaches	19
1.7 Original Contributions and Novelty of This Work.....	20
1.8 Thesis Outline	21
Chapter 2: Literature Review	23
2.1 Introduction.....	23
2.2 Applications and QoS Demands of 5G WSNs in URLLC Scenarios.....	23
2.2.1 What is a 5G Wireless Sensor Network?	24
2.2.2 Canonical URLLC Application Domains	24
2.2.2.1 Autonomous Vehicle	24
2.2.2.2 Telemedicine and Remote Surgery	25
2.2.2.3 Industrial Automation.....	25
2.2.3 Domain-Specific URLLC QoS Requirements and Budgets	27
2.2.4 Cross-cutting QoS Challenges	29
2.3 Limitations of Traditional Approaches.....	32
2.3.1 Static Resource Slicing and Reservation.....	32

2.3.2 Fixed Priority Scheduling	32
2.3.3 Conventional Optimisation Techniques	33
2.3.4 Latency in Decision Loop	33
2.3.5 Limited Adaptability	34
2.4 ML Approaches (Non-Deep) for URLLC Resource Allocation and Prioritisation Schemes in 5G WSNs.....	34
2.4.1 RL for Scheduling	35
2.4.2 Multi-Armed Bandit and Online Learning.....	35
2.4.3 Heuristic ML (Fuzzy Logic, and similar methods)	36
2.4.4 Federated Learning (FL) for URLLC.....	36
2.4.5 Supervised Learning and Traffic Prediction	37
2.4.6 Hybrid ML-Optimisation Approaches	37
2.5 DL Approaches for 5G WSN (URLLC) Resource Allocation	38
2.5.1 Neural Network-based Scheduling Decisions.....	38
2.5.2 DL for Prediction and Proactive Allocation.....	39
2.5.3 DL for Channel Estimation and Link Adaptation	39
2.5.4 Advanced Neural Network Architectures	40
2.5.5 Real-Time Inference and Complexity	40
2.5.6 Training Data and Strategy	40
2.5.7 Key Achievements of DL in URLLC for 5G WSNs	41
2.6 DRL Approaches for URLLC in 5G WSNs	42
2.6.1 Formulating URLLC Scheduling as a DRL Problem	42
2.6.2 Value-Function-Based DRL (DQN)	42
2.6.3 Policy-Gradient and Actor-Critic DRL	43
2.6.4 Meta-Learning and Transfer in DRL	43
2.6.5 Distributed and Multi-Agent DRL	44
2.6.6 QoS Results from DRL	45
2.6.7 Interpretability and Safety	46
2.7 Where AI-Driven 5G QoS Falls Short.....	47
2.8 How This Study Advances AI-Based QoS: Predict-Verify-Adapt.....	48
2.9 Conclusions and Future Directions	50
2.9.1 Summary of Findings	50
2.9.2 Challenges and Considerations	51

2.9.3 Outlook for 6G and Beyond	51
2.9.4 Concluding Remarks	52
Chapter 3: Research Methodology	53
3.1 Introduction.....	53
3.2 Research Approach	54
3.2.1 Philosophical Assumptions	54
3.3 Research design	55
3.3.1 Simulation and Modelling Approaches for Comparison of scheduling algorithms using ML/DL models with data captured from an operational network.....	55
3.3.1.1 Experimental Environment Setup.....	55
3.3.1.2 Experimental Network Environment.....	55
3.3.1.3 Data Collection and Dataset Creation	56
3.3.1.4 Rationale for Applying ML Models to Predict and Classify Live Network Data under QoS Schemes	57
3.3.1.5 Model Selection Rationale.....	58
3.3.1.6 Common Modelling Pipeline for ML/DL/DRL	61
3.3.1.7 Feature Selection	61
3.3.1.8 Feature creation	63
3.3.2 Framework for Designing and Implementing AI Models (ML, DL, DRL) for Classification/Prediction of Scheduling Algorithms with Data Captured from an Operational Network	63
3.3.2.1 Experimental Setup	63
3.3.2.2 Scheduling Algorithms	63
3.3.2.3 Wireless sensor network traffic generation	64
3.3.2.4 Data collection in Live Network Environment.....	65
3.3.2.5 Data preprocessing	65
3.3.2.6 Network traffic data conversion for CNN Model.....	66
3.3.2.7 Rationale for Using DL in Scheduling Algorithm Classification and Prediction.....	67
3.3.3 Validation and Evaluation Methods.....	69
3.3.4 Summary	73
Chapter 4: Design and Implementation of Experimental Systems.....	74
4.1 Introduction.....	74

4.2 Comparison Evaluation of Scheduling Disciplines Using ML Regression Models	78
4.2.1 Scheduling Algorithms.....	79
4.2.2 Experimental Configuration and Parameters	79
4.2.3 Collected QoS Metrics	80
4.2.4 Data Logging and ML-Ready Output for Enhanced Scheduler Comparison	82
4.3 Classification/Prediction of Scheduling Algorithms Using ML/DL Models with Data Captured from Operational Network.....	83
4.3.1 Experimental Setup and Scenario Design	83
4.3.2 QoS implementation using Prio Queuing Discipline	84
4.3.3 QoS Implementation using Hierarchical Token Bucket Queuing Discipline	85
4.3.4 QoS Implementation Using Hybrid HTB-PRIO Queueing Disciplines.....	87
4.3.5 QoS Implementation Using Class-Based Queueing Discipline	88
4.3.6 QoS Implementation using Class-Based Queueing (CBQ) and Hierarchical Queueing Disciplines	90
4.3.7 Real-time data collection.....	91
4.4 Design and Implementation of ML Algorithms for Traffic Scheduling Analyses	93
4.4.1 ML Classification Models	98
4.4.2 ML Regression Models for Live Network Traffic Data	102
4.5 Design and Implementation of a DL (CNN) Model for Image-based Representation of Network Traffic Dataset	104
4.5.1 CNN-Based Network Traffic Classification	104
4.6 Design and Implementation of DL (CNN-LSTM) Model for Image-based Representation of Network Traffic Dataset	110
4.6.1 CNN-LSTM Model for Network Traffic Classification	110
4.7 Design and Implementation of LSTM Model for Scheduling Classification and Prediction	111
4.8 Experimental Design and Implementation of Delay Prediction Models	113
4.8.1 Dataset and Preprocessing.....	113
4.8.2 Implementation Tools and Reproducibility.....	114
4.8.3 Hybrid Linear-LSTM-Transformer Model	115
4.8.4 LSTM Model.....	116
4.8.5 Random Forest Baseline.....	117
4.8.6 Comparative Summary of Model Architecture	118

4.9 Design and Implementation of a Transformer-based RL Agent for Network Traffic Scheduling	119
4.9.1 Problem Formulation.....	120
4.9.2 State Representation.....	121
4.9.3 Action Space Definition	121
4.9.4 Reward Function Design.....	121
4.9.5 Transformer Architecture Setup.....	121
4.9.6 Policy Network Definition	123
4.9.7 Value Network Construction.....	123
4.9.8 Training Loop Initialisation	123
4.9.9 Exploration vs. Exploitation.....	124
4.9.10 Gradient Update and Learning	124
4.9.11 Policy Evaluation and Adjustment.....	124
4.9.12 Deployment and Real-Time Adaptation	125
4.9.13 Experimental Setup in a Live Network.....	125
Chapter 5: Results / Findings and Discussion	133
5.1 Introduction.....	133
5.1.1 Research Problems and Scientific Questions	133
5.1.2 Chapter Roadmap.....	135
5.2 Experiment 1: Comparative Evaluation of Scheduling Algorithms with ML Regression Models.....	136
5.2.1 Proportional Fair Dataset Results.....	136
5.2.1.1 Summary of PF Dataset.....	136
5.2.1.2 Performance Metrics for PF Scheduler Experiment.....	139
5.2.1.3 Interpretation of Results for PF Scheduling Experiment	146
5.2.2 Round Robin Dataset Results.....	149
5.2.2.1 Summary of Round Robin Dataset.....	149
5.2.2.2 Performance Metrics for RR Scheduler Experiment.....	151
5.2.2.3 Interpretation of Round Robin Scheduler Experiment Results	157
5.2.3 Comparison of PF and RR Results.....	160
5.2.4 Statistical Significance of Validation Metrics (PF vs RR, EGPR (Experiment 1))	162
5.3 Experiment 2: Classification-based Comparison of Scheduling Algorithms Using Operational Network Traffic	163

5.3.1 Performance Evaluation of ML Models on the PRIO Dataset.....	164
5.3.1.1 Summary of Prio Dataset.....	164
5.3.1.2 Performance Metrics for PRIO Dataset.....	165
5.3.1.3 Interpretation of Results for Prio Dataset.....	171
5.3.1.4 Comparative Performance Analysis (PRIO Dataset): Optimizable Tree vs. Fine Gaussian SVM.....	179
5.3.2 Performance Evaluation of ML Models on the Hierarchical Token Bucket (HTB) Dataset	181
5.3.2.1 Summary of HTB Dataset	181
5.3.2.2 Performance Metrics for HTB Dataset	182
5.3.2.3 Interpretation of Results for HTB Dataset.....	187
5.3.2.4 Comparative Performance analysis (HTB Dataset): Optimizable Tree vs. Fine Gaussian SVM vs. Fine Tree.....	188
5.3.3 Performance Evaluation of ML Models on the Class-Based Queuing (CBQ) Dataset.....	189
5.3.3.1 Summary of CBQ Dataset	189
5.3.3.2 Performance Metrics for CBQ Dataset.....	190
5.3.3.3 Interpretation of Results for CBQ Dataset	194
5.3.3.4 Comparative Performance Analysis (CBQ Dataset) Across Models (Optimisable Tree, SVM, and Fine Tree).....	197
5.3.4 Dataset-Specific Performance for the Optimizable Tree	199
5.3.5 Statistical Significance of Classification Metrics (Experiment 2)	200
5.4 Experiment 3: Comparison of Scheduling Algorithms Using DL Models with Data Captured from Operational Network	201
5.4.1 CNN-Based Scheduling Classifications: Results on the Combined Image-Based Dataset (Prio, HTB, and CBQ).....	202
5.4.1.1 Summary of Image-based Dataset.....	202
5.4.1.2 Performance Metrics for the CNN Model	202
5.4.1.3 Interpretation of Results for CNN Model.....	207
5.4.2 CNN-LSTM –Based Scheduling Predictions: Results on the Combined Image-Based Dataset (No_QoS, Prio, HTB, and CBQ).....	209
5.4.2.1 Performance Metrics for the CNN-LSTM Model	209
5.4.2.2 Interpretation of Results for CNN-LSTM Model.....	211
5.4.3 LSTM-Based Scheduling Classification: Results on the Prio, HTB, and CBQ Datasets	214

5.4.3.1 Considerations Regarding the Datasets for the LSTM Model	215
5.4.3.2 Performance Metrics for the LSTM Models	215
5.4.3.3 Interpretation of Results for LSTM-Based Scheduling Prediction Model	219
5.4.4 Comparison of CNN-Based, CNN-LSTM-Based and LSTM-Based Scheduling Predictions Models.....	222
5.4.5 Statistical Significance of Validation Metrics (Experiment 3)	223
5.5 Experiment 4: Comparative Evaluation of Network-Delay Prediction Models .	225
5.5.1 Summary of Experimental Setup	225
5.5.2 Hybrid Model (LSTM and Transformer) Performance Analysis.....	226
5.5.3 Pure-LSTM Model Performance Analysis.....	229
5.5.4 Random Forest Baseline Performance Analysis	232
5.5.5 Quantitative Results Analysis	235
5.5.6 Comparison Analysis	236
5.5.7 Statistical Significance of Prediction Metrics (Experiment 4).....	238
5.5.8 Discussion of Findings	238
5.5.9 Implications for URLLC in 5G WSNs.....	240
5.6 Experiment 5: Transformer-Based RL Agent for Network Traffic Scheduling .	242
5.6.1 Data summary for the Transformer-Based RL Scheduling Agent.....	242
5.6.2 Performance Metrics for the Transformer-Based RL Model	242
5.6.2.1 Experiment 1 for Transformer-Based RL Scheduler.....	243
5.6.2.2 Experiment 2 for Transformer-Based RL Scheduler.....	244
5.6.2.3 Experiment 3 for Transformer-Based RL Scheduler.....	245
5.6.3 Interpretation of Results for Transformer-Based RL Scheduler	246
5.6.3.1 Observed Results	247
5.6.3.2 Impact of Model's Components	247
5.6.3.3 Comparison with Similar Studies	248
5.6.3.4 Alignment with Industrial Benchmarks.....	250
5.6.4 Enhanced Transformer-Based RL Scheduling Agent	251
5.6.4.1 Enhanced Model Performance Analysis.....	252
5.6.5 Statistical Significance for RL Scheduling Metrics (Experiment 5).....	256
5.6.5.1 Updated Benchmark Alignment after Tail-Latency and Statistical Validation	262

5.6.5.2 Robustness Implications for the Prior-Work Comparison under Scaled, Tail-Aware Evaluations	263
5.7 Cross-Dataset and Cross-Experiment Comparison and Discussion	263
5.7.1 Cross-Dataset Comparison	263
5.7.2 Cross-Experiment Comparison	266
5.7.3 Unified Performance Comparison across Experiments	267
Chapter 6: Synthesis and Implications of Findings	273
6.1 Introduction.....	273
6.2 Findings and Implications: Comparative Evaluation of PF and RR Scheduling with EGPR	273
6.2.1 Brief Context	273
6.2.2 Summary of Key Findings (PF vs RR)	273
6.2.3 Contributions to the Field.....	275
6.2.4 Answer to Research Questions.....	276
6.2.5 Limitations and Future Directions.....	278
6.3 Findings and Implications: Classification-based Comparison of Scheduling Algorithms Using Operational Network Traffic.....	279
6.3.1 Brief Context.....	279
6.3.2 Summary of Key Findings	280
6.3.3 Contributions to the Field.....	281
6.3.4 Answer to Research Questions.....	282
6.3.5 Limitations and Future Directions.....	284
6.4 Findings and Implications: Comparison of Scheduling Algorithms Using DL Models with Data Captured from Operational Network.	286
6.4.1 Brief Context.....	286
6.4.2 Summary of Key Findings	287
6.4.3 Contributions to the Field.....	288
6.4.4 Answer to Research Questions.....	289
6.4.5 Limitations and Future Directions.....	290
6.5 Findings and Implications: Experimental Design of Delay Prediction Models (Hybrid (Linear+LSTM+Transformer), LSTM, Random Forest)	292
6.5.1 Brief Context.....	292
6.5.2 Summary of Key Findings	292
6.5.3 Contributions to the Field.....	293

6.5.4 Answer to Research Questions.....	294
6.5.5 Limitations and Future Directions.....	295
6.6 Findings and Implications: Transformer-Based RL Agent for Network Traffic Scheduling	296
6.6.1 Brief Context.....	296
6.6.2 Summary of Key Findings	296
6.6.3 Contributions to the Field.....	297
6.6.4 Answer to Research Questions.....	297
6.6.5 Limitations and Future Directions.....	298
6.7 Broader Implications for 5G and Emerging Applications Domains.....	298
6.7.1 Broader Implications for Experiment 1: Comparative Evaluation of PF and RR Scheduling.....	298
6.7.2 Broader Implications for Experiment 2: Classification-Based Comparison of Scheduling Algorithms.....	299
6.7.3 Broader Implications for Experiment 3: DL-Based Scheduler Identification	301
6.7.4 Broader Implications for Experiment 4: Predictive QoS Forecasting.....	302
6.7.5 Broader Implications for Experiment 5: Transformer-Based RL Scheduler	303
6.8 Recommendations for Practitioners and Researchers.....	304
6.8.1 Recommendations for Experiment 1	304
6.8.2 Recommendations for Experiment 2	305
6.8.3 Recommendations for Experiment 3.....	307
6.8.4 Recommendations for Experiment 4.....	308
6.8.5 Recommendations for Experiment 5.....	309
Chapter 7: Overall Conclusions.....	311
7.1 Introduction.....	311
7.2 Synthesis of Key Findings: A Narrative from Analysis to Autonomy	311
7.3 Primary Contributions to Knowledge and Broader Implications	312
7.4 Limitations and Future Research Directions.....	313
7.5 Cross-cutting limitations.....	314
7.6 Future Research Directions.....	316
7.7 Concluding Remarks.....	317
References	319
Appendices	329

Appendix A: Implementation of ML Algorithms for Classification in MATLAB ..	329
Appendix B: Implementation of ML algorithms for Regression in MATLAB.....	330
Appendix C: Implementation of DL Models in MATLAB	334
Appendix D: Implementation of ML and DL Models Using Python	340
Appendix E: Source Codes for Experiment 1 (Comparative Evaluation of Scheduling Algorithms with ML Regression Models)	363
Appendix E.1: Source Code for Proportional Fair Scheduler	363
Appendix E.2: Source Code for Round Robin Scheduler	380
Appendix F: Source Code for Experiment 3 (Comparison of Scheduling Algorithms using DL Models with Data Captured from Operational Networks).....	394
Appendix F.1: Source Code for the LSTM Model for Scheduling Classification	395
Appendix G: Source Code for Experiment 4 (Comparative Evaluation of Network-Delay Prediction Models)	413
Appendix G.1: Source Code for the Hybrid (Linear+LSTM+Transformer) Model for QoS Prediction	413
Appendix G.2: Source Code for the pure-LSTM Model for QoS Prediction	432
Appendix G.3: Source Code for the RF Baseline Model for QoS Prediction.....	448
Appendix H: Source Code for Experiment 5 (Transformer-Based RL Scheduler Agent)	458
Appendix H.1: Transformer-Based RL Scheduler Version 2.0	458

LIST OF FIGURES

Figure 3.1: The figure illustrates the design workflow to achieve the objective.....	66
Figure 3.2: Design workflow for ML/DL Models.....	68
Figure 3.3: Design workflow to train neural network models.....	69
Figure 4.1: Illustration of the Experimental Network Scenario.	84
Figure 4.2: PRIO queueing discipline configuration on a Linux router.	85
Figure 4.3: Illustration of the HTB queueing discipline configuration on a Linux router.....	86
Figure 4.4: Illustration of a hybrid HTB-PRIO queueing discipline Implementation....	87
Figure 4.5: Illustration of experiments using the CBQ queue to gather data for analysis.	89
Figure 4.6: Illustration of CBQ configured with three classes (cbq, pfifo, and sfq).	91
Figure 4.7: Illustration of Wireshark Traffic Capturing From Linux Router Interface..	92
Figure 4.8: Illustration of ML Models Training with PRIO Dataset in MATLAB's Classification Learner App.....	100
Figure 4.9: Extract of the Image Dataset for CNN DL Model.	105
Figure 4.10: The Proposed CNN Architecture.	107
Figure 4.11: Distribution of the training data for DL Classification Models.	108
Figure 4.12: Illustration of Training Options Interface in the MATLAB DL App.	109
Figure 4.13: Network Topology and Traffic Model.	127
Figure 4.14: Flowchart of the proposed model.....	132
Figure 5.1: Delay prediction using the EGPR model under PF scheduling. The close overlap between predicted and true delays across throughput values indicates accurate delay-throughput modelling, with only a few low-throughput delays slightly underestimated.....	141
Figure 5.2: EGPR model performance for delay prediction versus average throughput under PF scheduling. The weak correlation and large scatter indicate that average throughput is a poor predictor of delay, and the model struggles to capture the high-delay variability, especially at lower average throughputs.....	142
Figure 5.3: EGPR model performance for delay versus jitter under PF scheduling. Delay generally increases with jitter, confirming a positive association; however, the wide scatter – especially at low jitter – shows substantial variability that the model does not fully capture.....	143
Figure 5.4: EGPR Model Performance – Delay vs Device in PF network. Predicted and true delays reveal that Device 3 has a markedly higher median delay and wider spread	

than Devices 1, 2, 3, and 4, consistent with the descriptive statistics in Table 5.4. The EGPR model correctly ranks devices by delay but slightly underestimates the extreme values for Device 3. 144

Figure 5.5: EGPR model performance – delay vs packet loss in a PF network. Delay shows a non-linear relationship with packet loss: some of the most significant delays occur when only 2-4 packets are lost, while very high loss (> 10 packets) often coincides with lower delays. The model broadly follows this pattern; however, the wide spread of points indicates that packet loss alone is an insufficient predictor of delay. .145

Figure 5.6: EGPR model performance on the PF dataset. The combination of low RMSE/MAE and high R^2 indicates that the model fits the PF delay data well, capturing the main delay-throughput relationships with relatively small prediction errors. 147

Figure 5.7: EGPR model performance – delay vs instantaneous throughput in a RR network. Delay decreases sharply as instantaneous throughput increases, and the EGPR predictions follow this inverse trend, confirming instantaneous throughput as a strong delay predictor under RR, with only some substantial delays at high throughputs slightly underestimated. 153

Figure 5.8: EGPR model performance – delay vs average throughput in a RR network. Average throughput shows only a very weak, highly scattered relationship with delay, and the EGPR model fails to capture many high-delay outliers – especially at lower average throughputs – confirming that average throughput is a poor predictor of delay in this scenario. 154

Figure 5.9: EGPR model performance – delay vs jitter in a RR network. Delay generally increases with jitter, confirming a positive association. However, the greater scatter (especially at low jitter) and systematic underestimation at high jitter indicate that the model captures the overall trend while failing to fully represent the variability of high-delay events. 155

Figure 5.10: EGPR model performance – delay vs device in a RR network. The box plots show that Device 3 has a much higher median delay and wider spread than Devices 1, 2, and 4, and the model correctly identifies this worst-performing device, although it underestimates the full variability and extreme delay outliers. 156

Figure 5.11: EGPR model performance on the RR dataset. The high validation R^2 (≈ 0.94) and low MAE (≈ 0.91 ms) show that the model captures most of the delay variability, while the RMSE (≈ 6 ms) indicates non-negligible errors on extremely high-delay events due to the wide delay range (up to ≈ 300 ms). 158

Figure 5.12: Optimisable Tree validation confusion matrix (number of observations) for the Prio dataset. Most high-volume protocols (especially TCP and TLS variants are correctly classified, with only occasional confusion between TLSv1.2, TLSv1.3 and a few HTTP/DNS flows, indicating generally strong but not perfect protocol discrimination. 166

Figure 5.13: Optimisable Tre PPV-FDR validation confusion matrix (Prio dataset). Most protocols (e.g., ARP, DNS, and Quick UDP Internet Connections (QUIC)) exhibit very high PPV and low FDR, indicating few false positives, whereas TLSv1 and especially TLSv1.3 show much higher FDR, revealing substantial confusion within the TLS family. 167

Figure 5.14: Optimisable Tree TPR-FNR validation confusion matrix (Prio dataset). ARP, NTP, and SSDP achieve near-perfect TPR with very low FNR, whereas MDNS and TLSv1 exhibit much higher FNR, revealing systematic blind spots in these protocols that could affect QoS decisions.	168
Figure 5.15: Fine Gaussian SVM TPR-FNR validation confusion matrix (Prio dataset). The SVM detects TCP and TLSv1.2/TLSv1.3 with very high TPR and low FNR, but performs poorly on DNS, MDNS, QUIC, UDP, NTP, OCSP, and TLSv1, indicating that several non-encrypted or low-volume protocols remain difficult to recognise reliably.	169
Figure 5.16: Optimisable Tree Validation ROC curve (Prio dataset). Very high AUC values for UDP, TCP, TLSv1.2, and QUIC indicate that these protocols are, in principle, highly separable, although the QUIC operating point trades a high TPR for a relatively high FPR, highlighting a QoS trade-off between sensitivity and false alarms.	170
Figure 5.17: Minimum classification error for the Optimisable Decision Tree (Prio dataset). Hyperparameter optimisation reduces the minimum classification error from ~0.092 to ~0.07, demonstrating that tuning tree depth and split criteria yields a meaningful improvement in protocol classification performance.	171
Figure 5.18: Optimisable Tree PPV-FDR validation confusion matrix (HTB dataset). Most protocols (ARP, DNS, MDNS, NTP, SSDP, and UDP) achieve very high PPV and low FDR, indicating few false positives, whereas ICMP, TCP and especially the TLS variants show higher FDR, revealing notable cross-misclassification and reduced precision for these classes.	183
Figure 5.19: Optimisable Tree TPR-FNR validation confusion matrix (HTB dataset). ARP, ICMP, MDNS, and SSDP achieve perfect detection (TPR=100%), while DNS, NTP, and TCP retain high TPRs ($\geq 85\%$) with limited misses. In contrast, QUIC and UDP show much weaker detection (0-56% TPR), and TLSv1 remains hard to recognise, highlighting clear protocol-specific strengths and weaknesses of the classifier.	184
Figure 5.20: Fine Gaussian SVM validation confusion matrix (TPR-FNR) for the HTB dataset. ARP, SSDP, TCP, TLSv1.2 and TLSv1.3 achieve near-perfect TPRs (~99-100%), while MDNS, NTP and QUIC are never detected (TPR=0%, FNR=100%), and UDP, ICMP and DNS show very high FNRs, revealing strong performance for major TCP/TLS traffic but serious weaknesses for several low-volume or non-TCP protocols.	185
Figure 5.21: Optimisable Tree validation ROC curve (HTB dataset). Most protocols (ARP, MDNS, ICMP, NTP, TLSv1.2, and TLSv1.3) achieve very high AUCs, indicating excellent separability, whereas QUIC (AUC ≈ 0.50) and, to a lesser extent, UDP (AUC ≈ 0.78) remain hard to distinguish, consistent with their poorer performance in the confusion matrices.	186
Figure 5.22: Optimisable Tree TPR-FNR validation confusion matrix (CBQ dataset). The model achieves perfect detection for ARP, SSDP and TCP and high TPR for TLSv1.3 and MDNS; however, it completely fails to detect QUIC, NTP, ICMPv6 and almost all UDP traffic (FNR $\approx 100\%$), indicating that while CBQ-shaped bulk protocols are reliably classified, low-volume or encrypted flows remain poorly learned, risking systematic QoS misallocation for these protocols.	191

Figure 5.23: Fine Gaussian SVM TPR-FNR validation confusion matrix (CBQ dataset). ARP, SSDP, TCP and TLSv1.2/TLSv1.3 achieve high TPRs ($\approx 75\text{-}100\%$), but MDNS, ICMPv6, NTP, QUIC and almost all UDP traffic are never detected (FNR $\approx 100\%$), showing that while the SVM models significant TCP/TLS traffic well, it fails on several low-volume or UDP-based protocols, which can bias QoS decisions against these flows.	192
Figure 5.24: Fine Tree validation ROC curve for the CBQ dataset. Protocols such as ARP, SSDP, TCP and TLSv1.2/TLSv1.3 achieve AUCs close to 1.0, indicating excellent separability, whereas NTP, QUIC and especially ICMPv6 show much lower AUCs (~ 0.6 and 0.33), highlighting poor discrimination and a higher risk of misclassification for these protocols.	193
Figure 5.25: Training progress for the CNN model (30 epochs, LR=0.01). Validation accuracy remains at chance level (25%) while training loss decreases, indicating poor generalisation. This suggests the network is merely memorising specific training examples rather than capturing distinct feature patterns.	203
Figure 5.26: Training progress for the CNN model (80 epochs, LR=0.001). Lowering the learning rate and extending training raises validation accuracy to 62.5%, showing better generalisation than the first run; however, the gap to higher accuracies indicates that further tuning or more data are still needed.....	204
Figure 5.27: Training progress for the CNN model (200 epochs, LR=0.001). Extending training at a low learning rate increases validation accuracy to 87.5% and yields smooth, convergent loss curves, demonstrating that longer training under tuned hyperparameters enables the CNN to learn highly discriminative features from image-based QoS data.	205
Figure 5.28: Training progress for the CNN model (1000 epochs, LR=0.01). Validation accuracy reaches 100% on the held-out set, indicating perfectly separable QoS classes under this configuration; however, the near-flawless curves also suggest the need for additional checks (e.g., revise cross-validation or external data) to exclude overfitting or data leakage.	206
Figure 5.29: Confusion matrix for the CNN model (1000 epochs, LR=0.01). All four classes (CBQ, HTB, No_QoS, and PRIO) are perfectly classified, with 100% of samples on the diagonal and zero off-diagonal entries, visually confirming the 100% validation accuracy and indicating that the CNN has learned entirely separable representations for these scheduling disciplines on the held-out set.	207
Figure 5.30: Training progress for the CNN-LSTM model (500 epochs, piecewise learning rate). Validation accuracy stagnates at 25% despite a smoothly decreasing loss, indicating that the learning rate schedule (decaying to 0) prevents the model from learning discriminative features and leaves performance at chance level.....	210
Figure 5.31: Training progress for the CNN-LSTM model (400 epochs, piecewise LR to 0). Training and validation accuracy both plateau around 50%, while the loss stabilises, indicating underfitting: the model only learns rudimentary features and fails to improve beyond moderate performance under this learning rate schedule.....	211
Figure 5.32: Training and validation loss for the LSTM-based scheduling prediction model. Both curves converge to similar low values after a brief early spike in validation loss, indicating stable learning, no marked overfitting, and good generalisation for predicting the appropriate scheduling discipline.	215

Figure 5.33: Confusion matrix for the LSTM-based scheduling prediction model on the combined PRIO, HTB, and CBQ datasets, showing that > 99% of samples fall on the diagonal with only minor confusion for CBQ, confirming near-perfect discrimination between scheduling algorithms. 216

Figure 5.34: ROC curves for the LSTM-based scheduling prediction model. All three classes (CBQ, HTB, and PRIO) achieve AUC=1.0, with curves clustered near the top-left corner, confirming almost perfect discrimination between scheduling algorithms and very low false-positive rates, in line with the confusion matrix and classification report.....217

Figure 5.35: Precision-recall curves for the LSTM-based scheduling prediction model. All three classes (CBQ, HTB, and PRIO) remain near the upper-right corner, with average precision ≈ 1.0 , confirming that the model maintains high precision and recall across decision thresholds, in line with its near-perfect ROC and confusion-matrix results.....218

Figure 5.36: Classification report for the LSTM-based scheduling prediction model, showing near-perfect precision, recall, and F1-scores for CBQ and HTB and slightly lower precision for PRIO (0.97). The overall accuracy of 0.9948 and the macro- and weighted-average scores of ≈ 0.99 confirm that the model reliably distinguishes among the three scheduling algorithms.....219

Figure 5.37: Training and validation MSE loss for the hybrid Linear-LSTM-Transformer QoS predictor. Loss drops rapidly in the first ≈ 20 epochs and then plateaus, with a minimum validation loss of ≈ 7.9 and a small persistent gap to the training curve, indicating good overall fit, and mild overfitting on unseen devices. 227

Figure 5.38: Final regression RMSE, MAE, and R^2 for the hybrid Linear-LSTM-Transformer model. The much better training than validation scores (RMSE 1.76 vs 4.33 ms; R^2 0.989 vs 0.911) confirm moderate overfitting, yet validation MAE of ≈ 1.07 ms and $R^2 \approx 0.91$ reveal that the model still predict latency for unseen devices with reasonably high accuracy, apart from occasional larger errors. 228

Figure 5.39: True vs predicted delay for the hybrid Linear-LSTM-Transformer model on the validation device. Most points lie close to the diagonal for 0-100 ms delays, confirming good overall accuracy ($R^2 \approx 0.91$), while the wider spread for > 100 ms delays shows occasional misestimation of worst-case latency, which may limit strict tail-latency guarantees. 229

Figure 5.40: Training and validation loss for the LSTM-only model. Loss drops rapidly in the first few epochs and reaches a best validation MSE of ≈ 5.0 around epochs 10-15, with train and validation curves closely aligned, indicating strong convergence, minimal overfitting, and better fit than the hybrid model.....230

Figure 5.41: Final regression RMSE, MAE and R^2 for the LSTM-only model. The small gap between training (RMSE 1.28 ms, R^2 0.994) and validation (RMSE 2.36 ms, R^2 0.974) metrics indicates minimal overfitting, while the low validation MAE (≈ 0.59 ms) shows that the LSTM predicts latency for unseen devices with sub-millisecond average error outperforming the hybrid model.....231

Figure 5.42: True vs predicted delay for the LSTM-only model on the validation device. Most points lie very close to the diagonal for 0-100 ms delays, with only small deviations at the highest delays, confirming the high R^2 (≈ 0.97) and showing that the LSTM predicts latency more accurately and with less spread than the hybrid model. .232

Figure 5.43: Regression performance of the RF baseline, showing sub-millisecond validation error (RMSE ≈ 0.62 ms, MAE ≈ 0.15 ms, $R^2 \approx 0.998$) and a minimal train-validation gap, indicating excellent generalisation to the held-out device and near-deterministic mapping from QoS metrics to delay.	233
Figure 5.44: True vs predicted delay for the RF model on the validation set. Points lie almost exactly on the diagonal across the full delay range, with only tiny deviations, confirming the very high R^2 and showing that the RF predicts latency for the unseen device with near-deterministic accuracy.	234
Figure 5.45: Feature importance scores from the RF model. Instantaneous throughput (bps) is by far the most influential predictor of delay, while other QoS metrics (e.g., average throughput, jitter, packet loss and device-related features) have lower importance, confirming that current transmission rate is the primary driver of latency in this dataset.	235
Figure 5.46: Results for RL Experiment 1 (250 epochs, LR = $1e-6$ - $1e-3$). RTTs range from ≈ 2.5 to 5.5 ms (192.168.1.142 fastest, 1.1.1.1 slowest), the loss initially fluctuates then converges near zero, cumulative reward increases linearly to $\approx 250,000$, and the success ratio remains at 100%, showing that the scheduler rapidly learns a stable, highly reliable, low-latency policy in the live network.	243
Figure 5.47: Results for RL Experiment 2 (350 epochs, LR = $1e-6$ – $1e-3$). RTTs remain low (≈ 2.7 - 6.5 ms) with some sites improving compared to Experiment 1. The loss drops rapidly to a near-zero, stable level, cumulative reward increases linearly to $\approx 350,000$, and the success ratio remains at 100%, confirming a rapidly convergent and highly reliable scheduling policy in the live network.....	244
Figure 5.48: Results for RL Experiment 3 (500 epochs, LR = $1e-6$ – $1e-3$). RTTs remain low (≈ 3 - 8.5 ms) with 192.168.1.140 consistently fastest and 192.168.1.141 slowest; the initially very high loss (≈ 1200) rapidly decays and converges near zero, cumulative reward increases linearly to $\approx 500,000$, and the success ratio stays at 100%, confirming a stable, highly reliable scheduling policy under longer training in the live network.	245
Figure 5.49: Final average latency per site for the enhanced Transformer-based RL scheduler. Most sites achieve low average RTTs (a few milliseconds), while a small number of distant or less favourable sites (e.g., site 12) exhibit much higher delay, showing that the learned policy consistently favours stable low-latency paths when more devices are present.....	253
Figure 5.50: Training loss over epochs for the enhanced Transformer-based RL scheduler. Loss decreases steadily and then stabilises near zero, confirming that the agent converges reliably even after scaling to a larger number of devices.	253
Figure 5.51: Overall reliability for the enhanced Transformer-based RL scheduler. The ping success ratio remains 100% across all epochs and sites, indicating zero packet loss and confirming that scaling to more devices does not degrade reliability in the live network.	254
Figure 5.52: Average reward over epochs for the enhanced Transformer-based RL scheduler. The reward remains consistently high (≈ 980 - 992) across training, confirming that the agent persistently selects low-latency paths even after scaling the number of devices, with only minor fluctuations due to exploration of alternative sites.	255

Figure 5.53: Final average throughput per site for the enhanced Transformer-based RL scheduler. The policy concentrates traffic on the best-performing low-latency site (site 3, > 120,000 bps), while higher-latency sites receive much lower throughput, confirming that the agent exploits the latency-throughput trade-off to favour stable, high-capacity paths.	255
Figure 5.54: Latency Tail Visibility by Network Segment and Compliance with Stringent URLLC Targets.	257
Figure 5.55: Per-Site Latency Distribution (p50-p99) Grouped by Network Segment.	258
Figure 5.56: Full Latency Probability Densities (Violin Plots) per Site: Visualising Skewness and Tail Risks.	260
Figure 5.57: Temporal Evolution of Rolling 99th Percentile (p99) Latency across Network Segments.....	261
Figure 5.58: Cross-experiment latency and prediction error. (a) One-way operational latency under PF and RR (Experiment 1) and the Transformer-based RL scheduler (Experiment 5); bars indicate the mean latency value and whiskers denote the observed range (min-max). (b) Validation prediction error (RMSE and MAE, ms) for EGPR, hybrid Linear + LSTM + Transformer, LSTM, and Random Forest delay prediction models.....	270
Figure 5.59: Envelope of best performance across experiments. Radar plot of normalised scores (0-1) where larger values indicate better performance, formed by the strongest result on each metric across experiments: classification AUC (LSTM classifier), regression R^2 and validation RMSE (Random Forest regressor), and minimum one-way latency and reliability (Transformer-based RL scheduler). The polygon, therefore, represents an envelope of the best results from different models rather than a single model profile.	271
Figure 5.60: Reliability comparison across scheduling approaches. Per-device delivery ratios for PF and RR (Experiment 1) are shown alongside per-device success ratios for the Transformer-based RL scheduler (Experiment 5).	272

LIST OF TABLES

Table 2.1: Comparison of Applications by Operational Principle, Traffic Characteristics and Requirements.	26
Table 2.2: Domain-Specific URLLC QoS Requirements and Budgets.....	27
Table 2.3: QoS Requirements for Delay-Sensitive Applications.	28
Table 3.1: Model Rational and Fit (Used on Packet/Flow/KPI Datasets).	58
Table 3.2: Capabilities Matrix (Network Traffic Dataset).....	59
Table 3.3: Model Utilisation in This Study.	60
Table 4.1: How Our Approaches Diverge From Existing Models.	76
Table 4.2: Experimental Configuration for Scheduling Trials.	80
Table 4.3: Collected Data Output for Experiment 2.	80
Table 4.4: Framework for Implementing ML Models for QoS Dataset Analysis.	93
Table 4.5: Conceptual Framework for Comparing Scheduling Algorithms Using ML Models.	95
Table 4.6: Network Traffic Data for Datasets	98
Table 4.7: Applications Associated with the Datasets.....	99
Table 4.8: Implementation of ML models for network traffic classification in MATLAB.	100
Table 4.9: Components of a regression model.	102
Table 4.10: Structure of CNN DL Model for Image Dataset of Network Traffic.	106
Table 4.11: CNN Training Process.....	108
Table 4.12: Implementation Steps for LSTM-Based Model for Scheduling Algorithm Classification.	112
Table 4.13: Key Architectural Features of the Three Models Compared.....	118
Table 4.14: Mathematical notations and definitions.....	119
Table 4.15: Specification of Devices in the Network Architecture.	126
Table 4.16: Transformer-based RL Model Configuration and Parameters.	128
Table 5.1: Proportional Fair Scheduling Dataset Overview.	136
Table 5.2: Summary of Continuous Features for the PF Dataset.	137
Table 5.3: Round-Averaged Performance by WLAN Station with PF Scheduler.	137
Table 5.4: Final Experiment Summary under Proportional Fair Scheduling.	139
Table 5.5: Round Robin Scheduling Dataset Overview.	149
Table 5.6: Summary of Continuous Features for the RR Dataset.	150
Table 5.7: Round-Averaged Performance by WLAN Station under RR Scheduling.	150

Table 5.8: Final Experiment Summary under Round Robin Scheduling algorithm.....	152
Table 5.9: Comparative analysis of the PF and RR scheduling disciplines.	160
Table 5.10: Descriptive Statistics for Numeric Attributes in the PRIO Dataset.....	164
Table 5.11: Protocol Distribution for the PRIO Dataset.....	164
Table 5.12: Comparative analysis between Optimisable Tree and Fine Gaussian SVM on the PRIO Dataset.	179
Table 5.13: Descriptive Statistics for Numeric Attributes in the HTB Dataset.....	181
Table 5.14: Protocol Distribution for the HTB Dataset.	181
Table 5.15: Comparative Analysis between the Optimisable Tree vs Fine Gaussian SVM and Fine Tree on the HTB Dataset.	188
Table 5.16: Descriptive Statistics for Numeric Attributes in the CBQ Dataset.....	189
Table 5.17: Protocol Distribution for the CBQ Dataset.....	190
Table 5.18: Comparative ML Model Performance by Protocol (TPR and AUC Metrics) for the CBQ Dataset.	197
Table 5.19: Dataset-Specific Performance Comparison for the Optimisable Tree.	199
Table 5.20: Classification Report for LSTM model.	220
Table 5.21: DL Models (CNN, CNN-LSTM, and LSTM) Comparison for Network QoS Classification.	222
Table 5.22: Performance of Each Model on the Held-out Validation Set (Unseen Devices).	236
Table 5.23: Comparison of the results of the Transformer-based RL scheduler with similar works.	249
Table 5.24: Performance of the Proposed Transformer-Based RL Scheduler Compared to 5G Industry Latency Standards.	251
Table 5.25: Statistical Summary of Latency Performance per Network Segment.	261
Table 5.26: Unified Dataset Summary.	265
Table 5.27: Unified Summary of Best-Performing Models and Key Metrics Across All Experiments.	269
Table 6.1: EGPR Model Performance Comparison.....	274
Table 6.2: Framework for Comparing Scheduling Algorithms Using Statistical Analysis and ML Models.	276
Table 6.3: EGPR Model Evidence of Stability for the PF Scheduling Discipline.	277
Table 6.4: Classification Performance and Observations across QoS Scheduler-Specific Dataset (PRIO, HTB, and CBQ).	282
Table 6.5: Insights into Scheduler Behaviour Learned by ML Models.....	284
Table 6.6: Stakeholder Perspective and Recommended Focus Areas for QoS Classification and Telemetry.	307

Table C.1: Rationale for Using CNN for Scheduling Algorithm Classification.	334
Table C.2: Rationale for Using LSTM with image-based representation of network data.	338
Table D.1: Data Preparation Workflow for Decision Tree Models.	341
Table D.2: Steps in Decision Tree Model Building.	341
Table D.3: Decision Tree Model Evaluation Steps.	342
Table D.4: Steps in Decision Tree Model Hyperparameter Tuning.	342
Table D.5: Final Steps in Evaluating the Decision Tree Model.	343
Table D.6: Data Preparation Workflow for EGPR Models.	343
Table D.7: Steps in EGPR Model Building.	344
Table D.8: EGPR Model Evaluation Steps.	344
Table D.9: Steps in EGPR Model Hyperparameter Tuning.	345
Table D.10: Final EGPR Model Evaluation Steps.	345
Table D.11: Data Preparation Workflow for SVMs Models.	346
Table D.12: Steps in SVMs Model Building.	347
Table D.13: SVM Model Evaluation Steps.	347
Table D.14: Steps in SVM Model Hyperparameter Tuning.	348
Table D.15: Final SVM Model Evaluation Steps.	348
Table D.16: Data Preparation Workflow for Ensemble Models.	349
Table D.17: Steps Involved in Base Learner Selection for Ensemble Models.	349
Table D.18: Steps in Ensemble Model Building.	350
Table D.19: Ensemble Model Evaluation Steps.	350
Table D.20: Steps in Ensemble Model Hyperparameter Tuning.	351
Table D.21: Final Ensemble Model Evaluation Steps.	351
Table D.22: Data Preparation Workflow for LSTM Models.	352
Table D.23: Steps in LSTM Model Building.	352
Table D.24: Steps in LSTM Model Training.	353
Table D.25: LSTM Model Evaluation Steps.	353
Table D.26: Steps in LSTM Model Hyperparameter Tuning.	354
Table D.27: Final LSTM Model Evaluation Steps.	354
Table D.28: Data Preparation Workflow for CNN Models.	355
Table D.29: Steps in CNN Model Building.	356
Table D.30: Steps in CNN Model Training.	356
Table D.31: CNN Model Evaluation Steps.	357

Table D.32: Steps in CNN Model Hyperparameter Tuning.	357
Table D.33: Final CNN Model Evaluation Steps.	358
Table D.34: Data Preparation Workflow for Transformer Neural Network Models. ..	358
Table D.35: Steps in Transformer Neural Network Model Building.	359
Table D.36: Steps in Transformer Neural Network Model Training.....	360
Table D.37: Transformer Neural Networks Model Evaluation Steps.....	360
Table D.38: Steps in Transformer Neural Network Model Hyperparameter Tuning...	361
Table D.39: Final Transformer Neural Network Model Evaluation Steps.....	361
Table D.40: Fundamental Steps in RL Model Implementation.....	361

List of Abbreviations / Glossary of Terms

3 rd Generation Partnership Project (3GPP)	Earliest Deadline First (EDF)
5G New Radio (NR)	end-to-end (E2E)
5G QoS Identifier (5QI)	enhanced mobile broadband (eMBB)
Address Resolution Protocol (ARP)	European Telecommunications Standard Institute (ETSI)
Advanced Actor-Critic (A2C)	Extended Reality (XR)
Application-Specific Integrated Circuits (ASICs)	False Discovery Rate (FDR)
Area Under the ROC Curve (AUC-ROC)	False Negative Rate (FNR)
Artificial Intelligence (AI)	Federated Deep Reinforcement Learning (FDRL)
Artificial Intelligence for IT Operations (AIOps)	Federated Learning (FL)
Augmented Reality (AR)	Fifth-generation (5G)
Central Processing Units (CPUs)	Gaussian Process Regression (GPR)
Channel State Information (CSI)	Gaussian Processes (GPs)
Class-Based Queuing (CBQ)	General Data Protection Regulation (GDPR)
Continuous Deployment (CD)	Google Remote Procedure Call (gRPC)
Continuous Integration (CI)	Graph Neural Network (GNN)
Convolutional Neural Network (CNN)	Graphics Processing Unit (GPU)
Cooperative Awareness Message (CAM)	Health Insurance Portability and Accountability Act (HIPAA)
Cumulative Distribution Function (CDF)	Hierarchical Token Bucket (HTB)
Customer Premises Equipment (CPEs)	Hybrid ARQ (HARQ)
Decentralised Environmental Notification Message (DENM)	HyperText Transfer Protocol (HTTP)
Deep Deterministic Policy Gradient (DDPG)	Intelligent Transportation Systems (ITS)
Deep Learning (DL)	Internet Control Message Protocol (ICMP)
Deep Neural Network (DNN)	Internet Control Message Protocol version 6 ICMPv6
Deep Q-Networks (DQN)	(ICMPv6)
Deep Reinforcement Learning (DRL)	Internet Engineering Task Force (IETF)
Domain Name System (DNS)	Internet of Vehicles (IoV)
	Internet Protocol (IP)

Interquartile Range (IQR)	Radio Frequency (RF)
Key Network Performance Indicators (KPIs)	Receiver Operating Characteristics (ROC)
Light Detection and Ranging (LIDAR)	Red, Green, and Blue (RGB)
Local Interpretable Model-Agnostic Explanations (LIME)	Reinforcement Learning (RL)
Long Short-Term Memory (LSTM)	Recurrent Neural Network (RNN)
Machine Learning (ML)	Remote Procedure Calls (RPCs)
Markov Decision Process (MDP)	Representational State Transfer (REST)
massive machine-type communications (mMTC)	Road Side Units (RSU)
Mean Absolute Error (MAE)	Root Mean Squared Error (RMSE)
Mean Squared Error (MSE)	Round Robin (RR)
Mobile Edge Computing (MEC)	Round-Trip Time (RTT)
Multicast Domain Name System (MDNS)	Security Information and Event Management (SIEM)
Network Function Virtualisation (NFV)	Self-Optimising Network (SON)
Network Operations Centre (NOC)	Service Level Agreement (SLAs)
Online Certificate Status Protocol (OCSP)	Shapley Additive Explanations (SHAP)
Open Network Automation Platform (ONAP)	Signal to Noise Ratio (SNR)
Open Neural Network Exchange (ONNX)	Simple Service Discovery Protocol (SSDP)
O-RAN Service Management and Orchestration (SMO)	Software-Defined Networking (SDN)
Physical Resource Blocks (PRBs)	Stochastic Fair Queuing (SFQ)
Positive Predictive Value (PPV)	Stochastic Gradient Descent with Momentum (SGDM)
Principal Component Analysis (PCA)	Support Vector Machines (SVMs)
Priority queuing (Prio)	The Software Development Kit (SDK)
Proportional Fair (PF)	Time-Aware Shaper (TAS)
Proximal Policy Optimisation (PPO)	Time-Sensitive Networking (TSN)
Quality of Service (QoS)	Time-to-Live (TTL)
Quick UDP Internet Connections (QUIC)	Transmission Control Protocol (TCP)
Radial Basis Functions (RBF)	Transmission Time Interval (TTI)
radio access network (RAN)	Transport Layer Security (TLS)
	True Positive Rate (TPR)
	Twin Delayed Deep Deterministic Policy Gradient (TD3)

Ultra-Reliable Low-Latency
Communications (URLLC)
Upper Confidence Bound (UCB)
User Datagram Protocol (UDP)
User Plane Function (UPF)
vehicle-to-everything (V2X)

Virtual Private Clouds (VPCs)
Virtual Reality (VR)
Voice over Internet Protocol (VoIP)
Weighted Fair Queuing (WFQ)
Wireless Local Area Network (WLAN)
Wireless Sensor Networks (WSNs)

Chapter 1: Introduction

1.1 Introduction

The deployment of 5G networks promises not only enhanced mobile broadband (eMBB) but also massive machine-type communications (mMTC) and Ultra-Reliable Low-Latency Communications (URLLC) to support the connectivity of billions of IoT sensors and devices. Wireless sensor networks are poised to leverage 5G's high bandwidth and low latency to transmit real-time IP packets, such as time-critical sensor data, video feeds, or control signals, with stringent QoS requirements. Emerging applications in telemedicine, autonomous vehicles, and smart factories demand extremely low end-to-end latencies without sacrificing reliability. Ensuring such QoS for real-time IP packets in 5G-integrated WSNs is paramount: delayed or lost sensor data in such scenarios could degrade performance or pose safety risks [1]-[4].

Achieving high QoS for real-time WSN traffic over 5G, however, is a challenging task. The sheer scale of connected IoT devices, the variability of wireless channels, and the heterogeneity of traffic types (e.g., telemetry vs. bursty emergency messages) can strain network resources [1]-[4]. Conventional scheduling and resource management techniques often struggle to adapt to these highly dynamic conditions. This has spurred intense research into more intelligent resource allocation and traffic prioritisation schemes. In particular, artificial intelligence (AI) and machine learning (ML) are increasingly seen as key enablers for dynamic network optimisation, enabling more intelligent resource allocation, real-time IP traffic management, and predictive network adjustment to maintain QoS [1]-[4]. This research study aims to investigate, identify, review, and evaluate QoS issues, especially end-to-end latency and reliability, for 5G WSNs, and develop novel AI-driven mechanisms to improve QoS for emerging applications that require URLLC in the WSN domain.

1.2 Context and Background

1.2.1 5G WSN Overview and Key Characteristics

5G WSNs are sensor systems that utilise 5G cellular connectivity to transmit data. Compared to earlier wireless generations, 5G networks introduce several transformative features that greatly enhance WSN performance. The three fundamental service categories defined for 5G are eMBB, URLLC, and mMTC [1], [2], [3], [4]. Each of these services corresponds to key performance characteristics that benefit WSN applications:

1) URLLC: This 5G capability was designed to provide end-to-end latencies around 1 millisecond and reliability on the order of 99.999% or higher for packet delivery [5], [6]. Additionally, this extreme performance is crucial for real-time control and safety applications (e.g., wireless industrial control loops, remote surgery, and vehicle collision avoidance). For instance, the 3GPP standards (Rel. 15/16) set goals of < 1 ms one-way radio latency and packet error rates as low as 10^{-5} (99.999% success) for URLLC traffic. Achieving URLLC entails design features such as short transmission time intervals,

grant-free scheduling, and packet duplication to eliminate delays and packet losses. This level of ultra-reliability and latency is unprecedented in prior networks, enabling WSNs to be used in cases where even a few milliseconds of delay or a single dropped packet could be catastrophic (e.g., emergency brake signals in vehicles or haptic feedback in telesurgery) [5]-[7].

2) mMTC: The 5G network was designed to connect an enormous number of devices efficiently. It can support up to one million devices per square kilometre, far beyond the capacity of 4G/LTE or Wi-Fi networks. This massive connectivity is crucial for large-scale WSN deployments such as smart cities, environmental sensor grids, or extensive IoT systems. Each sensor or device may only send small amounts of data (bytes or kilobytes); however, the network must handle many simultaneous connections with low overhead. 5G achieves this through features such as narrowband IoT modes, improved random access, and efficient signalling. Massive machine-type communication enables the scalability of WSNs, allowing thousands or millions of sensors to share the 5G network reliably without congestion. For instance, 5G can incorporate NB-IoT and LTE-M (Cat-M1) technologies (specified by 3GPP) to support low-power wide-area sensor connectivity with extended coverage and battery life, complementing mainstream 5G New Radio (NR) for higher-rate devices [6]-[8].

3) eMBB: Meanwhile, many sensors only need to send small telemetry packets; however, some WSN applications involve high-bandwidth data streams, such as video, high-resolution images, or augmented reality feeds. The 5G's eMBB mode delivers fibre-like speeds wirelessly – multi-gigabit-per-second throughput is achievable (with peak rates over 10 Gbps under ideal conditions) [9], [10], [11]. Even in realistic deployments, eMBB can consistently provide hundreds of Mbps. This enables use cases such as real-time ultra-HD video transmission from remote sites (e.g. an ambulance or a surveillance drone), or rapid downloading of large sensor datasets. In the WSN context, eMBB can support data-intensive sensors (e.g., streaming 4K/8K video from a medical camera or a machine vision system on a factory floor). Prior wireless networks struggled with such loads; thus, 5G's high capacity and spectral efficiency (through massive MIMO, millimetre-wave bands, etc.) make high-bandwidth sensor data feasible [9]-[11].

1.2.2 5G QoS Architecture and WSN Integration

Fifth-generation (5G) wireless technology is revolutionising WSNs by providing unprecedented performance in terms of packet transmission speeds, latency, throughput, connectivity, and reliability [12]-[14]. Traditional WSNs – networks of spatially distributed sensors that communicate wirelessly – have historically been constrained by limited bandwidth and high latency, often relying on short-range protocols (such as Zigbee or Wi-Fi) for connectivity. With the introduction of 5G, WSNs can be embedded into cellular networks that support massive device connectivity and real-time data delivery. The integration of 5G with WSNs (often referred to as 5G WSNs) presents significant opportunities for new applications in autonomous transportation, remote healthcare, and industrial automation that require stringent QoS guarantees. Furthermore, 5G WSNs exhibit key characteristics such as ultra-low latency, ultra-high reliability, and

massive connectivity, which are paramount for time-critical and mission-critical sensor data applications. This ability to deliver differentiated performance is managed by 5G's native flow-based QoS architecture [12]-[14].

The 5G network incorporates a robust QoS architecture that supports diverse service requirements. Data traffic in 5G is carried over QoS flows identified by a 5G QoS Identifier (5QI) – an index that maps to a set of QoS characteristics (such as priority level, packet delay budget, packet error rate, etc.) [12]-[14]. Unlike legacy networks, 5G's QoS model is flow-based and flexibly configurable; by specifying a 5QI, the network implicitly knows how to treat that traffic in terms of scheduling priority, acceptable latency, and reliability. For instance, conversational voice is standardised as 5QI=1 with 100 ms delay budget and high priority, whereas real-time gaming may use 5QI=3 with a tighter 50 ms delay budget but slightly lower priority. These QoS parameters directly influence how the 5G radio access network (RAN) schedules packets, affecting scheduling weights, queue management thresholds, and admission control. For 5G WSNs, this architecture is crucial, as it enables data from a mission-critical industrial sensor to be assigned a high-priority 5QI, ensuring guaranteed low latency. In contrast, routine telemetry from non-critical sensors can be assigned a lower-priority, best-effort 5QI.

Wireless sensor networks in the 5G era typically fall under the mMTC and URLLC use cases. Many IoT-oriented devices may connect through 5G NR interfaces (including NB-IoT or other 5G IoT modes) or via local gateways. The integration of WSNs with 5G enables sensor data to be transmitted via the Internet Protocol (IP) over the cellular network, benefiting from 5G's wide-area coverage and QoS guarantees. Nonetheless, WSN nodes often have constrained energy and may generate bursty traffic (e.g., event-triggered alerts) alongside steady flows (e.g., periodic measurements). This heterogeneity needs careful resource allocation on the network side. The 5G network slicing is one mechanism to handle this: a specific slice can be configured for IoT sensor traffic, isolating resources and customising QoS parameters to meet that slice's requirements [12]-[14]. Each slice or service may use a different 5QI mapping to ensure, for example, that an urgent alarm signal from a sensor receives high-priority, low-latency treatment, whereas routine sensor reports use a lower priority.

1.2.3 Resource Allocation and Prioritisation in 5G

In the RAN (Radio Access Network), resource allocation refers to the assignment of time-frequency resources, such as Physical Resource Blocks (PRBs) and timeslots, as well as transmission parameters (including modulation and coding), to users and data flows. Prioritisation involves assigning preferential scheduling to specific traffic flows or devices based on their QoS requirements. The 5G NR's flexible frame structure (with scalable subcarrier spacing and mini-slot scheduling) was designed to accommodate low-latency scheduling for urgent traffic [15], [16]. For instance, URLLC transmissions can utilise eMBB resources within a subframe to meet latency demands, and mini-slots (as short as two symbols) enable the immediate scheduling of high-priority traffic without waiting for a full slot [15], [16]. Along with this, the 5G-core network manages QoS by

mapping flows to appropriate bearers and enforcing policies; for instance, packets that violate the QoS flow's packet delay budget might be considered lost for URLLC services, effectively tightening reliability standards for real-time traffic [15], [16].

Furthermore, in a 5G-connected WSN scenario, consider a mix of traffic: sensor streams for critical control (real-time) and sensor updates for logging (non-real-time). The network must allocate radio resources so that critical streams always meet their deadlines. This could entail applying priority scheduling algorithms at the base station (gNodeB). Traditional schedulers such as Proportional Fair (PF) or Round Robin (RR) alone are insufficient to guarantee strict delay bounds [15], [16]. Enhancements such as deadline-aware scheduling or priority queuing are applied, for instance, by using weighted-priority queues that serve IP packets from sensors first. Admission control and bandwidth reservation might be used - the network can reserve a portion of bandwidth for high-priority IoT traffic (possibly via a dedicated slice or QoS class) to ensure availability during congestion periods. All these mechanisms form the background toolkit for QoS management in 5G WSNs [15], [16].

1.3 Motivation and Relevance

The imperative to integrate AI into 5G-enabled WSNs stems from the inherent complexities and stringent demands of emerging applications. Meanwhile, 5G provides native support for URLLC; however, the dynamic, resource-constrained nature of WSNs poses significant hurdles to achieving the required QoS. Traditional network management schemes are proving insufficient for the complexities of 5G. According to Qian et al. [17], there is an imperative necessity to overcome the inefficiencies of traditional optimisation methods in UAV-aided 5G/6G networks and enable intelligent, adaptive, and efficient QoS management through DRL. The relevance of this work stems from its ability to overcome the computational inefficiencies of traditional optimisation methods in dynamic UAV-aided 5G/6G environments. By employing a hybrid framework that integrates nonlinear optimisation for path planning with Multi-Agent DRL (MADRL), the work offers an intelligent, adaptive solution capable of handling non-convex problems in real time. This approach significantly enhances critical QoS metrics, specifically energy efficiency and latency, providing a robust resource management strategy for safety-critical network applications.

The integration of 5G-enabled Internet of Vehicles (IoV) is becoming indispensable for intelligent transportation systems, offering the high bandwidth and low latency required for real-time traffic scheduling and smart city management. However, the reliability of these systems is frequently compromised by data sparsity, where sensor failures and transmission exceptions create incomplete datasets that hinder accurate traffic analysis and decision-making. To overcome these limitations, Hu et al. [18] propose a Digital Twin-assisted framework that utilises Locality-Sensitive Hashing (LSH). By creating virtual replicas of physical vehicles, this method effectively reconstructs missing data patterns to predict traffic flow and velocity in real time, ensuring system stability despite gaps in physical sensors.

With the rapid evolution towards Beyond 5G (B5G) networks, the demand for highly flexible and efficient resource management has intensified as the diversity of services, devices, and QoS requirements grows. Traditional, static, rule-based network management schemes are increasingly inadequate for handling the complexity and dynamic nature of B5G environments. Motivated by this challenge, Aloupogianni et al. [19] propose an ML-driven framework for network slicing, leveraging data-driven models, particularly neural networks, to automate slice optimisation and allocation. Their approach aims to enhance scalability, adaptability, and resource utilisation efficiency across large, heterogeneous communication networks, providing a practical path toward intelligent, self-optimising network infrastructures.

The rise of 5G-enabled WSNs introduces new demands for ultra-reliable, low-latency, and energy-efficient data transmission. Traditional routing protocols fail to adapt effectively to the dynamic, data-intensive nature of modern sensor environments. Motivated by these challenges, Vijayakumar et al. [20] review recent advances in DL-based routing schemes, emphasising their potential to enable intelligent, context-aware, and adaptive network management. The authors propose that leveraging deep neural models can revolutionise routing efficiency by providing autonomous decision-making, resilience to environmental changes, and optimised energy utilisation across large-scale 5G WSN deployments.

The sheer scale and heterogeneity of devices in mMTC, coupled with the critical nature of applications in Industry 4.0, smart healthcare, and autonomous vehicles, demand a more intelligent and adaptive approach to network management. AI and ML offer the potential to move beyond reactive, rule-based systems to predictive and proactive network optimisation, which is crucial for realising the full potential of 5G in WSNs [17]-[19].

This research is highly relevant as it directly confronts the critical challenges hindering the widespread deployment and optimal performance of 5G WSNs. By designing and developing AI-driven solutions, this research study aims to provide the mechanisms for intelligent resource allocation, dynamic traffic management, and adaptive network slicing, which are crucial for meeting the diverse and often conflicting QoS requirements of various applications. For example, AI algorithms can analyse vast amounts of data in real time to classify and predict traffic patterns, identify potential congestion, and proactively adjust resource distribution or traffic prioritisation to maintain low latency and high reliability. The application of AI extends to enhancing the energy efficiency of WSN nodes, ensuring seamless mobility management, and addressing key operational and scalability challenges. Ultimately, this investigation is motivated by the transformative potential of creating autonomous and self-optimising 5G WSNs that can intelligently adapt to their environment and the specific needs of mission-critical applications, paving the way for a new era of reliable and efficient wireless connectivity [17]-[20].

1.3.1 Ultra-Low Latency Requirements

WSN traffic requiring URLLC demands that the network latency be pushed to unprecedented lows. In 5G NR, a minimum slot duration of 0.25 ms is defined for sub-6 GHz bands; however, a single packet often spans multiple slots due to scheduling and processing, making it extremely challenging to achieve 1 ms end-to-end delays [7], [10], [17]. While millimetre-wave (mmWave) frequencies allow for much shorter transmission slots ($\sim 15 \mu\text{s}$), they introduce reliability issues. Recent 5G testbed measurements found that sub-millisecond latencies in mmWave 5G systems were achieved only $\sim 4.4\%$ of the time – far below the $> 99.999\%$ reliability target for URLLC. This highlights the challenge of consistently meeting the URLLC latency requirements in real-world applications. Several factors contribute to latency, including protocol overhead (scheduling requests and HARQ feedback), processing delays in devices and base stations, and radio propagation delays [7], [10], [17]. For instance, if any of these sources is not carefully minimised, it can bottleneck the end-to-end delay. Furthermore, latency and reliability goals often conflict. While techniques such as using very short packets or grant-free access can reduce delay, they usually do so at the expense of coding and diversity gains, undermining reliability [7], [10], [17]. In contrast, schemes used to boost reliability (such as re-transmissions or robust encoding) incur extra delay. The goal is to achieve ultra-low latency consistently, not just on average; network practitioners emphasise that maintaining deterministically low latency (with minimal jitter) is necessary for mission-critical control [7], [10], [17]. This may require over-provisioning and redundancy (e.g., denser cell sites and duplicate links), as the delay variance of any single network element can exceed the 1 ms budget [7], [10], [17]. In summary, minimising latency in 5G WSN is a cross-layer challenge that demands streamlined protocols, edge computing for local processing, and agile scheduling – all while preserving near-perfect reliability.

1.3.2 Reliability and Wireless Channel Challenges

The URLLC's reliability target (e.g. $< 10^{-5}$ packet error rate) is orders of magnitude stricter than typical mobile broadband. Ensuring such reliability in a wireless sensor context is challenging due to time-varying channel conditions, signal interference, and hardware limitations. Wireless links are prone to fading and shadowing, which can be aggravated in specific environments. For instance, in industrial IoT scenarios such as factories, the Radio Frequency (RF) environment is highly dynamic – moving robotic arms, machinery, and human workers constantly alter propagation conditions [11]. This dynamism can cause sudden drops in signal quality, making it challenging to consistently meet the “ultra-reliable” criterion.

Real-world measurements indicate that current networks struggle to address these challenges. The study by Caso et al. [21] provides evidence revealing that, contrary to the expectation of network maturity, 5G Non-Standalone (NSA) performance has shown signs of degradation over time. They highlight that wireless channel variability and handover robustness remain key obstacles to achieving consistent and ultra-reliable connectivity, insights directly relevant to ongoing 5G-to-6G reliability research.

To mitigate these issues, techniques such as multi-connectivity (sending duplicate packets over multiple paths) can be employed. However, these methods are resource inefficient and require careful optimisation to avoid spectrum wastage. Furthermore, in densely deployed sensor networks, simultaneous transmissions create significant interference, degrading packet delivery rates [11], [21].

Addressing these fundamental limitations requires a new approach. According to Mahmood et al. [22], the randomness of wireless channels and electromagnetic interference constitute the primary bottleneck for achieving ultra-reliability in critical industries. To address this, the authors propose a paradigm shift from reactive adaptation to proactive environment management, advocating a special-purpose 6G architecture that creates deterministic channels through AI-driven prediction and active propagation control using technologies such as Reconfigurable Intelligent Surfaces (RIS).

1.3.3 Core Resource Management Constraints

The challenge of delivering URLLC is compounded by fundamental constraints at the device, network, and environmental levels, turning it into a complex resource management problem.

At the device level, energy efficiency is a primary constraint. Many WSN nodes are battery-powered, yet URLLC's "always-on" requirement directly conflicts with energy-saving sleep modes. To meet the stringent latency bounds, a device must stay alert, poll the network frequently, and often transmit at higher power, drastically shortening its operational lifetime. This creates a critical trade-off: the computational and transmission power required for high reliability directly undermines the sensor node's energy budget. This problem is further exacerbated by mobility. When nodes move, the standard handover procedure between cells introduces delays and packet loss that violate URLLC guarantees. While solutions such as make-before-break or multi-connectivity can mitigate this, they are inherently resource-intensive, consuming more power and spectrum to maintain a seamless link [3], [5], [23].

At the network level, scalability and interference pose a significant challenge. 5G WSNs in scenarios such as factories or highways must support a dense population of devices, resulting in intense competition for radio resources. Scheduling conflicts arise when multiple devices demand immediate, guaranteed bandwidth, forcing the network into complex prioritisation schemes, often at the expense of other traffic, such as mobile broadband. This density naturally creates a harsh RF environment where interference, from co-existing wireless systems (e.g. Wi-Fi) and other 5G traffic can degrade packet delivery and force retransmissions, jeopardising reliability and latency [5], [23].

Fundamentally, these competing demands for energy, seamless mobility, and interference-free scalability mean that simply over-provisioning resources is often impractical or cost-prohibitive. Consistently meeting URLLC targets requires more than just raw network capacity; it demands a sophisticated and dynamic approach to resource allocation and prioritisation to navigate these conflicting constraints intelligently. In this context, Wang et al. [23] specifically address these management complexities by

proposing a dynamic resource allocation framework that integrates traffic prediction with RL. By utilising a hybrid Convolutional Neural Network (CNN) and Bi-LSTM (Long Short-Term Memory) model to forecast traffic fluctuations and a Duelling Double DQN algorithm to optimise the distribution of physical resource blocks (PRBs), their work demonstrates how delay-tolerant and delay-sensitive traffic can be balanced to minimise resource consumption while strictly adhering to QoS constraints.

1.3.4 Application Domain and Use Cases

The aforementioned difficulties have tangible consequences across diverse application areas. Below, we focus on three key domains – industrial IoT, autonomous vehicles, and smart healthcare – and highlight their specific requirements and challenges in the context of 5G WSNs with URLLC capabilities.

1) Industrial IoT and Industry 4.0: Modern industrial automation (Industry 4.0 or “smart factories”) is a prime use case for URLLC-enabled WSNs. In these environments, wireless sensors and actuators work together to control machinery with stringent timing requirements. For instance, motion control loops on assembly robots may require end-to-end latency around 1 ms and extremely high reliability (error rates on the order of 10^{-5}) [24]-[26]. Tasks such as closed-loop control of robotic arms, real-time process monitoring, or factory safety shutoff systems demand such performance, and achieving this over wireless links is challenging. One issue is the heavy multipath and obstruction in factories – metal equipment and moving parts create a harsh radio environment that can cause rapid fading. As noted, the RF is highly dynamic; system reliability becomes a challenge when robotic components or vehicles on the factory floor continuously block or unblock signal paths. Ensuring that every critical command or sensor reading reaches its destination on time may require redundant communication methods. Some factories are exploring multi-connectivity (where each sensor node connects to two or more access points or base stations) to tackle wireless drops. Others consider using local 5G networks in combination with wired connections for fail-safe operations [24]-[26].

Managing device concentration and data communication patterns poses an additional challenge in IIoT environments. A production facility may have thousands of sensors (such as temperature, pressure, and vision cameras) and actuators, many of which may transmit data simultaneously or in bursts. Even though each URLLC packet might be small, the aggregate traffic can be significant, and it often arrives in a sporadic, bursty manner (for instance, an alarm system may be mostly silent but then send a flood of alerts during an incident). This unpredictable, non-streaming traffic is hard for traditional networks to schedule efficiently [24], [25], [26]. It can lead to momentary congestion, which is problematic for latency. Thus, techniques such as network slicing can help by reserving capacity for critical industrial traffic, and massive MIMO can increase overall capacity. However, effectively managing bursts of URLLC traffic remains an area of active research. Many industrial IoT devices were historically connected through wired networks (for reliability). Transitioning to wireless 5G WSNs raises concerns about reliability and security, as factories require assurance that wireless links can be as dependable as wired cables. This has sparked interest in private 5G networks for industry

– dedicated on-site 5G installations that can be tailored and controlled to meet the factory’s specific needs (limiting outside interference, customising protocols, etc.). The deployment of such networks involves significant cost and technical expertise, which can be a barrier to adoption [24]-[26].

Furthermore, security in industrial URLLC applications is also paramount. An attack or breach of a factory’s sensor network could lead to equipment damage, worker safety hazards, or costly downtime. The infamous TRITON malware incident (targeting safety control systems in critical infrastructure) underscores the stakes. In a 5G-enabled factory, each critical system (such as sensors and actuators) must be authenticated and protected. Features such as network slicing provide isolation, for example, by separating the factory control slice from any general corporate network slice, thereby reducing risk [24]-[26]. However, authenticating and controlling access to potentially thousands of IoT devices is a daunting task. Managing device credentials, secure onboarding (perhaps through zero-touch provisioning), and regular security updates are all challenges that scale with the number of devices. These operational security issues are as vital as the communication security itself.

To overcome the limitations discussed in point 1, Do-Duy et al. [24] propose a digital twin-assisted framework for MEC. It addresses latency in industrial automation by using a digital twin to estimate network status and edge server computing resources in real time. This allows IoT devices to make intelligent offloading decisions, selecting the best edge server to minimise end-to-end latency while adhering to energy and resource constraints.

Kim et al. [25] proposed a DRL framework for Wireless Time-Sensitive Networking that adaptively schedules transmissions in real-time to overcome the unreliability caused by channel fluctuations.

To solve the latency and scalability bottlenecks of using DL for real-time control, Gallup et al. [26] propose a novel “Encoder-Decoder Vision Transformer” (EDVT) architecture for real-time optimisation and model predictive control (MPC). It modifies the standard attention scheme to handle spatiotemporal data (inspired by vision transformers) and was tested on data from a 450 MW coal-fired plant. The authors report that their model achieved an 83% reduction in Mean Squared Error (MSE) and an 84% reduction in computation time compared to conventional Recurrent Neural Networks (RNNs) such as LSTMs and GRUs.

In summary, industrial IoT stands to gain enormously from URLLC (flexible wireless reconfiguration of production lines, real-time analytics, etc.). Nonetheless, it requires solutions to address reliability in dynamic environments, network scaling, and stringent security to truly realise the wireless factory of the future. This underscores the need for dynamic resource allocation and traffic prioritisation tools capable of managing bursty traffic and reliability in harsh RF conditions [24]-[26].

2) Smart Healthcare and Medical IoT: The healthcare sector is positioned to leverage the capabilities of 5G WSNs featuring URLLC. This will facilitate a range of advanced applications, including telesurgery, remote patient observation, intelligent emergency

vehicles, and augmented reality systems for clinicians. The common theme is that lives may be on the line – thus, reliability and low latency are non-negotiable. For example, in telesurgery, a surgeon could remotely control robotic instruments; any perceptible lag or dropped command could endanger a patient. Such applications typically require one-way latencies on the order of a few milliseconds and virtually zero packet loss. 5G URLLC has been envisioned to meet these requirements (e.g. enabling a surgeon to operate on a patient in a distant location with instantaneous haptic feedback) [27]-[29].

Nevertheless, to date, these remain experimental due to the extreme network demands and the need for certified reliability. Even in less extreme cases, such as streaming high-definition video from an ambulance to a hospital (for diagnostic help en route), networks can be taxed – they require a combination of high throughput and low latency. Edge computing is expected to play a pivotal role here: by processing data (such as preliminary image analysis or vital sign anomaly detection) at the edge of the network (e.g., a hospital’s 5G edge cloud), the system can respond faster and alleviate backhaul load [27]-[29]. Indeed, industry initiatives are developing medical edge clouds tailored to hospital needs, effectively bringing cloud capabilities on-site to meet URLLC and privacy requirements.

Deployment challenges in healthcare include ensuring coverage in varied environments. For instance, urban hospitals might get 5G small cells indoors for dedicated coverage; however, rural healthcare centres may not yet have 5G access. Recent studies have noted that deploying 5G in rural healthcare can dramatically reduce latency and improve service quality; nonetheless, current deployments are still in their early stages [27]-[29]. Only a small fraction of smart healthcare projects so far have fully operational 5G implementations, underscoring that the technology is still in early adoption. For widespread telehealth, consistent network availability is as big a challenge as the raw latency metric. Another issue is device heterogeneity: medical IoT devices range from wearable sensors and implants to large imaging machines. Many are power-constrained (e.g., an implantable heart monitor cannot have a large battery or be easily recharged). The previously mentioned energy efficiency challenges are particularly pronounced in healthcare. Designers have to find a balance between keeping the device responsive (for URLLC) and preserving battery life to avoid frequent surgeries or replacements. Some solutions involve using the 5G link sparingly (only for critical or urgent data) while employing local processing for routine monitoring – effectively a tiered approach to communication [27], [28], [29].

Furthermore, privacy and security are perhaps more crucial than anywhere else in healthcare. Patient data is protected by strict regulations [such as the Health Insurance Portability and Accountability Act (HIPAA) in the USA or the General Data Protection Regulation (GDPR) in Europe], and breaches can erode trust and cause harm. In a 5G WSN, patient vital data might be continuously transmitted from body-worn sensors to a hospital’s network. This data must be encrypted and kept confidential. At the same time, doctors need assurances of data integrity – for instance, that an alert about a patient’s heart arrhythmia is genuine and not tampered with. The challenge is implementing robust

encryption and authentication without introducing unacceptable processing delay or power consumption, a core trade-off that any resource allocation framework must consider [27]-[29]. Lightweight encryption algorithms, as well as novel methods such as one-time pad keys derived from physiological signals or channel characteristics, have been explored to secure medical data in transit with minimal delay.

There is also a growing interest in blockchain or distributed ledger technologies for healthcare IoT to ensure an immutable log of data and access, though integrating such technologies with URLLC (which requires rapid processing) is non-trivial. Notably, researchers have identified several unresolved problems and research impasses in 5G smart-healthcare IoT solutions, revealing that many open questions remain in effectively implementing these systems [27]-[29]. These include optimising network architectures for hospital settings, meeting regulatory requirements, and ensuring seamless interoperability between medical devices and 5G networks. In summary, 5G URLLC could enable transformative healthcare applications (smarter ambulances, remote expertise, automated drug delivery systems responding in real-time to sensor data, etc.); nonetheless, significant work is needed to ensure that the network can deliver the required performance securely and reliably, particularly as patient safety is at stake. Effectively balancing these demands requires an intelligent resource management system that can prioritise critical data while respecting strict energy and security constraints [27]-[29].

To address the challenges discussed in this section, Salahdine et al. [27] provide a comprehensive analysis of key technology enablers that collectively mitigate connectivity problems that hinder high levels of QoS and QoE. They highlight the roles of mmWave, network slicing, and MEC in satisfying healthcare's strict demands. Furthermore, they discuss integrating AI to automate network management and enhance security, outlining a roadmap for transitioning from 5G to 6G to support these mission-critical applications.

In the context of smart healthcare and medical IoT, Alhammadi et al. [28] address the problems of latency, reliability, and bandwidth limitations in current wireless networks, which hinder the deployment of critical, real-time medical applications. The authors propose the integration of AI into 6G networks as a solution to enable URLLC. This convergence allows for transformative healthcare applications, including remote robotic surgery, where AI ensures the necessary precision and real-time responsiveness, as well as intelligent health monitoring and holographic telepresence for remote medical consultations.

Medical IoT relies heavily on Software Defined Internet of Things (SD-IoT) architectures to manage the vast connectivity of medical devices (IoMT). Thus, to address the security challenges associated with these network infrastructures, Deepthy et al. [29] tackle the critical problem of Distributed Denial-of-Service (DDoS) attacks that can disrupt vital patient monitoring and data services. Their proposed solution is a regularised hybrid DL model that combines a CNN for feature extraction with a bidirectional LSTM network for temporal sequence learning. By regularising the loss function to prevent overfitting, the model achieves high accuracy (99.8% on the CICDDoS2019 dataset) in detecting and

predicting malicious traffic, thereby safeguarding the availability and reliability of the SD-IoT network infrastructure underpinning smart healthcare systems.

3) Autonomous Vehicles and Intelligent Transportation: Among the most prominent applications of URLLC are autonomous driving and vehicle-to-everything (V2X) communication. In an intelligent transportation system, vehicles, roadside infrastructure, pedestrians' devices, and cloud systems all exchange information to improve safety and efficiency. For instance, vehicles might share sensor data (such as camera or Light Detection and Ranging (LIDAR) feeds) with traffic management systems in real-time. The constraints for latency and reliability are exceedingly rigorous in this application, where a deviation of a few milliseconds can be critical to accident prevention at high velocities [6], [7], [30], [31], [32]. The European Telecommunications Standards Institute (ETSI) for vehicular safety messages [Cooperative Awareness Message (CAM) and Decentralised Environmental Notification Message (DENM)] implicitly demands low latency and high reliability to ensure that vehicles have up-to-date awareness of their environment. The URLLC mode for 5G and cellular V2X protocols is designed to meet some of these needs. Specifically, direct vehicle-to-vehicle communication (the PC5 interface in 3GPP C-V2X) allows low-latency exchange without routing via a base station for nearby vehicles. This is particularly useful in settings such as platooning (the tight coordination of a convoy of vehicles) or collision avoidance, where an end-to-end latency of under 5 ms is required for effectiveness [6], [7], [30], [31], [32].

Nevertheless, achieving URLLC performance in vehicular scenarios faces unique challenges. Mobility is the first challenge – vehicles can move at high speed via coverage areas, as discussed earlier under handover issues. The network must handle Doppler shifts and rapidly changing signal quality. Even beamforming, which is used at higher frequencies to improve link budget, becomes challenging because beams must track moving cars with a very low error margin. A mmWave beam could be lost in an instant if a vehicle turns a corner or an obstacle blocks the line of sight between the transmitter and receiver. While mmWave 5G can theoretically provide very high data rates and low latency, its reliability for vehicles is questionable unless supplemented by lower-frequency links. Practical tests have shown that the ideal sub-millisecond latencies are rarely sustained in mobile mmWave conditions [6], [7], [30], [31], [32]. Vehicles may require multi-link solutions – for instance, utilising sub-6 GHz 5G or even legacy Intelligent Transportation Systems (ITS) bands as a fallback when mmWave connectivity is lost, ensuring continuity.

Network coverage and edge infrastructure are also problematic for wide-area vehicular use. Unlike a factory or hospital, which are confined areas where dedicated infrastructure can be installed, vehicles roam freely over cities and highways. URLLC performance might be attainable in dense urban centres (with many base stations and edge computing nodes); however, along a rural highway, the density of network infrastructure is much lower. As one industry analysis highlighted, truly reliable low-latency service for self-driving cars will require much higher network density and redundancy than many anticipated – including overlapping cell coverage and robust computing at the edge –

which will be expensive and time-consuming to deploy across entire road networks [6], [7], [30], [31], [32]. For the near future, this suggests that autonomous vehicles must be heterogeneous: leveraging 5G connectivity when available, but also able to fall back on on-board sensors and local decision-making when network QoS drops. This graceful degradation is, in fact, a vital safety feature: vehicles should never be solely reliant on real-time network updates for critical manoeuvres. Efforts such as Continental’s “predictive connectivity” concept show how vehicles might anticipate network dead zones and prepare by adjusting their driving strategy or downloading data in advance [6], [7], [30], [31], [32].

From a data perspective, autonomous vehicles generate vast amounts of data from their onboard sensor suites, including cameras, radar, and other systems. Transmitting raw or processed data to the cloud or other cars can easily require both URLLC and high bandwidth, a combination that pushes the limits of 5G. For example, teleoperation of vehicles (remote driving) demands streaming high-definition video upstream from the vehicle with minimal latency, while sending control commands downstream with ultra-low latency [6], [7], [30], [31], [32]. In a trial, even an optimised 4G/5G network achieved a latency of ~ 15 ms for teleoperation and could support only a handful of video streams per cell. Scaling that to a citywide fleet is a challenge for 5G, though improvements such as new spectrum and MIMO will help increase capacity. The reliability of the network link directly affects transportation safety. Service uptime agreements may need to be far stricter than they are today. One can imagine that for future autonomous vehicle services, mobile operators will be required to meet very high uptime and fault-recovery standards (with penalties for outages). This is driving discussions on how to provide redundant paths (e.g., multiple operators or via satellite backup) for vehicular communications. Security is also a concern: a breach of an autonomous vehicle communication system could be catastrophic (e.g., a hacker causing a traffic jam or an accident). Authentication of messages (so a car only trusts messages from legitimate sources) is a key requirement being built into V2X standards. Yet, as noted, doing this without adding latency is tough. Researchers have proposed optimised cryptographic mechanisms for V2X; however, these need to be thoroughly vetted for robustness [6], [7], [30], [31], [32].

Cislaghi et al. [30] address the challenge of validating Tele-Operated Driving (ToD), a service in which remote human operators control vehicles via 5G networks that impose stringent requirements for high bandwidth and ultra-low latency, making real-world testing both difficult and risky. Their solution is a comprehensive co-simulation framework that integrates CARLA (for realistic vehicle dynamics and sensor data generation) with OMNET ++ (for simulating 5G network behaviour), enabling researchers to safely evaluate how network conditions, such as different 5G slice configurations, affect critical safety metrics such as glass-to-glass latency and control reliability.

Sun et al. [31] tackle the challenge of accurately classifying complex time-series data in 6G-supported intelligent transportation systems, where diverse sensors, vehicles, and environments generate data. Traditional DL models often require large, labelled datasets

and fail to generalise well to new, unseen traffic or environmental conditions, limiting their real-world applicability. To overcome this, the authors propose a meta-transfer metric learning (MTML) framework that combines meta-learning and transfer learning to enable fast model adaptation with minimal labelled data. Their method learns generalised feature representations across tasks and uses a metric-based similarity measure to efficiently classify time series. Experiments show that MTML achieves higher accuracy and faster adaptation compared to conventional DL and transfer learning models, making it a promising approach for real-time, data-efficient pattern recognition in next-generation ITS environments.

Kim et al. [32] address the reliability of asynchronous traffic in time-sensitive in-vehicle networks (IVNs), where sensors and control units generate data streams with varying timing and criticality. This presents a significant challenge, given that strict latency and determinism are required for safety functions. To resolve this, the authors propose a specialised strategy to manage heterogeneous traffic flows, ensuring that time-critical messages meet their deadlines even in the presence of asynchronous traffic.

In summary, while 5G URLLC is a cornerstone for connecting autonomous vehicles and intelligent transport systems, significant gaps remain in deployment (coverage and consistent performance), reliability under mobility, and integrated security. Many of these gaps are active research fronts in the automotive and telecommunications industries. This highlights the critical need for predictive and agile resource allocation and prioritisation tools that can manage multi-link connectivity, prioritise safety-critical data under high mobility, and ensure graceful service degradation when network conditions are suboptimal [6], [7], [30], [31], [32].

1.3.5 Research Gaps and Future Directions

Although significant advancements have been made in 5G URLLC, its current capabilities remain insufficient to meet the stringent requirements of real-time WSN applications. This discrepancy highlights key areas for future research and innovation:

1) Balancing Latency, Reliability, and Energy Trade-offs: One fundamental gap is the joint optimisation of the triad of ultra-low latency, ultra-high reliability, and energy efficiency. Most research solutions improve one aspect at the expense of another. For example, using shorter packets and a minimal protocol handshake reduces latency but compromises the reliability, whereas adding redundancy or strong codes improves reliability but consumes more time and energy [1], [5], [33]. Future work must explore cross-layer schemes (spanning the physical layer to the application layer) that can dynamically balance these trade-offs. For instance, techniques such as adaptive modulation or coding, intelligent packet bundling, or on-demand wakeups can be tuned via ML to find a Pareto-optimal operating point in real-time. An emerging concept is AI-driven network optimisation – using ML to predict traffic patterns and channel conditions, enabling the network to proactively schedule resources and avoid collisions or delays [1], [5], [33]. Early studies indicate that AI/ML can help manage unpredictable traffic and

improve scheduling for URLLC; however, deploying such solutions in live networks is still in its infancy.

2) Scalability and Wide-Area URLLC: As mentioned, current 5G systems struggle to support URLLC guarantees as the number of devices or coverage area grows. This reveals a critical gap not only in infrastructure but also in intelligent control. Future network architectures, such as ultra-dense small cells or cell-free massive MIMO, are proposed to address this scalability challenge. However, while these strategies improve coverage, they exponentially increase the complexity of resource management [4], [34], [35].

Simply adding more base stations results in more frequent handovers and severe inter-cell interference, which must be meticulously managed in real time to avoid violating the strict URLLC budget. Traditional, rule-based scheduling algorithms are ill-prepared to handle the dynamic and high-dimensional nature of these scaled environments. This leads to a primary research direction: the development of AI-driven resource allocation and prioritisation schemes. By leveraging ML, DL and RL, networks can learn traffic patterns, predict mobility and channel conditions, and perform proactive, coordinated scheduling across multiple cells. The future lies not just in deploying more hardware, but in creating the intelligent control plane capable of orchestrating these complex resources to deliver scalable URLLC performance efficiently [4], [34], [35].

3) Intelligent Mobility Management: The unresolved issue of maintaining URLLC during high-speed mobility is a fundamental research gap. While concepts such as make-before-break handover or multi-link diversity exist, their practical implementation hinges on predictive and adaptive resource management [9], [11], [36]. This is fundamentally an intelligent problem: how can the network anticipate a device's movement and make optimal decisions in real-time? This suggests a clear future direction: the development of AI-driven mobility managers. Such a system would not just react to a handover but predict it using device trajectory and network load data. The core research challenge lies in using this prediction to allocate resources within the targeted cell proactively, prioritise traffic across multiple available links (e.g., 5G and Wi-Fi), and dynamically adjust transmission parameters to counter Doppler effects [9], [11], [36]. Instead of being a disruptive, protocol-driven event, a handover becomes a seamless transition orchestrated by an intelligent agent. Developing these AI models, capable of making split-second, context-aware decisions, is critical to solving the URLLC mobility challenge for applications such as autonomous vehicles.

4) Adaptive Security and Resource Management: The inherent conflict between security overhead and the URLLC latency budget creates a significant research gap, not just for new cryptographic protocols, but also for their intelligent application. A one-size-fits-all security policy is inefficient for WSNs: applying maximum-strength encryption to every packet would be too slow and power-intensive, while weak security is unacceptable for critical data [29], [35], [37].

Furthermore, this issue points to a clear future direction: adaptive, context-aware security management. The core challenge is to dynamically select the appropriate security level

for a given data flow based on its application, current threat level, and network conditions. This transforms security from a static property into another dimension of QoS that must be actively managed alongside latency and reliability. For instance, a critical control command might require robust encryption, while a routine sensor reading could use a more lightweight scheme [29], [35], [37].

Making these fine-grained, real-time decisions is an ideal task for an AI-driven resource allocation tool. Such a system could learn to balance trade-offs, deciding when to apply physical-layer security, which cryptographic suite to use, and how to route traffic through secure network slices without violating the end-to-end latency budget. The future of URLLC security lies in intelligent orchestration, ensuring that protection is applied precisely where and when needed, without unnecessarily compromising performance [29], [35], [37].

5) Holistic End-to-End Orchestration: A fundamental gap exists in the lack of a holistic, cross-layer framework capable of guaranteeing QoS in real-time. While individual components such as real-time monitoring and edge computing are being developed, they often operate in isolated domains, preventing proper end-to-end optimisation. The key challenge is no longer just improving one layer, but orchestrating all layers in concert [38], [39], [40].

Furthermore, this highlights a clear research direction: the creation of a unified, AI-driven control plane that acts as the network's central nervous system. Such a system would ingest data from all layers – application-level latency requirements, real-time channel state from the physical layer, and processing load from edge nodes – to make integrated decisions. For instance, it could dynamically decide whether to process data on-device to save latency, offload it to the edge for more complex analysis, or adjust radio parameters, all based on a unified model of the end-to-end performance [38], [39], [40].

The development of this intelligent orchestration, likely using techniques such as RL to learn optimal policies for resource allocation and task placement, is the key to unlocking genuine, verifiable end-to-end URLLC assurance [38], [39], [40].

1.4 Problem Statement

The paradigm shift towards 5G and beyond networks is predicted to deliver URLLC, the cornerstone for mission-critical WSN applications. However, the core scheduling mechanism responsible for managing network resources remains a significant bottleneck. Traditional, rule-based schedulers (e.g., Proportional Fair and Round Robin) are fundamentally ill-equipped to handle the stochastic traffic patterns, dynamic channel conditions, and competing QoS demands inherent to real-world 5G deployments [1], [5], [8], [15], [28].

While AI is widely touted as the solution, its application to this domain reveals a set of profound and interconnected research gaps that this research systematically addresses. The overarching problem is a lack of a comprehensive, empirically grounded framework

that progresses from passive analysis to active, autonomous control of network traffic [1], [5], [8], [15], and [28].

This problem manifests in two primary, sequential gaps:

1) The Performance Insight and Prediction Gap

Before intelligent control can be achieved, a deep, quantitative understanding of scheduling performance in a live network is required. Currently, there is a lack of robust methodologies to move beyond theoretical models and use operational data to rigorously evaluate and predict scheduler behaviour. This gap encompasses several specific challenges:

a) Quantitative Comparison: How can we accurately compare the real-world QoS delivery of traditional algorithms under dynamic conditions? This requires moving beyond simple metrics to sophisticated regression models that can reveal the complex, underlying relationships between scheduler choice and performance indicators, such as latency and throughput.

b) Behavioural Classification: Can we “fingerprint” a scheduling discipline (e.g. Priority queuing (Prio), Hierarchical Token Bucket (HTB), and Class Based Queuing (CBQ)) based solely on its resulting traffic patterns? Solving this classification problem is fundamental for real-time monitoring and diagnostics.

c) Predictive Forecasting: The ability to accurately forecast critical QoS metrics, particularly network delay, is paramount for proactive resource management. This is a non-trivial time-series problem that demands comparing advanced sequential DL models (such as LSTMs and Transformers) with strong ML baselines to determine the most effective forecasting architecture.

2) The Autonomous Scheduling and Control Gap

Predicting network state alone is insufficient; the ultimate goal is to influence it for optimal performance. This leads to the second, more advanced gap: the absence of a viable and autonomous agent capable of learning and executing optimal, real-time scheduling policies. Traditional optimisation methods are too slow for dynamic 5G environments. The challenge is to reframe network scheduling not as a static configuration problem, but as a dynamic, closed-loop control problem. This requires:

a) Formalising the Environment: Mathematically structuring the scheduling challenge as a Markov Decision Process (MDP), where an agent makes decisions (actions) based on the current network state to maximise a long-term reward signal that encapsulates competing goals such as high throughput and low delay.

b) Developing an Intelligent Agent: Designing and implementing a sophisticated RL agent, such as one built on a Transformer architecture, capable of learning a complex, non-linear scheduling policy directly from its interactions with the network environment.

The central problem addressed by this research study is the transition from static, rule-based scheduling to dynamic, intelligent, and ultimately autonomous resource

management [1], [5], [8], [15], [28]. This encompasses creating and validating a spectrum of AI-driven tools that can first characterise and predict the behaviour of network schedulers with high fidelity and then leverage that insight to enable a self-learning agent to actively control network traffic and deliver on the stringent QoS promises of 5G URLLC.

The fundamental scientific research question that drives this work is: Can advanced AI techniques – specifically ML, DL, and DRL – be effectively applied to dynamically prioritise and schedule real-time IP packets in 5G WSNs, thus significantly improving QoS metrics (latency, jitter, throughput, and reliability) compared to traditional methods, especially under conditions of high network load and diverse traffic demands?

1.5 Research Aims and Objectives

1) Research Aim

The primary aim of this investigative study is to design, implement, and empirically validate a hierarchical suite of AI-driven methodologies – encompassing predictive modelling and autonomous control – to overcome the limitations of traditional scheduling and significantly enhance QoS for real-time traffic in 5G WSNs.

2) Research Objectives

To achieve the aim established for this research study, the following specific objectives will be pursued:

a) To quantitatively evaluate and compare traditional scheduling algorithms using advanced ML:

- Develop and apply regression models (e.g. Exponential Gaussian Process Regression (EGPR) model) to establish an empirical performance baseline for traditional schedulers (PF and RR)
- Design and train a range of supervised ML and DL classification models to accurately “fingerprint” and differentiate the behaviour of various scheduling disciplines from raw operational traffic.

b) To develop and assess high-fidelity predictive models for QoS forecasting:

- Implement and evaluate advanced DL architectures (e.g. Convolution Neural Networks (CNN), LSTM, CNN-LSTM, Transformers) for their ability to predict scheduling outcomes and key QoS metrics from complex, time-series network data.
- Conduct a comparative analysis to determine the superior model for accurately forecasting next-step network delay, a critical enabler for proactive resource management.

c) To design and implement an autonomous, self-learning agent for dynamic traffic scheduling:

- Formulate the real-time network-scheduling problem as a Markov Decision Process (MDP), providing a formal framework for intelligent decision-making.
- Develop and train a novel autonomous scheduling agent using DRL, leveraging a Transformer-based architecture to learn an optimal scheduling policy directly from network interactions.

d) To evaluate the proposed AI-driven methodologies in a realistic Network environment:

- Conduct a series of controlled experiments within a simulated 5G WSN testbed to systematically evaluate the performance of each developed model and agent.
- Demonstrate the significant improvements in key QoS metrics (delay, jitter, throughput, and reliability) offered by the proposed predictive and autonomous control approaches compared to traditional scheduling methods, especially under high load and dynamic traffic conditions.

1.6 Methodological Overview of Proposed Approaches

This investigative study employs a multi-stage, empirical-grounded methodology designed to systematically progress from passive analysis to active, autonomous control of network traffic. The entire framework is validated using data generated from a live network testbed, ensuring the relevance and applicability of the findings. The methodology is structured around three core stages:

1) Performance Characterisation via Supervised Learning: The initial stage establishes a robust empirical baseline for scheduler performance. This is achieved through a two-pronged supervised learning approach. First, advanced regression models (e.g. EGPR) are applied to quantitatively evaluate and compare the QoS delivery of traditional scheduling algorithms. Concurrently, a suite of ML and DL classification models – including novel architectures that process image-based representation of network traffic – is developed to accurately “fingerprint” and differentiate the behaviour of various scheduling disciplines based on their operational traffic patterns.

2) Predictive QoS Forecasting with DL: Building upon this analytical foundation, the methodology transitions to predictive forecasting. This stage focuses on developing high-fidelity models for real-time prediction of critical QoS metrics, specifically network delay. A rigorous comparative evaluation is conducted between advanced sequential DL models (LSTM and Transformer) and a strong, non-sequential baseline (Random Forest) to ascertain the superior architecture for capturing complex temporal dependencies in live network data.

3) Autonomous Scheduling via DRL: The final and most advanced stage addresses the core challenge of autonomous control. Here, the network-scheduling problem is formally re-framed as a Markov Decision Process (MDP). A novel, self-learning scheduling agent built on a Transformer-based DRL architecture has been developed. This agent is trained

online within the live environment to learn and execute an optimal scheduling policy, demonstrating a complete shift from static configuration to dynamic, intelligent resource management.

This comprehensive, tiered methodology enables a holistic investigation, starting with understanding the problem through data, progressing to predicting its evolution, and culminating in an autonomous solution that actively manages it.

1.7 Original Contributions and Novelty of This Work

This study delivers an operational, AI-driven toolkit for QoS assurance, validated end-to-end on a live network. Across three complementary studies – PF vs RR predictability analysis, passive scheduler fingerprinting, and a Transformer-based RL scheduler – we move beyond simulation to establish predict → schedule → verify as a practical path for 5G and beyond QoS. Below is a list of the original contributions of this work:

- 1) Live, quantitative PF vs RR comparison:** A reproducible workflow using real traffic (delay, jitter, instantaneous/average throughput, and delivery ratio) separates PF’s latency consistency from RR’s throughput-centric behaviour, closing the lab-to-field gap.
- 2) QoS predictability as a first-class KPI:** Using EGPR, we show PF exhibits tighter error and higher R^2 than RR (e.g., Root Mean Squared Error (RMSE) 3.20 ms vs 5.99 ms), elevating predictability latency alongside classic averages for SLA design.
- 3) Actionable latency predictors:** We identify instantaneous throughput (stronger than averages) and jitter as first-order drivers of delay, yielding precise targets for control and monitoring.
- 4) Hybrid evaluation method:** We contribute a replicable analytic blueprint that fuses descriptive KPI statistics with EGPR inference to judge scheduler stability – not just mean performance.
- 5) Passive verification of queuing policies:** We demonstrate that ML classifiers (notably Optimisable Trees) can fingerprint PRIO/HTB/CBQ directly from packet traces, enabling low-overhead observability and compliance checks.
- 6) Protocol-level insights into scheduler behaviour:** We map which protocols expose each scheduler’s “signature” (e.g., HTB highly distinguishable; PRIO easy to fingerprint; CBQ smoothing reduces separability), informing both tuning and privacy trade-offs.
- 7) Transformer-based RL scheduler with field evidence:** We design and deploy a Transformer-based RL agent for network traffic scheduling that converges rapidly (loss → 0, with high rewards), and sustains low single-digit-ms latency with 100% scheduling success ratio in live operation.
- 8) Scalability and edge readiness:** We reveal that the RL approach and analytics remain effective under heterogeneous, resource-constrained settings, positioning the methods for MEC/edge and dense IoT deployments.

9) Broader impact pathways: We translate results into practitioner guidance (AIOps loops, telemetry hooks), standardisation angles (predictability metrics in SLAs), and research roadmaps (data, robustness, radio-aware integration).

1.8 Thesis Outline

The remainder of this Thesis is structured to systematically address the research objectives outlined in this chapter. The work is organised into seven chapters, each building upon the last to create a cohesive narrative from foundational knowledge to novel contributions.

Chapter 1: Introduction

This chapter has introduced the research project, establishing the context, motivation, and relevance of improving QoS in 5G WSNs. It has defined the core research problem, articulated the specific aims and objectives, provided a high-level overview of the proposed AI-driven methodologies, and outlined the structure of this Thesis.

Chapter 2: Literature Review

This chapter provides a comprehensive review of the existing body of knowledge. It begins by examining the QoS demands of URLLC applications and limitations of traditional, model-based scheduling approaches. It then critically analyses the state of the art in applying ML, DL, and DRL schemes to the problem of resource allocation and traffic prioritisation in 5G networks, identifying key research gaps that this Thesis aims to address.

Chapter 3: Research Methodology

This chapter outlines the overarching methodological framework that guides the research. It outlines the specific research design, detailing the simulation and modelling approaches used for initial comparisons and, most critically, presenting the comprehensive framework for designing and implementing the suite of AI models (ML, DL, and DRL) using data captured from a live, operational network testbed.

Chapter 4: Design and Implementation of Experimental Systems

This chapter provides a detailed technical account of the design and implementation of the experimental systems. It covers the setup for each of the five core experiments, from the configuration of scheduling disciplines for data capture to the specific architectural design and implementation of the ML models, the classification algorithms, the various DL architectures (CNN, CNN-LSTM, LSTM), the delay prediction models, and the culminating Transformer-based RL agent.

Chapter 5: Results and Discussion

This chapter presents and discusses the empirical results obtained from the experiments detailed in Chapter 4. Each of the five main experimental investigations is analysed in turn, with a focus on interpreting the performance, accuracy, and behaviour of the

developed models. The chapter concludes with a cross-experiment comparison that synthesises the initial findings and highlights key performance trade-offs.

Chapter 6: Synthesis and Implications of Findings

This chapter moves from individual results to a holistic synthesis of the entire body of experimental work. It integrates the findings from all five experiments to discuss their collective contributions, the broader implications for theory and practice, the limitations of research, and specific, actionable recommendations for practitioners and researchers in the field.

Chapter 7: Overall Conclusions

This final chapter provides the overall conclusions of the Thesis. It summarises the key findings in a high-level narrative, definitively answers the central question, and reflects on the primary contributions to knowledge. The chapter concludes with a brief discussion of the most impactful directions for future research, therefore solidifying this work's contributions and its place within the broader academic landscape.

Chapter 2: Literature Review

2.1 Introduction

This chapter presents a literature review for the research study focused on enhancing QoS in 5G WSNs through selective prioritisation of real-time IP packets, particularly for applications demanding URLLC. The advent of 5G enables novel WSN applications across domains such as industrial control, autonomous systems, and remote healthcare, which often impose stringent latency and reliability requirements that traditional QoS mechanisms cannot meet. Addressing these demands is critical for the successful deployment and operation of these next-generation systems [1], [20].

This review critically evaluates the current state-of-the-art relevant approaches to this challenge. It will begin by examining QoS metrics and challenges specific to WSNs, followed by an analysis of URLLC requirements within the 5G framework. Existing packet prioritisation techniques and resource allocation schemes will be surveyed. Furthermore, the potential contributions and limitations of enabling technologies such as Software-Defined Networking (SDN), Network Function Virtualisation (NFV), Edge Computing, and Machine-Learning-based solutions for managing QoS in these complex environments will be explored [1], [20].

By synthesising insights from the existing literature, this chapter aims to identify open research challenges and opportunities for developing adaptive and intelligent prioritisation mechanisms suitable for URLLC traffic in 5G WSNs. This analysis will directly inform the refinement of the research questions and the methodological framework guiding this investigative study [1], [20].

2.2 Applications and QoS Demands of 5G WSNs in URLLC Scenarios

5G-enabled WSNs extend traditional sensing platforms with millisecond-level latency and “five-nines” reliability capabilities collectively branded as URLLC. This combination unlocks mission-critical services in autonomous driving, industrial automation, smart healthcare, and massive-scale IoT [1], [5], [41].

Initially, an overview of 5G WSN is provided, highlighting the radio-access features – short transmission-time intervals, massive MIMO, network slicing – that make them uniquely suited to URLLC workloads. Next, this study maps three emblematic domains (autonomous vehicles, remote surgery, and smart factories) to concrete QoS targets such as sub-10 ms end-to-end delay, and 99.999% or more reliability, and sustained Mbps-level uplink throughput [1], [5], [41].

Meeting these targets in dense, highly dynamic environments presents three cross-cutting challenges: congestion control under bursty traffic, fine-grained resource allocation across co-located slices, and robust security without introducing delay [1], [5], [41].

2.2.1 What is a 5G Wireless Sensor Network?

A 5G-WSN couples dense, resource-constrained sensor nodes with radio access and core network features introduced in 3GPP Releases 15 and 18, most notably URLLC numerology, mmWave/massive MIMO capacity enhancements, and network slicing [1], [7], [12], [35].

Compared with legacy WSNs, 5G-WSN sustains ≤ 1 ms air-interface latency, $> 99.999\%$ reliability, and multi-Mbps uplink throughput per device. These figures are enabled by (1) mini-slot scheduling and grant-free uplink that bypass the request-grant delay, (2) massive MIMO beamforming that multiplies capacity while extending range, and (3) edge-anchored user-plane functions that keep control loops local. Sophisticated orchestration (SDN + NFV) allows thousands of sensor slices to coexist while meeting per-slice service-level agreements [1], [7], [12], [35].

The platform already underpins V2X hazard exchange, < 10 ms motion control in Industry 4.0 lines, remote endoscopy with 4K video, and city-scale environment monitoring. Field trials report 50% energy savings over Wi-Fi multi-hop meshes when traffic exceeds 10 MB h^{-1} per node [1], [7], [12], [35].

Release 18 expands side link relay and AI-native air-interface functions, pointing towards self-optimising 5G-WSNs that can re-slice resources in milliseconds. As deployment costs fall, analysts project more than 45 billion 5G-connected sensors by 2030 [1], [7], [12], [35], pushing research towards adaptive prioritisation and in-network intelligence.

2.2.2 Canonical URLLC Application Domains

2.2.2.1 Autonomous Vehicle

Autonomous vehicles represent a transformative application of 5G WSNs, critically dependent on ultra-low latency and high reliability. These vehicles require a continuous stream of detailed sensor data (such as LIDAR and high-definition video) for real-time navigation, obstacle detection, and decision-making. Consequently, network performance must support millisecond-level latency to minimise reaction times and ensure immediate responses to changing road conditions, while also supporting high data rates for sensor feeds. 5G WSN enables this by providing the reliable, high-bandwidth connectivity that is vital for the safe and efficient operation of autonomous vehicles, particularly in complex urban environments [7], [30], [32].

For instance, a level-4 car ingests 4-8 Gbps of raw sensor data; within the vehicle, fusion reduces this to 50-150 Mbps, which must then be transmitted to a roadside or edge node for cooperative perception and HD map updates. Safety-critical packets (e.g., braking intent) carry ≤ 300 bytes but demand ≤ 5 ms end-to-end latency and $\geq 99.999\%$ reliability as specified in 3GPP TS 22.186 Release 17 [7], [30], [32].

5G-WSN satisfies these bounds through three NR-V2X mechanisms:

- Mini-slot and grant-free uplink eliminate request-grant delay.
- PC5 sidelink enables direct vehicle-to-vehicle exchange in < 1 ms, bypassing the gNB;

- Network slicing, combined with edge-hosted User Plane Function (UPF) [7], [30], and [32], keeps perception loops local, meeting the 10-km radio + backhaul budget at 120 km/h. Release 18 further tightens the URLLC profile, targeting sub-1 ms air latency and improved Doppler robustness.

2.2.2.2 Telemedicine and Remote Surgery

In telemedicine, and particularly within the highly demanding domain of remote robotic surgery, 5G WSNs are envisioned as a foundational enabler. Performing intricate procedures remotely necessitates the near-instantaneous, reliable transmission and synchronisation of multiple critical data streams: high-definition (potentially 4K or even 8K) video from endoscopy cameras, real-time haptic feedback allowing the surgeon to “feel” tissue resistance, precise teleoperation commands for robotic actuators, and vital sign telemetry [27], [28].

Consequently, the network traffic characteristics are exceptionally stringent. According to [27], [28], ultra-low end-to-end latency, often targeting sub-10 millisecond RTT, is paramount for seamless hand-eye coordination and realistic haptic feedback; delays beyond this threshold can induce simulator sickness or compromise surgical precision. Equally critical is ultra-high reliability, often specified at 99.999% (“five-nines”) or higher (a hallmark of URLLC), alongside minimal latency variation (jitter). Packet loss or connection instability during a procedure is unacceptable due to the life-critical nature.

Furthermore, while latency and reliability are primary, significant, guaranteed bandwidth is required to simultaneously support multiple high-resolution video streams and complex sensor data without congestion. Strict data integrity is also non-negotiable to ensure control signals are transmitted and received without corruption. 5G network aims to meet these challenges through URLLC features (such as specific numerology, mini-slots scheduling) and potentially dedicated network slices, which reserve resources and isolate critical surgical traffic from other network activity. This capability to provide consistent, high-fidelity, secure connectivity is key to enabling surgeons to operate remotely with confidence, dramatically expanding access to specialised medical expertise [27], [28].

2.2.2.3 Industrial Automation

The advent of 5G technology is poised to significantly transform industrial automation by providing enhanced wireless connectivity tailored to demanding operational requirements. Industrial environments often generate complex network traffic patterns, characterised by a confluence of high-volume data streams from numerous sensors, a massive density of connected devices, and critical control loops requiring ultra-low latency and exceptionally high reliability [25], [26], [42]. Additionally, 5G is designed to address these multifaceted needs through distinct service categories and foundational technologies. URLLC targets mission-critical applications, such as closed-loop control and functional safety, providing latency potentially in the sub-millisecond range and high reliability (e.g., 99.999% or higher), achieved through mechanisms like shorter transmission time intervals and grant-free uplink access. Concurrently, eMBB provides the high data rates (Gbps-level) required for applications such as high-definition video monitoring, augmented reality interfaces, and the transfer of large datasets from complex

sensors or digital twins. For scenarios involving a vast number of relatively simple devices, mMTC provides scalable connectivity, supporting connection densities potentially up to one million devices per square kilometre [25], [26], [42].

Crucially, 5G introduces network slicing, enabling the creation of multiple virtual networks on a common physical infrastructure. Each slice can be customised with specific QoS parameters (latency, throughput, reliability), ensuring that diverse applications – from critical control to background monitoring – receive the appropriate network resources without mutual interference. Furthermore, 5G’s low-latency transport synergises effectively with Mobile Edge Computing (MEC), allowing data processing and analytics to be performed closer to the data source. This reduces end-to-end latency for real-time decision-making and alleviates backhaul network load. The integration potential between 5G URLLC and Time-Sensitive Networking (TSN) standards is also a key area, aiming to offer seamless, deterministic communication across both wireless and wired segments of the industrial network [25], [26], [42].

These technological advancements directly enable significant improvements in industrial automation applications. For example, predictive maintenance strategies can leverage eMBB for high-frequency sensor data acquisition and MEC for real-time analytics, allowing the accurate forecasting of equipment failures. Automated manufacturing lines benefit substantially from the precise, low-latency synchronisation capabilities afforded by URLLC, potentially enhanced by TSN integration, which is vital for optimising production efficiency, ensuring product quality, and maintaining operational safety [25], [26], [42] (see Table 2.1).

Table 2.1: Comparison of Applications by Operational Principle, Traffic Characteristics and Requirements.

Application	Operational Principles	Network Traffic Characteristics	Demanding Requirements
Autonomous Vehicles	Utilise sensors for navigation, obstacle detection, and real-time decision-making. Autonomous operation in various environments	Ultra-low latency, high data rates for transmitting sensor data (e.g., LIDAR, HD video).	Ultra-reliable connectivity, minimal reaction times, high bandwidth, and real-time data processing.
Telemedicine (Remote Surgery)	High-precision remote control of surgical instruments. Real-time transmission of high-resolution video and control signals for surgery.	Ultra-low latency, high bandwidth, and high data integrity for video and instrument control.	Exceptionally stable, reliable network connections, precise and immediate transmission of control commands.

Industrial Automation	Real-time monitoring and control of machinery and manufacturing processes. Predictive maintenance and automated manufacturing lines.	High data rates and low latency for a large volume of data from sensors and devices.	Robust, scalable connectivity, instantaneous decision-making, and synchronisation between machines.

2.2.3 Domain-Specific URLLC QoS Requirements and Budgets

URLLC is a critical service category in 5G NR and future 6G systems, designed for mission-critical applications such as industrial automation, autonomous driving, and remote surgery. Unlike previous mobile communication services, URLLC targets extremely demanding QoS parameters such as ≤ 1 ms one-way user-plane latency and $\geq 99.999\%$ success probability (or $\leq 10^{-5}$ packet error rate) for a 32-byte packet over the wireless interface, as defined by the 3GPP specifications [5], [7], [27], [43].

Delivering these stringent figures end-to-end (E2E) presents significant challenges. The tight latency budget must be meticulously allocated across different network segments while maintaining high reliability despite fundamental wireless impairments and network dynamics [5], [7], [27], [43] (see Table 2.2).

Table 2.2: Domain-Specific URLLC QoS Requirements and Budgets.

Segment	Typical Budget (μ s)	Main Contributors	Key NR/5G Techniques for URLLC
RAN (Air Interface)	400-600	Scheduling delay, HARQ retransmissions, PHY/MAC processing	Mini-slot scheduling (2-7 OFDM symbols), repetition coding, grant-free uplink (configured grant), packet duplication (via Dual Connectivity).
Transport/Fronthaul	150-300	Switching, queuing, and serialisation delays	Deterministic networking (e.g., Time-Sensitive Networking – TSN over 5G), prioritised queues, dedicated resources (e.g., WDM).
Core and Edge/MEC	200-300	User Plane Function (UPF) processing,	Local UPF deployment, Multi-

		application server logic	access Edge Computing (MEC) platforms, lightweight containerised VNFs, optimised service-based architecture.

Achieving the required reliability often entails specific 5G NR features designed to tackle packet loss. Schemes such as packet duplication (sending the same packet over multiple paths, e.g., via different carriers or base stations using Dual Connectivity), coding repetition, and robust modulation and coding schemes are employed. Furthermore, fast Hybrid ARQ (HARQ) processes, ultra-lean design minimisation overhead, and control signalling delays [5], [7], [27], and [43]. Specific QoS Flows identified by the 5G QoS Identifier (5QI), such as 5QI values 80-84, are standardised to handle URLLC traffic with appropriate priority and resource allocation [5], [7], [27], [43].

The challenge is further compounded by the need for URLLC traffic to coexist efficiently with other service types, such as eMBB and mMTC, on the same network infrastructure. To manage this, 3GPP Release 17 introduced mechanisms such as dynamic multiplexing and preemption indicators, allowing high-priority URLLC packets to interrupt ongoing transmissions of lower-priority traffic. Network slicing allows for dedicated virtual network resources for URLLC services. Integration with Time-Sensitive Networking (TSN) standards, crucial for industrial automation, is also a key focus in ongoing 5G evolution (e.g., Release 18) to guarantee deterministic communication over wireless links [5], [7], [27], [43].

The specific E2E QoS requirements can vary significantly across application domains, dictating precise budget allocation and technology choices (see Table 2.3).

Table 2.3: QoS Requirements for Delay-Sensitive Applications.

Application Domain	E2E Latency (max)	Packet Error Rate (max)	Jitter (max, e.g., 99 th %)	Typical Payload/Bandwidth	Notes
Telemedicine (Remote Surgery)	$\leq 1\text{-}5\text{ ms}$	$\leq 10^{-6}$	$\leq 50\text{ }\mu\text{s}$	High (e.g., HD video), 4-8 Mbps, 250-B control	Requires bi-directional haptic stream; MEC essential.
Autonomous Driving (Collision Avoidance)	$\leq 5\text{-}10\text{ ms}$	$\leq 10^{-5}$	$\leq 100\text{ }\mu\text{s}$	~300-B CAM/CPM messages; Moderate BW	Often uses sidelink (PC5) + Network

					(Uu) for redundancy.
Industrial Automation (Motion Control)	≤ 10 ms	$\leq 10^{-5} - 10^{-6}$	≤ 10 μ s	64-256 B cyclic data; Moderate BW	TSN integration is critical, often with private 5G networks.
Power Grid (Fault Isolation)	≤ 3 ms	$\leq 10^{-6}$	≤ 10 μ s	~ 100 B control messages; Low BW	Requires strict E2E slice isolation, including transport.
AR/VR (Tactile Feedback)	≤ 10 ms	$\leq 10^{-5}$	≤ 1 ms	Varies widely; potentially High BW	Low jitter is crucial for immersive experiences.

Meeting domain-specific URLLC requirements requires a holistic approach. Every microsecond of latency and fraction of reliability counts, demanding optimised resource allocation across RAN, transport, and core/edge domains, enabled by advanced 5G NR features and intelligent coexistence mechanisms. The placement of application logic at the network edge (MEC) is often indispensable for satisfying the most stringent latency targets [5], [7], [27], [43].

2.2.4 Cross-cutting QoS Challenges

1) Stringent QoS Requirements: The foremost challenge in URLLC is meeting its stringent latency and reliability requirements simultaneously. URLLC applications demand latency below 1 ms and success probabilities of 99.999% or higher, which is an order of magnitude more demanding than typical broadband services [5], [7], [27], [39], [40], [43]. In practice, achieving such ultra-low latency with high reliability is highly challenging, as even minor transmission delays or errors can violate QoS targets. Traditional cellular frameworks (e.g., LTE) were not designed for such low latency – LTE’s Transmission Time Interval (TTI) is 1 ms, which already consumes the entire latency budget for URLLC, and achieving 5-nines reliability often relied on multiple retransmissions that introduced additional delay. The very nature of URLLC traffic pushes wireless networks to their limits, as even slight jitter, queuing delays, or retransmissions can be catastrophic for URLLC performance.

2) Resource Allocation Trade-offs: URLLC traffic usually shares network resources with other service types (notably eMBB traffic in 5G). This creates a fundamental trade-off in resource allocation; thus, to meet URLLC’s latency requirement, the network must often pre-empt or prioritise URLLC transmissions over eMBB traffic. The scheduler may need to immediately transmit URLLC packets by cancelling eMBB transmissions to

honour URLLC's strict deadlines [5], [7], [27], [39], [40], [43]. While such preemptions ensure low latency for URLLC, they can severely degrade eMBB throughput if not carefully managed. The challenge is to provide URLLC with the resources it requires, exactly when needed, without unnecessarily starving other services. Coexistence of heterogeneous traffic with vastly different QoS needs is non-trivial. A naïve strict-priority scheme for URLLC guarantees its latency; however, eMBB users would experience frequent interruptions (puncturing), thereby reducing their QoS.

On the other hand, if URLLC is not given enough priority or reserved resources, it may suffer delays during congestion. This **heterogeneous QoS balancing** is cited as a significant challenge in next-generation networks. The throughput vs latency/reliability trade-off is essentially a zero-sum situation unless more innovative scheduling is employed. According to [5], [7], [27], [39], [40], [43], network slicing and other static methods often struggle to meet both URLLC and eMBB demands concurrently, underscoring the need for more adaptive solutions.

3) Bursty and Unpredictable Traffic: URLLC traffic patterns can be sporadic and unpredictable. Many URLLC applications (such as emergency messages or control signals) occur in bursts. The network must be ready to allocate resources on very short notice. Traditional scheduling algorithms that operate on fixed intervals or require advanced resource requests (grants) may introduce excessive latency [5], [7], [27], [39], [40], [43]. For instance, a grant-based uplink transmission in LTE/5G involves scheduling requests and grant signalling, which can take multiple milliseconds – unacceptable for URLLC. The 5G has introduced features such as grant-free access and mini-slots to support URLLC; however, these introduce challenges (e.g., potential collisions in grant-free access and signalling overhead for mini-slot schedules). The stochastic nature of arrivals necessitates highly dynamic and predictive resource allocation. If the network over-provisions resources to anticipate URLLC bursts, it risks under-utilisation when URLLC traffic is absent. On the contrary, if it waits until URLLC packets arrive, it might be too late to meet the deadline. According to these studies [5], [7], [27], [39], [40], [43], these challenges necessitate intelligent prediction and rapid adaptation in the resource allocation strategy, as well as effective management of the following:

a) Wireless Channel Variability: The wireless channel's time-varying nature adds another layer of challenge. Other issues, such as fading, interference, and mobility, can all cause sudden changes in link quality. A resource allocation decision that is optimal at one moment (e.g., which user or traffic to serve on a given timeslot) might become suboptimal a few seconds later due to channel fluctuation. URLLC's reliability target means that even rare deep fades or interference spikes can be problematic [5], [7], [27], [39], [40], [43]. Typically, reliability is ensured by mechanisms such as redundancy or retransmissions. However, in URLLC, there is often no time for more than perhaps one retransmission (in some cases, none at all). If a retransmission fails, the system may attempt a quick retransmission or multi-connectivity (sending duplicate packets via multiple base stations or channels). However, coordinating these within 1 ms is exceptionally challenging. The dynamic channel also complicates scheduling: the base

station must decide, in essentially real time, which resource (time or frequency) to allocate and which modulation and coding scheme to use to ensure the packet is delivered with high probability. The uncertainty in channel conditions thus demands a more proactive, adaptive allocation strategy than traditional static algorithms can offer [5], [7], [27], [39], [40], [43].

b) Strict Reliability and Safety Constraints: URLLC often entails safety-critical operations (e.g., messages for vehicle collision avoidance). This imposes an additional challenge: the resource allocation and prioritisation scheme not only needs to be efficient, but it must guarantee a certain level of reliability. In practice, this means the scheduling algorithm should ensure that virtually all URLLC packets meet their deadlines, even under heavy loads or adverse conditions. Traditional best-effort scheduling algorithms, or even QoS-class-based scheduling (as in LTE/5G quality classes), may not suffice because they do not guarantee delivery; they only differentiate service priorities. URLLC calls for a more deterministic approach to network behaviour, which is difficult to achieve in a probabilistic wireless medium. This has led the research community to explore methods such as packet duplication, proactive scheduling of backup resources, and robust admission control schemes to meet reliability targets. Each of these techniques, however, comes at the cost of additional resource usage (for instance, sending duplicate packets consumes extra bandwidth). The challenge lies in efficiently utilising such methods to ensure reliability without wasting excessive capacity [5], [7], [27], [39], [40].

4) Summary of key Challenges: In summary, URLLC, which is a service category within the 5G standard that enables certain types of demanding 5G WSN applications, faces several interrelated challenges regarding resource allocation and traffic prioritisation [5], [7], [27], [39], [40], [43]:

- **Latency vs Reliability Trade-off:** Ensure sub-millisecond latency while maintaining extreme reliability, even when traditional HARQ (hybrid ARQ) and scheduling timelines are too slow.
- **URLLC vs eMBB Coexistence:** Dynamically prioritising URLLC traffic without unduly harming eMBB throughput – for instance, finding a balance so that high-priority URLLC packets get through immediately, but lower-priority eMBB traffic still gains reasonable service.
- **Resource Utilisation:** Avoiding both resource under-utilisation (from over-reserving for URLLC that might be idle) and QoS degradation (from under-provisioning and thus missing URLLC deadlines). This is particularly difficult with unpredictable URLLC traffic bursts.
- **Real-Time Decision Making:** Making scheduling decisions on the fly at millisecond or sub-millisecond granularity, which requires fast processing and often prediction of near-future events (traffic arrivals or channel quality).
- **Network Dynamics:** Coping with time-varying channel conditions, user mobility, and network topology changes (for instance, vehicles entering/leaving a cell). The system should adapt resource allocation policies as conditions evolve, ideally in an online manner.

- **Complexity and Scale:** Achieving the above in a scalable way as networks grow in size (many users, devices, and base stations). Exhaustive search or brute-force optimisation for scheduling is computationally infeasible at the required time scale, especially in multi-cell or multi-user scenarios. The state space (possible network states) is enormous, making traditional optimisation or rule-based systems inadequate in many cases.

The challenges mentioned above have motivated researchers to seek smarter, more adaptive and predictive algorithms. In particular, AI (ML, RL, DL, and DRL) techniques are seen as promising to “learn” efficient resource allocation policies that can handle complexity and uncertainty better than fixed algorithms [5], [7], [27], [39], [40], [43]. Before delving into AI-based solutions, we next outline why conventional approaches fall short in handling the above challenges.

2.3 Limitations of Traditional Approaches

Traditional resource allocation and traffic scheduling strategies in mobile networks include fixed priority scheduling, static resource slicing, rule-based algorithms, and classical optimisation techniques. While such approaches have served well for 4G/5G eMBB services, they encounter significant limitations when applied to URLLC’s stringent demands [15], [19], [35].

2.3.1 Static Resource Slicing and Reservation

One straightforward strategy to ensure URLLC performance is to reserve dedicated resources (time slots, frequency bands, network slice resources, and configured grant resources) exclusively for URLLC traffic. For instance, a network operator might allocate a certain number of mini-slots per frame for URLLC, or dedicate one of multiple sub-channels entirely to URLLC use. While this isolation can guarantee that other services do not delay URLLC traffic, it is highly inflexible. The major drawback is inefficient resource utilisation [15], [19], [35]. If URLLC traffic is sporadic (which it often is), the reserved resources remain unused during gaps in URLLC arrivals, resulting in wasted capacity that eMBB users could have utilised.

In contrast, if URLLC demand exceeds the reserved portion, those extra URLLC packets will either be dropped or will have to wait (breaking the latency requirement). Tuning the reservation to the correct amount is extremely difficult for variable traffic. As Adhikari *et al.* summarise in their survey, static slicing techniques struggle to meet the heterogeneous specifications of eMBB vs URLLC, especially under bursty URLLC loads [110].

2.3.2 Fixed Priority Scheduling

Another conventional strategy is assigning URLLC the highest priority in the scheduler. For instance, a base station might always serve any backlogged URLLC packets before it serves eMBB packets. 5G standards even allow puncturing of ongoing eMBB transmissions to accommodate new URLLC arrivals. This priority-based scheduling ensures URLLC is served with minimal delay; however, it can drastically reduce eMBB

throughput if URLLC traffic is continuous or if the scheduler is overly aggressive [8], [25], [42].

Traditional priority schedulers typically lack a dynamic adjustment mechanism – they either pre-empt or do not. In heavy URLLC scenarios, eMBB performance collapses. In lighter URLLC scenarios, giving absolute priority might not significantly reduce throughput, but it could still be suboptimal if some URLLC latency slack is available. In short, strict priority can be too rigid. An intelligent strategy might occasionally defer a URLLC packet by a fraction of a millisecond if it can be done safely, to allow a large eMBB transfer to finish, but simple rule-based schedulers lack this nuance. They also do not explicitly account for channel quality or reliability needs; instead, they serve based on traffic class. The result is either suboptimal eMBB service or risk to URLLC packets if priorities are relaxed [8], [25], [42].

2.3.3 Conventional Optimisation Techniques

Resource allocation can be formulated as an optimisation problem (maximise throughput, or minimise latency violations, subject to constraints). Traditional methods might use linear programming, convex optimisation, or heuristic algorithms to solve these problems. While such methods have been successful for LTE and 5G eMBB scheduling, URLLC introduces complexities that often render the problems intractable or extremely hard to solve optimally in real time. For instance, consider joint optimisation of scheduling and power control to maximise a utility function that rewards eMBB throughput and penalises URLLC latency. This problem typically becomes a mixed-integer nonlinear program, which is known to be NP-hard. Researchers have resorted to decomposition techniques and approximation algorithms [5], [23], [44], [45]—for example, splitting the problem into sub-problems (bandwidth allocation, then sub-channel assignment, then power allocation) and solving each with greedy or convex methods. Even then, the solution can be complex. Recent studies indicate that traditional resource allocation strategies often optimise for one aspect (either network throughput or URLLC latency) while neglecting the intricate trade-offs between them. In other words, classical solutions tend to be myopic or oversimplified. For instance, a scheduling algorithm might maximise total throughput (serving mostly eMBB) and fail URLLC deadlines, or conversely, meet URLLC deadlines by reserving resources at a considerable cost to throughput. The multi-objective nature of URLLC/eMBB coexistence is complex to capture in a static formula. Furthermore, classical optimisation usually requires accurate models and parameters (e.g., arrival rates, channel statistics); if the actual traffic deviates from the assumptions, performance suffers [5], [23], [44], [45].

2.3.4 Latency in Decision Loop

A subtle yet essential limitation of many traditional algorithms is the latency associated with decision-making. Many scheduling algorithms (such as those employed in LTE) operate on a periodic interval (e.g., every Transmission Time Interval (TTI)). If that interval is too long (for instance, 1 ms or more), it is too slow for URLLC. Reducing the interval (to sub-ms) increases computational load; legacy algorithms might not keep up. Additionally, older systems typically did not incorporate prediction; instead, they reacted

to the current queue status and channel conditions. However, in URLLC, reacting may be too late; one often needs to anticipate a packet arrival or channel fade. Traditional approaches without predictive capability cannot exploit patterns in traffic (e.g., periodic control messages) or pre-schedule resources before the packet arrives. This reactive nature leads to either waiting delays or reliance on fast access procedures (such as semi-persistent scheduling or configured grants) that still have limitations (semi-persistent scheduling can allocate resources periodically, but if a packet misses its slot, it waits for the next, etc.) [46], [47], [48].

2.3.5 Limited Adaptability

Perhaps the overarching limitation is that conventional techniques are not adaptive or intelligent enough to handle the full range of scenarios. They typically employ one-size-fits-all rules or parameters established through offline analysis. Once deployed, they do not learn or improve. If network conditions change (for instance, new interference, more URLLC devices connecting, etc.), human engineers must re-tune the parameters or upgrade the algorithm in future standard releases [46]-[48]. This is a slow cycle. In rapidly evolving or highly variable environments (for example, a dense factory with dozens of wireless sensors turning on and off), a fixed strategy will inevitably falter in some cases. As the studies in [15], [19], [35], [39], [40], [46], [47], and [48] note, the heterogeneity of traffic and the dynamic nature of wireless channels exacerbate the complexity of resource allocation, making it extremely hard for manually designed algorithms to cope in all conditions. The result is often either degraded QoS or overly conservative resource use. For instance, LTE's scheduler achieved high reliability through robust modulation and multiple HARQ retransmissions; however, this resulted in high latency (tens of milliseconds), which is unacceptable for URLLC. Traditional systems sacrificed latency to ensure reliability or vice versa; they struggle to achieve both together.

Overall, while traditional approaches (static slicing, fixed priority, rule-based scheduling, and classical optimisation) laid the groundwork, they fall short for URLLC because they lack flexibility, adaptability, and foresight. Additionally, they do not effectively handle the trade-offs or uncertainties inherent in URLLC scenarios. This is where AI-based approaches become attractive. ML and DL can potentially learn complex scheduling policies from data or ongoing experience, adapt to changing conditions, and make near-instant decisions that better satisfy the multi-dimensional QoS requirements. The following sections review how recent research has applied ML, DL, and DRL to overcome the above limitations, focusing on the methods used and the improvements achieved [15], [19], [35], [39], [40], [46], [47], [48].

2.4 ML Approaches (Non-Deep) for URLLC Resource Allocation and Prioritisation Schemes in 5G WSNs

ML in the context of resource allocation/prioritisation schemes for 5G WSNs requiring URLLC refers to techniques that do not necessarily rely on deep neural networks, but instead on classical ML algorithms or simpler learning paradigms. These include tabular RL and small function approximators, multi-armed bandit algorithms, supervised

learning models (such as SVMs, decision trees, and ensemble methods), and emerging paradigms such as federated learning for distributed model training. In the last couple of years, several works have explored these non-deep AI schemes to improve URLLC scheduling and prioritisation [8], [15], [16], [49].

2.4.1 RL for Scheduling

RL provides a natural framework for resource allocation problems, where an “agent” (e.g., the base station scheduler) can observe the network state and take actions (allocate resources, schedule packets) to maximise a long-term reward (e.g., a reward for each packet delivered on time, minus penalties for latency drops). Early RL approaches used tabular Q-learning or policy-based methods to learn scheduling policies. However, URLLC scheduling involves an ample state space (queues, channel conditions, per-user or per-flow reliability and latency requirements, resource block allocation, etc.), which can be challenging for simple RL. Methodological approaches discussed in [8], [25], and [42] formulated the URLLC/eMBB multiplexing problem as a Markov Decision Process (MDP) and applied Q-learning to find a scheduling policy. Because the state space was huge, they combined Q-learning with function approximation (using a Deep Q-Network), essentially bridging the gap into DRL. This indicates that even in “non-deep” RL, some function approximators are needed for continuous state problems.

Nevertheless, the concept was groundbreaking: it showed that an RL agent could learn to schedule mixed traffic in a way that meets URLLC reliability while maximising eMBB throughput, without an explicit, handcrafted rule. The reward function in such works is often designed to capture the URLLC QoS – for instance, a significant positive reward for each URLLC packet delivered before the deadline, and a negative reward for a packet dropped or delayed beyond the deadline. Over time, the Q-learning agent adjusts its scheduling decisions to optimise this reward, effectively learning when to serve URLLC versus eMBB, how to utilise any slack, and so on. The results from these early RL attempts showed improved QoS trade-offs compared to fixed-priority scheduling, albeit at the cost of increased computational complexity (which later DL techniques help address) [8], [25], [42], [50].

2.4.2 Multi-Armed Bandit and Online Learning

A related class of methods treats resource allocation and prioritisation schemes as a multi-armed bandit problem, particularly in scenarios such as grant-free uplink access for URLLC. In such scenarios, each timeslot or resource block can be thought of as an “arm” to allocate to a user, and the outcome (success or failure, and the delay incurred) provides feedback. Bandit algorithms such as Thompson Sampling or Upper Confidence Bound (UCB) can learn an allocation strategy that optimises reliability or throughput over time. Recent studies [15], [51], and [52] examined resource allocation in a multi-cell system serving both URLLC and eMBB, utilising a Thompson sampling-based learning framework.

Furthermore, they deployed learning agents at distributed units (e.g., near-real-time RAN controllers in an O-RAN architecture) that continuously learn which cell or users to

prioritise. Thompson sampling (a Bayesian bandit approach) enables the algorithm to balance exploration (trying different allocation patterns) and exploitation (favouring the patterns known to work best). The use of Thompson sampling in URLLC is quite novel – it provides a way to handle uncertainty in the system (uncertainty about traffic rates or channel success probabilities) by maintaining probability distributions for each action’s expected reward. The authors reported that this distributed learning approach could make real-time scheduling decisions in the RAN and meet QoS requirements for both URLLC and eMBB traffic, while optimising resource utilisation [15], [51], [52]. This implies that the learning-based policy found a more effective way to share resources than a fixed scheme, likely by learning the optimal amount of puncturing or multiplexing to apply under various conditions. Bandit approaches are appealing for their relative simplicity (compared to full state-based RL) and their fast convergence in scenarios with repeated trials (such as scheduling slots). Nonetheless, they typically apply when the decision can be modelled without long-term state (or where each timeslot is a fresh decision), which may not capture everything (e.g., queues).

2.4.3 Heuristic ML (Fuzzy Logic, and similar methods)

Some works have tried fuzzy-logic-based controllers or heuristic learning. For instance, a scheduler might use a fuzzy rule set tuned by an algorithm (such as a genetic algorithm or RL). These are less prominent in the recent two-year window and are generally overshadowed by RL and DL methods. The literature focus has shifted more to RL/DL, so we will not delve deeply into fuzzy approaches here, noting only that they were an earlier form of introducing “intelligence” into scheduling schemes [53], [54].

2.4.4 Federated Learning (FL) for URLLC

Federated Learning (FL) is an approach to training ML models in a distributed manner, where each device (or network node) computes updates on its local data and only aggregated updates are shared, not raw data. Federated learning has been proposed for wireless networks to enable collaborative model learning (e.g., traffic predictors or resource allocation policies) across multiple base stations or edge devices without a central dataset. In URLLC, federated learning has been investigated to manage resources in scenarios such as the Internet of Vehicles (IoV), where vehicles and roadside units might jointly learn policies for reliable communication.

In a 2023 study, Hazarika et al. [56] proposed an Asynchronous Federated Learning-based resource management scheme for URLLC-enabled IoV networks. The asynchronous aspect means nodes can update the global model at different times (which is essential when network nodes have variable connectivity). This FL approach allows the system to swiftly adapt to dynamic changes in network topology, such as vehicle mobility, base station, or Road Side Units (RSU) failures, or variations in channel quality. Essentially, each vehicle or RSU learns from its local environment (e.g., the observed success or failure of URLLC transmissions and local channel conditions) and contributes to a shared model that gradually improves scheduling and routing decisions for URLLC packets across the network. The cited research indicates that this distributed learning approach proved effective in managing resources and caching under URLLC IoV

constraints. The advantage of such an approach is twofold: (1) it can capture a wide range of scenarios by learning from many distributed experiences (robustness), and (2) it keeps data (such as local channel measurements or traffic patterns) mostly local, which can be beneficial for privacy and scalability [57], [58], [59].

Furthermore, federated learning, combined with RL (Federated RL), has also been explored – for instance, Liu *et al.* [60] demonstrated a case study where a Federated Deep Reinforcement Learning (FDRL) agent was as effective as a centrally trained agent for URLLC downlink channel access. This suggests that federated training can achieve near-centralised performance, which is promising for real deployments where central training might not be feasible.

2.4.5 Supervised Learning and Traffic Prediction

Traditional supervised ML (neural networks or simpler regression models) can be used to predict network conditions and support resource allocation. While most research momentum is towards RL and DRL, some works have explored using historical data to predict URLLC packet arrivals or channel outages. For instance, one could train a model to predict the probability of URLLC traffic in the next millisecond based on recent patterns, and then proactively schedule a mini-slot for it. Similarly, predicting when a user's channel will be good or bad could inform whether to send a URLLC packet now or wait 1 ms (if it can afford to). Studies [61], [62], [63], and [64] discussed the use of DL for channel-state estimation in a joint URLLC/eMBB allocation scenario. By improving channel estimation via ML, the scheduler has information that is more accurate and can allocate just enough redundancy or choose the proper modulation to ensure reliability, thus avoiding unnecessary retransmissions (saving time). Another conceivable supervised approach is to train a classifier to determine whether an incoming packet is URLLC-critical or can be treated as usual (in some systems, there may be mixed criticalities). This delves into traffic classification, which is simpler and less discussed in core URLLC (since URLLC traffic is usually distinctly tagged by the application). In summary, while pure supervised learning is not commonly used to directly output a resource allocation decision (because the decision space is complex and context-dependent), it often appears in supporting roles (such as predicting traffic or channel) to make the scheduler's job easier [61], [62], [63], [64].

2.4.6 Hybrid ML-Optimisation Approaches

A notable trend in recent research is the combination of classical optimisation with ML. Instead of replacing the scheduler entirely with a learning agent, some works integrate ML components into traditional frameworks. For instance, a paper by M. Alsenwi *et al.* [65] presented an optimisation-aided RL approach. They decomposed the resource slicing problem into several convex sub-problems (which can be solved efficiently by classical means) and then used a learning algorithm to handle the coupling between sub-problems. In their case, they solved for an optimal resource partition (bandwidth for URLLC vs. eMBB) using mathematical optimisation, but then used DRL to dynamically adjust how URLLC traffic is distributed among eMBB transmissions in real time. The result is a hybrid solution where the heavy lifting of finding a baseline resource split is performed

using known optimisation techniques, and fine-tuning or real-time adaptation is learned through ML. Such hybrid methods benefit from domain knowledge (e.g., optimisation ensures known constraints are met and gives a head-start solution) and the adaptability of ML to refine solutions online. This is a promising direction to avoid “re-inventing the wheel” – leveraging the strengths of both worlds [53], [54], [65].

Overall, classical ML approaches (without deep neural networks) have indeed shown benefits for URLLC resource allocation: RL agents can learn scheduling policies that outperform fixed algorithms, bandit algorithms can optimise retransmission strategies or resource blocks allocation on the fly, and federated learning enables collaborative adaptation in distributed settings, critical for vehicular or multi-cell URLLC scenarios. The common thread is that these methods introduce adaptability and learning from experience into the network. They address some limitations of traditional approaches by observing outcomes and continuously improving decisions. However, many of these methods still face scalability issues as the problem size grows. This is where DL comes in – as the following sections discuss, DL expands the capacity of ML methods to handle high-dimensional inputs and complex non-linear decision policies, which is vital for the high complexity of URLLC environments [19], [69], [70].

2.5 DL Approaches for 5G WSN (URLLC) Resource Allocation

DL refers to the use of multi-layer neural networks to learn representations or mappings from inputs to outputs. Regarding resource allocation and prioritisation schemes in 5G WSNs requiring URLLC, DL techniques have been used in various ways, ranging from predictive modelling to direct decision-making to approximating optimal policies. Unlike DRL (covered in the next section), this section focuses on DL methods that are not necessarily cast as RL problems but instead as supervised or unsupervised learning problems supporting QoS management [67], [71].

2.5.1 Neural Network-based Scheduling Decisions

One method is to train a DL model offline to approximate the outcome of an optimisation or to mimic an optimal scheduling policy and then use that model online for fast decisions. The idea is that solving the scheduling optimisation might be too slow for real-time use; however, if a model can be trained to take the current state (e.g., queues, channel conditions, etc.) as input and output an allocation decision, it can produce answers in microseconds during runtime. The authors in [8], [25], [41], and [54] propose neural network-based solutions to address the challenges facing resource allocation and traffic prioritisation in URLLC. The key point in these works is that DL models can quickly generate, for instance, the necessary power level and coding scheme to meet URLLC reliability requirements, or decide how to fragment a packet across multiple mini-slots. The result is a system capable of responding within the tight URLLC time budget, whereas iterative solvers might not. By comparing against baseline schemes, they demonstrated that a trained DL model could achieve the target reliability and latency with minimal overhead, essentially matching more complex optimisation strategies but at a

fraction of the decision time. This demonstrates the power of DL models to quickly and efficiently approximate complex resource allocation decisions.

2.5.2 DL for Prediction and Proactive Allocation

Another vital application of DL in the domain of communication networks (5G networks (WSNs)) is prediction. Since uncertainty and variability are major issues (as discussed in Challenges), being able to predict future events (even with some probability) can significantly enhance scheduling performance. DL models, such as RNNs or LSTM networks, are well-suited to detect patterns in time-series data [33], [38], [39], [62].

In the context of URLLC, researchers have considered using such models to predict, for example, changes in channel quality or traffic arrivals. For example, a DL model could be trained on historical traffic data of a sensor network to predict when a burst of URLLC packets is likely to occur (e.g., a specific sensor sends an emergency update every 100 ms, which the model can infer). Armed with that prediction, the base station can pre-allocate resources just before the burst arrives, serving the packets immediately and avoiding queue build-up. In the same way, an LSTM model might predict short-term channel fade events (if a user's channel quality is dropping, the model might foresee a deep fade in the next few milliseconds based on past Doppler patterns), prompting the scheduler to send critical URLLC data now rather than later. These kinds of proactive moves are difficult to implement with fixed algorithms; however, DL enables them by learning from data [33], [38], [39], and [62].

The works discussed in [33], [38], [39], [62], and [71] reiterate that “AI and ML schemes can optimise resource allocation and interference management in real-time, highlighting that predictive analytics can aid in anticipating network conditions. Essentially, DL can serve as the “brains” that forecast near-future network states, enabling the network to act one step ahead.

2.5.3 DL for Channel Estimation and Link Adaptation

Ensuring URLLC reliability often depends on accurate Channel State Information (CSI) and appropriate link adaptation (i.e., the choice of modulation, coding, and power). Traditional methods for CSI estimation (e.g., pilot signals and linear estimators) and link adaptation (Signal-to-Noise Ratio (SNR) thresholds for MCS selection) may not be sufficient in fast-changing or complex channels. DL has been applied to improve these lower-layer aspects, which indirectly boost URLLC performance. The works discussed in [72], [73], [74], and [75] demonstrated a DL-assisted CSI estimation technique for joint URLLC/eMBB transmissions. By utilising a DL model to denoise and predict channel quality, their scheme provided the scheduler with more reliable CSI. This indicates that the scheduler can be more confident in allocating just enough resources (e.g., just enough redundancy or a just-safe-enough modulation) to meet URLLC's error-rate target, thereby saving resources compared to worst-case assumptions. The outcome is that fewer URLLC packets experience unexpected outages because channel estimation is more accurate; hence, the required five-nines reliability is achieved with lower overhead. This example demonstrates how DL can enhance the information used in the resource allocation

decision (rather than deciding the allocation itself). Other studies have applied autoencoders or deep denoising networks for channel prediction with similar aims of improving reliability for low-latency transmissions.

2.5.4 Advanced Neural Network Architectures

Beyond fully connected networks or simple RNNs, researchers are leveraging architectures tailored to specific network topologies. One such promising tool is Graph Neural Network (GNN). In networks with many interacting links (such as interference graphs and connectivity graphs), GNNs can effectively capture relationships in graph-structured data (e.g., user-base station connectivity graphs). Research studies [76], [77], and [78] utilised a GNN to solve a joint eMBB/URLLC resource allocation problem, aiming to maximise eMBB throughput while minimising URLLC outage probability. The GNN was trained to allocate resources based on the interference graph and queue states, achieving a good balance between eMBB fairness and URLLC reliability. Their work underscores the trend of using graph-based DL to solve complex combinatorial resource allocation problems (such as matching or scheduling across multiple links). The success of GNNs in solving these problems suggests that DL can generalise better to different network scenarios (i.e., different numbers of users or different topologies) than a fixed algorithm, because the GNN can be trained on many instances and capture the underlying structure of the allocation problem [76], [77], and [78].

2.5.5 Real-Time Inference and Complexity

One might be concerned that DL models are heavyweight and slow, but a well-trained model can compute decisions very fast (a matter of microseconds on modern Central Processing Units (CPUs) or Application-Specific Integrated Circuits (ASICs)). The training phase may be intensive; however, it can be completed offline or updated periodically as needed. The benefit comes at runtime: when a URLLC packet arrives, feeding the current state into a neural network and obtaining a decision (e.g., which resource block to use, or whether to puncture eMBB) is essentially just a series of matrix multiplications – much faster than solving an optimisation from scratch. The studies discussed in [72], [74], and [77] have reported that inference using DL can significantly reduce decision latency. For example, the Parameter Generation Network mentioned earlier was explicitly designed for fast inference, enabling on-the-fly computation of scheduling parameters that meet URLLC's timing requirements. This is critical because any decision scheme that itself takes too long would defeat the purpose of URLLC. The research community has recognised this and is using techniques such as model compression, quantisation, and small but effective network designs to ensure deployment feasibility.

2.5.6 Training Data and Strategy

DL models require training data – in the context of URLLC for 5G WSNs, this could be generated via simulations or collected from real network logs. One approach is to use a high-fidelity network simulator to generate numerous examples of network states and the corresponding optimal (or at least reasonable) resource allocation decisions (perhaps

computed through an exhaustive search in a small scenario). These examples train the DNN in a supervised manner (imitation learning). A different approach is unsupervised or self-supervised: the network can be embedded in a training loop where it makes random decisions and is reinforced by a reward signal (which then transitions into the DRL domain). For pure DL (supervised), obtaining ground-truth labels (optimal decisions) can be challenging. Some studies prefer to integrate domain knowledge (such as scheduling policies) into the training – for instance, training the network to mimic weighted fair queuing for eMBB when URLLC is absent, and to mimic strict priority when URLLC load is high, allowing it to learn a smooth interpolation and everything in between. This sort of training ensures that the DNN does not make unintended decisions, while still maintaining its flexibility to adjust its choices [38], [62], [77].

2.5.7 Key Achievements of DL in URLLC for 5G WSNs

The application of DL in URLLC resource allocation and prioritisation schemes has yielded several notable achievements as reported in the literature:

1) Near-Optimal Performance: DL models have been able to closely approximate optimal or theoretically derived allocation strategies with far less computational burden. For instance, a trained model can implicitly solve a complex optimisation problem in real time, achieving reliability and latency metrics similar to those of the optimal solution computed offline [60], [65].

2) Improved Adaptability: Unlike fixed algorithms, a DL model can be retrained or updated with new data, allowing it to adapt to changes such as new traffic patterns, new interference sources, or even new objectives (for example, by changing the loss function). This adaptability was highlighted in scenarios where a model trained on one set of conditions was quickly fine-tuned for a new one, something impractical with hand-engineered solutions [25], [41].

3) Multi-metric Optimisation: By designing appropriate loss functions, DL models can inherently handle multi-objective problems. For instance, one can train a model to minimise the weighted sum of URLLC latency and eMBB capacity loss. The network will learn to balance these, effectively solving a multi-objective optimisation. In practice, researchers often include terms in the loss for URLLC delay violation count, URLLC drop count, and eMBB throughput, to achieve a balanced outcome [72], [77].

4) Real-World Validation: Some DL techniques have moved beyond simulation into testbed or hardware validation (e.g., utilising the NVIDIA GPUs or FPGAs for running the neural scheduler). Research papers in 2023 have started reporting not only simulation gains but also feasibility on actual RAN hardware. These experiments demonstrate that, for instance, a scheduling decision can be computed in under 100 microseconds using a neural network, which is well within the 1-ms slot duration of 5G.

In summary, DL enables URLLC by providing the network with a form of “intuition” or predictive power that traditional techniques cannot. Whether it is through a parameter generation network that yields instant decisions, a prediction model that pre-allocates resources, or advanced architectures such as GNNs that effectively coordinate complex

allocations, DL models are proving to be a valuable tool. However, DL models by themselves do not inherently handle long-term decision-making or the sequential nature of scheduling; they typically make a single-step decision based on the input. To address sequential decision-making under uncertainty (which scheduling inherently involves), DRL builds on these concepts. The following section discusses DRL, which combines the best of both worlds: the representational power of deep networks and the decision-making loop of RL [6], [13], [21].

2.6 DRL Approaches for URLLC in 5G WSNs

DRL has emerged as one of the most powerful approaches for resource allocation and traffic scheduling in URLLC. This technique combines neural networks with RL, enabling an agent to learn complex policies through trial and error, guided by rewards, while utilising deep neural networks to handle large state-action spaces. In the URLLC context, DRL algorithms can learn when to schedule URLLC packets, how to allocate resources (bandwidth and power) to URLLC versus other traffic, and how to adapt to network dynamics—all in an integrated framework. In recent years, various studies have applied DRL to multiple aspects of URLLC, demonstrating significant improvements in QoS [60], [65].

2.6.1 Formulating URLLC Scheduling as a DRL Problem

Typically, the network (or base station) is modelled as the DRL “agent.” The state may include information on URLLC and eMBB packet queue lengths, channel state information for each user, the time remaining before URLLC deadlines, jitter, packet loss rates, and other relevant metrics. The action could be a composite decision made every scheduling interval (e.g., every mini-slot or every 0.5 ms), which determines which user or traffic type to schedule on each available resource block, how much power to allocate, and perhaps which modulation and coding scheme to use, among other factors. The reward function is crucial – it is designed to reflect the QoS objectives. Thus, a standard reward design is to give a significant positive reward for each URLLC packet transmitted successfully within the deadline, and to impose a penalty for each microsecond of latency or for each dropped packet. A smaller positive reward might be awarded for eMBB throughput to ensure the agent also considers eMBB service (or conversely, penalise the agent for eMBB throughput loss to force it to be efficient). By running this as a sequential decision process, the agent learns policies that maximise long-term cumulative reward, which corresponds to high URLLC reliability and low latency, without sacrificing eMBB more than necessary [25], [60], [65], [68].

2.6.2 Value-Function-Based DRL (DQN)

One class of DRL algorithms utilises Q-learning with deep neural networks, commonly referred to as DQN. In the URLLC context, DQN is typically used to estimate the expected reward (QoS performance) of taking a specific scheduling decision in a given state. Early works (2019-2020) [79] applied DQN to simple URLLC/eMBB multiplexing scenarios, and more recent studies have scaled this up. For instance, [50], [52] implemented a DQN-based joint scheduler for eMBB and URLLC, where DQN learned

an optimal preemption policy. Fundamentally, the agent learned how often and when to puncture eMBB transmissions to send URLLC, ensuring that URLLC deadlines were met with minimal loss of eMBB rate. Their approach improved the throughput-latency trade-off compared to static schemes – URLLC reliability was guaranteed at the required level, meanwhile, eMBB throughput drop was kept to a minimum, because the DQN found an optimal balance (something a fixed slicing might miss). Studies presented in [50], [52], and [79] utilised a DQN agent to manage URLLC data in addition to scheduling eMBB traffic. Their approach involved training the agent to decide which URLLC packets to send (and in which resources) such that the latency constraint is met and the impact on eMBB is minimised. The reported outcome was that their DRL policy never violated URLLC latency and reliability constraints while maintaining minimal eMBB throughput loss. This is a significant achievement, as it reveals that the DRL agent has found a clever strategy. Perhaps it involves sending URLLC only in certain portions of time and frequency where it causes the least harm to eMBB, or combining multiple URLLC packets into one transmission, etc. - strategies that were not obvious a priori.

2.6.3 Policy-Gradient and Actor-Critic DRL

Beyond the DQN framework, many recent research works use policy-gradient methods (such as REINFORCE, Advanced Actor-Critic (A2C), and Proximal Policy Optimisation (PPO)) or actor-critic frameworks, which are well-suited to continuous or high-dimensional action spaces that occur in resource allocation. To illustrate, allocating power or splitting bandwidth between URLLC and eMBB may be a continuous process. Thus, Deep Deterministic Policy Gradient (DDPG) and its variants, such as Twin Delayed Deep Deterministic Policy Gradient (TD3), are popular choices for handling continuous control. Research studies done by [4], [16], [36], and [41] employed a DDPG-based algorithm for joint eMBB/URLLC scheduling. In their approach, the actor network directly outputs scheduling probabilities or a continuous parameter indicating how aggressively to serve URLLC, and the critic network evaluates the performance. Such actor-critic setups are advantageous because they can learn smooth trade-off policies – for instance, “serve URLLC immediately if its delay is above X, otherwise wait Y microseconds to serve an eMBB first,” where X and Y are implicitly learned thresholds.

Studies [4], [16], [36], and [41] also discuss the application of deep policy gradient with prioritised experience replay to train an agent for URLLC scheduling. The goal was to achieve a better long-term trade-off between throughput and latency compared to standard algorithms. By prioritising meaningful experiences during training, such as instances of dropped URLLC packets or missed deadlines, the agent learned to avoid these adverse outcomes more effectively. The study resulted in a DRL scheduler capable of maintaining the strict 1 ms delay bound for URLLC with high probability while ensuring a fair long-term resource allocation to eMBB.

2.6.4 Meta-Learning and Transfer in DRL

A recent exciting development is the incorporation of meta-learning into DRL for URLLC. Meta-learning (learning to learn) enables a model to rapidly adapt to new environments based on past experience. The research works discussed in [31], [80], and

[81] propose a DRL-based RA framework integrated with meta-learning to adaptively manage eMBB and URLLC. The motivation is that wireless network environments can change (with different path loss conditions and user loads), and a policy trained for one scenario may not work well in another. By using meta-learning, their DRL agent can quickly fine-tune its policy when the environment changes, rather than having to learn from scratch. They claim that the framework demonstrated superior adaptability across different path loss models and outperformed traditional methods under varying conditions. In practice, this means that if the network trained in an urban microcell is suddenly faced with a rural macrocell scenario (with, for instance, different propagation conditions and fewer users), the meta-DRL agent can adapt in just a few iterations to meet URLLC requirements [31], [80], [81].

In contrast, a non-meta agent would perform poorly or need extensive retraining. A related approach is transferring learning within DRL: for instance, the authors of [82] and [83] proposed using a Thompson sampling-based DRL in one part of the network and transferring its learned Q-values to another part as an initial policy. Researchers utilised a combination of on-policy and off-policy learning (PPO + DQN) to transfer knowledge and improve energy efficiency in URLLC scheduling. Furthermore, their DRL framework dynamically allocated resource blocks, avoided puncturing, and adjusted power, leveraging transfer learning to accelerate convergence. The result was rapid convergence to an optimal resource distribution that enhanced energy efficiency while maintaining URLLC QoS under unpredictable channel conditions. This illustrates that DRL approaches are not only focused on latency and reliability but also on other metrics, such as energy, a vital consideration for 6G networks [82], [83].

2.6.5 Distributed and Multi-Agent DRL

In large network environments, a single agent controlling everything may become too complex. Researchers have looked at multi-agent DRL for URLLC. For instance, multiple base stations, each with a DRL agent, can coordinate to ensure end-to-end high QoS for URLLC traffic in a multi-cell system (sometimes with a higher-level mechanism to coordinate agents, or through game-theoretic learning). One particular scenario is vehicular communications: Consider vehicles connected to multiple roadside units (RSUs) for reliability (multi-connectivity). Studies [35], [84], and [85] present a cooperative multi-agent DRL mechanism for downlink URLLC that can be applied in vehicular networks. Each connectivity link (or each RSU) had an agent that learned a power allocation policy, and higher coordination was used to maximise the overall packet success probability. The method also included robust power control to handle the fast channel variations at vehicular speeds. The result was an increase in reliability for vehicular URLLC traffic communications, as the DRL agents learned to utilise multiple links and allocate power smartly such that if one link was in outage, others could carry the traffic. This essentially trains the system to perform tasks such as simultaneous transmission or adaptive load balancing across links. The cooperative aspect signifies agents share some information or have a common reward (such as end-to-end success of a packet transmission), which encourages them to work together. Such multi-agent setups

are quite advanced and demonstrate the flexibility of DRL: one can design the observation and action for each agent, as well as a reward structure that leads to emergent behaviour fulfilling URLLC requirements [35], [84], [85].

2.6.6 QoS Results from DRL

The literature reports numerous performance gains from DRL approaches:

1) Latency and Reliability: DRL policies ensure that nearly 100% of URLLC packets meet the 1 ms deadline in simulations, even under high network loads, whereas traditional schemes might only achieve 95% under the same conditions. The difference can be life-or-death for applications such as industrial control. By learning, for instance, to drop a less important eMBB packet in a rare overload situation to save a URLLC packet, the DRL agent upholds reliability where a rule-based scheduler might falter. Many research studies show the URLLC packet delay Cumulative Distribution Function (CDF) with DRL is steep at the 1 ms mark (meaning very few miss the deadline), whereas with the simpler methods, there is a tail of packets that miss the deadline [8], [15], [25], [60].

2) Throughput Efficiency: DRL has demonstrated the ability to minimise the sacrifice of eMBB throughput. Research papers are revealing that DRL schedulers can deliver URLLC packets on time with a reliability of over 99.9%, while only reducing eMBB throughput by ~5% compared to a no-URLLC scenario. On the other hand, a static, strict-priority scheduling scheme reduced eMBB throughput by 15% under the same load. The DRL agents can figure out more efficient resource sharing – for instance, sending URLLC in unused mini-slots or using just enough power to get by, leaving more for eMBB traffic or users [8], [15], [25], [60].

3) Resource Utilisation: Some studies measure how well the channel resources are utilised (no idle resources when there is traffic that can use them). An intelligent DRL system will avoid both idle resources and overload. To illustrate, if URLLC traffic is low at a given moment, the agent will fill the channel with eMBB (utilisation ~100% for throughput); if URLLC spikes, the agent instantly shifts resources to URLLC and, if possible, temporarily buffers some eMBB. Conventional static slicing cannot achieve such fluid reallocation. Additionally, numeric results reveal that DRL achieves higher overall cell throughput while still meeting URLLC constraints, compared to static slicing [8], [15], [25], [60].

4) Energy Efficiency: Researchers are increasingly focusing on energy, especially for 6G. Some DRL approaches incorporate energy into the reward function. For instance, a DRL agent might be penalised for high transmit power use, pushing it to find low-power ways to meet URLLC (such as using diversity instead of brute-force power). Studies [1], [41], and [82] discuss energy-efficient URLLC packet delivery mechanisms using DRL, including frequency hopping and proactive dropping, to reduce energy consumption while maintaining reliability. The DRL optimised a three-step process (bandwidth, antenna, subchannel) to minimise power consumption under URLLC constraints, achieving significant power savings (improving energy efficiency by more than 20%)

while still meeting URLLC QoS. This highlights the DRL’s ability to address multi-factor optimisation in a way that earlier methods could not.

5) Robustness: DRL policies, especially those enhanced with meta-learning or trained on diverse scenarios, tend to be more robust to variations. For instance, a policy trained through DRL might handle sudden increases in URLLC load more gracefully (by automatically prioritising and perhaps dropping some nonessential load) than a fixed policy, which could become unstable or overwhelmed. Sohaib *et al.* note that their meta-DRL scheduler maintained performance across different fading conditions and even in the presence of uncertainties in the traffic model. This type of robustness is crucial for real-world deployment, where conditions are rarely exactly as assumed in the design [1], [41], and [82].

2.6.7 Interpretability and Safety

Meanwhile, DRL provides excellent performance; ongoing research is being conducted to make the decisions generated by AI-based tools more interpretable and ensure they meet strict guarantees. In safety-critical URLLC applications such as automotive, remote surgery, and industrial control, the DRL agent must always satisfy a reliability threshold. Some recent studies [86], [87] are examining risk-aware or safe DRL, which incorporates constraints into the learning process or utilises additional monitoring. For instance, an approach might use a Thompson sampling-based DRL with an interpretability layer to ensure the decisions can be explained and verified.

A different approach is to have a fall-back: if the DRL outputs something that potentially violates a rigid constraint (such as scheduling a non-URLLC packet when a URLLC deadline is about to expire), a rule-based override could take effect. This way, one can reap the benefits of learning without compromising the deterministic guarantees for critical flows. The research community is aware that just achieving good average performance is not enough for URLLC – worst-case behaviour matters. DRL algorithms are increasingly being designed with risk sensitivity (to avoid that one-in-a-million bad outcome) or being combined with conventional methods [86], [87].

In summary, DRL techniques have demonstrated the ability to learn highly efficient and flexible resource allocation and prioritisation policies for URLLC in 5G WSNs, previously unattainable with earlier methods. They address the core challenges by continuously adapting to network conditions and intelligently balancing competing needs (URLLC vs. eMBB, latency vs. throughput vs. energy, etc.) to maximise overall network utility, subject to URLLC’s strict requirements. DRL agents effectively “learn” the inherent scheduling policy that network operators might have wished to hand-design if it were possible. They represent the state of the art in 5G URLLC research, and many of these concepts are influencing how 6G networks might be managed [86], [87]. For instance, 3GPP and the O-RAN Alliance are exploring the incorporation of learned policies into RAN controllers (as seen in testbeds where a near-RT RIC runs a DRL-based xApp for scheduling). The success reported in the literature, such as meeting URLLC

demands and improving spectral efficiency compared to fixed slicing, is a strong testament to DRL's capability [86], [87].

2.7 Where AI-Driven 5G QoS Falls Short

Prior work reveals that model-based multiplexing and naïve DRL often struggle to capture fundamental eMBB/URLLC dynamics or to learn fast enough for operational use (Huang et al., 2020). Dynamic scheduling and per-TTI DRL decisions inflate control overhead and decision latency; classical outer-loop link adaptation (OLLA) is rigid due to fixed BLER targets and step sizes (Pang et al., [72]). DRL for network slicing and TSN face difficulty enforcing long-term constraints and incur high exploration costs, while purely model-based approaches lack adaptability to non-stationary traffic (Cai et al., 2024; Chen & Li). PHY-level guarantees are further complicated by short block-length coding and imperfect CSI, which undermine Shannon-style assumptions for URLLC reliability (Ren et al., [5]). Finally, the ML literature flags black-box interpretability and the limits of static dataset generalisation for beyond 5G control, weakening trust and transfer to live networks (Aloupogianni et al., [19]).

Radio-access studies reinforce these concerns: baseline NR/V2X analyses report poor scalability of downlink unicast and latency inflation from dynamic-scheduling signalling (Lucas-Estão et al.). At the same time, DRL-based link adaptation degrades with stale CQI and frequent MCS switching (Yet et al., 2023). Collectively, the record points to a persistent triad – learning speed, overhead/latency, and assurance/interpretability – that AI-driven approaches do not yet resolve consistently.

Complementary, cross-cutting gaps (post-2023) that amplify these shortfalls include:

- 1) Limited real-world validation:** Heavy reliance on simulations that fail to capture real-world complexities such as unpredictable traffic patterns, device heterogeneity, and environmental factors. More live network experiments and large-scale deployments are needed.
- 2) Dataset scarcity and standardisation:** Insufficient availability of diverse, standardised, and publicly accessible datasets for training and benchmarking AI models, limiting generalisation.
- 3) Overemphasis on model complexity:** Focus on computationally expensive AI architectures, which are challenging to deploy in resource-constrained environments. Lightweight and interpretable models are underexplored.
- 4) Lack of explainability:** AI models often function as black boxes, making it difficult for network operators to interpret decisions and ensure regulatory compliance. Explainable AI techniques are underexplored.
- 5) Scalability challenges:** Difficulty in scaling AI solutions for dense urban environments, massive IoT ecosystems, and large-scale deployments. Research on distributed AI frameworks and federated learning is limited.

6) Energy Efficiency: AI models are computationally intensive, leading to high-energy consumption, which conflicts with sustainability goals. Lightweight and energy-efficient AI models are needed.

7) Real-Time adaptability: Many AI models rely on offline training and fail to adapt to dynamic, non-stationary network environments in real time. Online learning and RL approaches are underexplored.

8) Multi-Objective optimisation: Overemphasis on single QoS metrics without addressing trade-offs between competing objectives such as energy efficiency, reliability, and fairness.

9) Alignment with 6G concepts and emerging applications: Ensure AI methods natively support 6G pillar – semantic communications, digital twins, tactile/holographic services, robotic, and space/NTN – by delivering cross-domain, low-latency, reliable, and interoperable intelligence that scales from edge to cloud.

10) Standardisation and interoperability: Absence of standardised frameworks for AI integration into 5G and 6G networks, leading to fragmented solutions and interoperability challenges.

11) Latency in AI decision-making: Computational complexity in AI models introduces latency, conflicting with ultra-low latency requirements of URLLC applications.

12) Edge intelligence: Limited exploration of lightweight AI models for research-constrained edge devices, which are critical for 6G networks.

13) Security and privacy concerns: AI integration introduces risks such as adversarial attacks, data poisoning, and privacy breaches, which are not adequately addressed.

14) Ethical and social implications: Ethical concerns, such as bias in resource allocation, fairness in network access, and user privacy, are underexplored.

2.8 How This Study Advances AI-Based QoS: Predict-Verify-Adapt

Our work aligns with an end-to-end, field-validated toolkit that prioritises reliability, overhead, and trust. First, a Transformer-based RL scheduler converges quickly (loss $\rightarrow$ 0, rewards rising) and sustains a 100% scheduling success ratio with single-digit-ms latency in live operation – evidence that sequence-aware policies can meet URLLC-like delay while remaining sample-efficient (aligning with where Huang et al. call out DDPG’s slowness).

Second, to reduce control-plane chatter and per-TTI inference costs highlighted by Pang et al., we propose learning with predictive analytics: EGRP forecasts delay from instantaneous throughput and jitter, enabling lower-frequency, forecasting-driven adjustments instead of constant reactive signalling. Empirically, PF delivers tighter, more “learnable” latency (PF vs RR: RMSE 3.20 ms vs 5.99 ms; MAE 0.385 ms vs 0.907 ms; R^2 0.965 vs 0.937), turning predictability into a first-class QoS target that supports tighter SLAs.

Third, our work confronts the black-box/generalisation critique with passive, traffic-only verification: tree, SVM, and LSTM classifiers fingerprint PRIO, HTB and CBQ from packet traces, exposing scheduler “signatures” and offering interpretable rules for policy audit – practical guardrails missing in unconstrained DRL.

Fourth, by operating at the data plane and validating in a busy live WLAN, the approach sidesteps PHY-model fragility and static dataset bias, yielding a replicable method that bridges lab-to-field and can plug into AIOps loops for zero-touch QoS assurance, SLA verification, and slice compliance checks.

Positioning statement: In contrast to strands that either over-model the PHY/MAC layers or rely on unconstrained DRL, this study integrates (i) fast-converging Transformer-RL for real-time scheduling, (ii) EGPR-based “predictable latency” as an explicit QoS objective, and (iii) interpretable, passive verification of enforced queuing policies. Together, these address the literature’s central gaps – overhead, reliability guarantees, interpretability, and external validity – advancing an operational AI pathway for 5G and beyond QoS. Below is a list showing how our work fits within (and advances) AI-based 5G QoS and beyond research:

1) Real-World validation: Our study moved beyond simulation with live-network experiments (busy WLAN/edge gateway), supplying field evidence often missing in AI-native 5G studies.

2) Dataset creation and standardisation: We publish well-described, multi-scenario traces (PF, RR, PRIO, HTB, and CBQ) that can serve as reproducible benchmarks, addressing the scarcity of open, standardised QoS datasets.

3) Lightweight and energy-efficient models: By prioritising Random Forests, Trees, EGPR, and compact LSTM, we demonstrate deployable accuracy on edge-class hardware – countering the trend towards over-heavy models.

4) Explainability: Tree-based models and EGPR feature effects expose interpretable drivers of latency and jitter, directly answering the black-box critique and enabling operator audit.

5) Scalability: Our modular “predict-schedule-verify” loop maps cleanly to distributed MEC/UPF nodes and can be replicated across cells or sites, supporting large-scale rollout.

6) Real-Time adaptability: Online data collection plus fast-converging Transformer-RL enables in-situ policy updates from instantaneous KPIs – closing the gap between offline training and live dynamics.

7) Cross-Layer and Multi-Objective: We jointly consider latency, jitter, throughput, and reliability; PF vs RR analysis quantifies trade-offs so policies meet multi-objective QoS rather than a single metric.

8) Alignment with 6G concepts: Predictive QoS, closed-loop RL control, and passive verification align with AI-native RAN/O-RAN ideals and are pertinent to tactile/holographic and digital-twin use cases.

9) Standardisation and interoperability: Our reproducible evaluation pipeline (common KPIs, folds, and reporting) is a candidate pattern for pre-standard test methods and cross-vendor comparability.

10) Security and privacy: Our methodology for passive, traffic-only scheduler verification supports anomaly surfacing without deep packet inspection; federated/edge deployment paths minimise data movement.

11) Latency optimisation: Our study elevated predictable low latency (tight EGPR errors; PF stability) as a first-class KPI – actionable for URLLC SLAs and near-RT control.

12) Human-in-the-loop: Interpretable models and policy signatures enable operator oversight, change management, and regulatory review of AI actions.

13) Economic feasibility: Lightweight models plus commodity Linux queuing reduce CapEx/OpEx barriers, improving adoption prospects versus monolithic AI stacks.

14) Long-Term stability and robustness: Rolling validation, leakage-safe cross-validation, and stress across the scheduler improve external validity and reduce concept-drift risk.

15) Ethical and social implications: We discuss fairness in prioritisation schemes (who gets low latency, when, and why), providing hooks for policy constraints in RL.

16) Interdisciplinary collaboration: Our work fuses telecom QoS theory, ML/DL and RL control into a practical blueprint that both network and AI teams can adopt.

17) Edge intelligence: Models run on edge gateways/UPF-like nodes, fitting 5G/6G's compute-network continuum and enabling local autonomy.

18) Emerging applications: Predictive QoS and adaptive scheduling are demonstrated in forms directly transferable to URLLC-class domains (industrial IoT, telemedicine, and autonomous vehicles)

2.9 Conclusions and Future Directions

AI-based methods – spanning classical ML, DL, and DRL – have clearly shown great promise in addressing the twin challenges of resource allocation and traffic prioritisation for 5G URLLC networks. Through this literature review, we have seen how these approaches overcome the limitations of traditional schemes by introducing adaptability, predictive power, and intelligent decision-making into the network scheduling function.

2.9.1 Summary of Findings

Recent research indicates that AI-driven resource allocation and prioritisation schemes can significantly enhance URLLC QoS. ML approaches, such as Q-learning and bandit algorithms, can dynamically adjust scheduling policies based on experience with network conditions, whereas fixed algorithms cannot. DL models can infer complex patterns (e.g., in traffic or channel conditions) and output near-optimal allocation decisions in real time, thereby meeting URLLC's tight timing requirements [46], [60]. DRL agents go a step

further by continually learning and refining a scheduling strategy that balances reliability, latency, and efficiency in a multi-user environment, often outperforming human-designed strategies. For instance, DRL-based schedulers have achieved URLLC packet success rates extremely close to 100% while minimising impact on other services [60], [65], effectively pushing the frontier of the latency-reliability-throughput trade-off. AI approaches shine particularly in high-complexity scenarios—such as multi-cell coordination, vehicular networks, or heterogeneous traffic mixes – where no simple manual solution exists. They learn to coordinate distributed resources (as seen in federated and multi-agent setups) and adapt to changes (as demonstrated by meta-learning integrated methods) in ways that legacy systems would struggle with [60], [65].

2.9.2 Challenges and Considerations

Despite the success of AI-based approaches, there remain vital considerations before these AI techniques are ubiquitously deployed in operational networks. One issue is training and convergence – some DRL algorithms may require extensive training or may converge slowly if the environment is highly complex. Techniques such as transfer learning and meta-learning are mitigating this by enabling models to adapt or reuse knowledge more quickly [27], [28], [71]. Another consideration is stability and safety – in a live network controlling critical URLLC traffic (such as remote surgery), we need guarantees. In theory, AI-based models can behave unpredictably outside their training distribution. The research community is actively addressing this issue by incorporating domain-specific knowledge and constraints into learning algorithms. Hybrid approaches (optimisation + learning) provide a safety net, ensuring the AI policy continues to adhere to critical rules. Recent work on explainable RL and anchored learning processes is improving the interpretability of learned schedulers, making their behaviour more transparent and verifiable [27], [28], [71].

Furthermore, runtime computational overhead is a factor. However, many papers report real-time feasibility, the network infrastructure will need sufficient computational resources (such as edge compute units) to run these algorithms (especially for DRL training updates or large neural networks). Fortunately, the trend in telecommunications is towards virtualisation and edge computing, which aligns well with deploying AI alongside base stations (e.g., in an edge cloud that hosts the RAN intelligent controller).

2.9.3 Outlook for 6G and Beyond

As we look ahead, URLLC is expected to play an even bigger role in 6G networks, expanding into applications such as holographic telepresence, tactile Internet, and advanced industrial robotics, which have even more stringent or multifaceted QoS requirements [22], [27], [28]. The literature suggests that AI will not only be a handy tool but also a necessity for managing these demands. Traditional schemes do not scale to the complexity of controlling, for example, a network of autonomous drones (a URLLC use case in 6G), where each drone requires millisecond updates and the network must avoid a single point of failure. AI-based models, especially DRL, are likely to be incorporated into network controllers to handle such tasks adaptively. Furthermore, the concept of “network intelligence” is gaining momentum: future networks might have self-learning

capabilities at various levels (from the physical layer to the application layer), coordinating to ensure end-to-end URLLC performance. This could encompass multi-domain resource allocation (for spectrum, compute, and storage) orchestrated by AI tools – some research studies are already discussing joint communication and computing resource allocation via DRL for mission-critical IoT [22], [27], [28].

Another direction is the combination of AI with edge computing and network slicing frameworks (such as O-RAN). Several works cited (for instance, Sohaib *et al.*, 2024) [82] are aligned with the O-RAN concept, placing DRL agents in the near-real-time RIC (RAN Intelligent Controller) to control RAN slices for eMBB and URLLC. This is also being prototyped in the industry. We can expect standards to evolve to allow AI agents to interface with RAN scheduling. Federated and distributed learning will be key to leveraging data from a massive number of devices in IoT scenarios while preserving privacy and ensuring scalability – the success reported in federated DRL for URLLC IoV [22], [27], [28], and [82] is likely to extend to other verticals (smart grid URLLC, remote healthcare networks, etc.).

2.9.4 Concluding Remarks

The past few years have solidified the role of AI in enhancing QoS for 5G networks requiring URLLC by addressing pressing challenges, such as ultra-low-latency scheduling, reliability assurance, and multi-service coordination. AI-based resource allocation schemes have achieved what was previously considered extremely difficult, if not impossible, in 5G networks. They represent a paradigm shift from static, one-size-fits-all networking to autonomous, context-aware networks that can learn and self-optimize. As we move towards 6G, which will demand even more from wireless networks (in terms of latency, reliability, flexibility, availability, and efficiency), the importance of these AI-driven schemes will only grow. Future research will likely focus on ensuring these methods are robust, secure, and easy to integrate into real-world systems, as well as on extending them to cover end-to-end URLLC (from radio to transport and the core network). Given the trajectory so far, we can be optimistic that the combination of ML, DL, and DRL will continue to drive innovations that make ultra-reliable and low-latency communication a ubiquitous reality across emerging network applications [15], [17], [25], [60], [62].

Chapter 3: Research Methodology

3.1 Introduction

This chapter discusses the research methodology adopted to achieve the aims and objectives established for this research project. The chapter outlines a comprehensive research methodology for developing and evaluating AI-based tools. These tools are designed to selectively prioritise real-time IP packets to improve QoS in 5G WSNs. By leveraging powerful AI techniques, such as ML, RL, and DL, this methodology aims to guide this research study to optimise IP traffic prioritisation, resource allocation, and network performance in the context of 5G networks.

The research methodology encompasses several vital stages, starting with an extensive literature review of telecommunications and AI to understand the current state of the art in IP traffic management, prioritisation schemes, and AI applications in 5G networks. This step follows a rigorous formulation of the problem and definition of the objective, clearly outlining the research goals, performance metrics, and evaluation criteria.

A detailed system modelling and simulation approach is described to facilitate the design, development, and testing of AI-based models that improve QoS in 5G networks. This involves creating realistic network simulation environments that incorporate relevant AI techniques and generating or collecting representative data for training and validation.

The chapter then delves into the core aspects of AI model development and training, exploring architectures and algorithms suitable for traffic classification, prediction, prioritisation, and resource allocation. Additionally, techniques such as ML, deep neural networks, and RL agents are discussed in detail, along with their strengths and limitations.

Rigorous evaluation and testing methodologies are presented, encompassing experiment- and simulation-based performance analysis, sensitivity analysis, and robustness testing under diverse network conditions and traffic patterns. The chapter also highlights the importance of optimisation and refinement processes, enabling iterative improvements to the AI models and prioritisation schemes based on the evaluation results.

Finally, the chapter emphasises the significance of proper documentation, validation, and dissemination of research findings. This process may include guidelines for reproducible documentation, peer review and external validation, as well as avenues for presenting and publishing research outcomes at relevant conferences and journals.

The chapter presents real-world examples, case studies, and potential challenges, offering a practical perspective on the research methodology. Furthermore, it addresses ethical considerations, potential biases in AI models, and the broader implications of the proposed solutions for various stakeholders within the 5G ecosystem.

Efficient prioritisation of real-time IP packets is critical for guaranteeing QoS requirements, such as latency and high reliability, in time-sensitive applications within the scope of 5G WSNs requiring URLLC. Traditional packet prioritisation schemes may not effectively handle the diverse and dynamic nature of real-time IP traffic in 5G WSNs,

leading to potential QoS degradation. Leveraging AI techniques can deliver intelligent, adaptive solutions for packet prioritisation, enabling real-time decision-making and optimisation based on network conditions and application requirements.

The development of AI-based tools can significantly improve QoS parameters, such as throughput, packet loss, latency and reliability, enhancing the performance of real-time applications in 5G WSNs. Hence, the proposed research aims to address the critical challenges of QoS provisioning in 5G WSNs by exploring the potential of AI techniques for intelligent scheduling and resource allocation, with far-reaching implications across domains such as industrial automation, telemedicine, and IoT.

The outcomes of this research study can contribute to advancing QoS management strategies in 5G WSNs, thereby fostering the development of more efficient and reliable real-time communication systems.

3.2 Research Approach

A multifaceted research design is proposed to address the research aims and objectives and to develop effective AI-based schemes that improve QoS in 5G WSNs. This design incorporates both simulation-based and real-world experimentation, leveraging the strengths of each approach to ensure comprehensive evaluation and validation.

3.2.1 Philosophical Assumptions

This investigative study adopts a pragmatic worldview, focusing on finding practical solutions to real-world problems and employing multiple methods to address the research questions. A pragmatic approach is suitable for this research, as it aims to develop an AI-based solution to enhance QoS in 5G WSNs—a practical problem that requires a combination of theoretical and empirical methods.

A vital aspect of the pragmatic worldview is that it endorses drawing on both quantitative and qualitative assumptions when conducting research in telecommunications, particularly in areas such as QoS for communications networks and QoE for end users. For this investigative study, quantitative data were collected and analysed through simulation experiments and testbed measurements to evaluate the performance of the proposed AI-based QoS solutions under various network conditions and scenarios. Nevertheless, qualitative insights also played a role by examining the practical requirements, constraints, and use cases in deploying such a solution in real-world 5G wireless sensor network environments.

The pragmatic methodology enables a flexible integration of deductive reasoning from existing theories in fields such as ML, DL, RL, optimisation, and computer networking, as well as an inductive understanding derived from observing phenomena and patterns in empirical data. This combination of deductive and inductive logic aligns well with the exploratory yet applied nature of developing novel AI-based techniques to tackle the multifaceted QoS challenges faced by 5G networks and beyond. The pragmatic philosophy focuses on “what works” in practice, making it well-suited for this exercise in designing and validating innovative AI-driven solutions.

3.3 Research design

This research study employed experimental designs to develop, implement, and test various AI techniques to improve QoS in 5G WSNs.

The experimental design enabled the systematic manipulation of variables, including network conditions, traffic patterns, and QoS parameters, and the observation of their effects on the performance of the proposed AI techniques.

3.3.1 Simulation and Modelling Approaches for Comparison of scheduling algorithms using ML/DL models with data captured from an operational network

3.3.1.1 Experimental Environment Setup

This study utilised Python and MATLAB to emulate established scheduling algorithms, such as PF and RR, and a separate experiment implemented PRIO, HTB, and CBQ Linux traffic control (tc). The data collected through these experiments were used in conjunction with ML/DL models for comparison purposes. This study designed an experimental environment to accurately represent the operational conditions of 5G WSNs. Thus, this involved experimenting with different network topologies, varying numbers of users (or nodes), and channel and traffic conditions. Additionally, key network performance indicators, including throughput, latency, packet loss, and fairness, were evaluated to compare the performance of the scheduling algorithms. The choice of these metrics aligns with the objectives of QoS improvement in 5G WSNs. Below is a brief description of the scheduling algorithms used in the experiments:

1) Proportional Fair (PF): Implement a Python-based simulator for the PF scheduling algorithm that balances fairness and throughput by prioritising users with relatively better channel conditions while preventing starvation of users with poorer channel conditions [88], [91].

2) Round Robin (RR): Implement the RR scheduling algorithm in Python that cyclically serves users, providing equal time slices to each user regardless of channel conditions or flow traffic characteristics.

3) Linux Traffic Control (PRIO, HTB, CBQ): Implements PRIO, HTB, and CBQ scheduling algorithms using Linux tc to capture granular network data, and utilise the resulting traces to fingerprint traffic patterns for scheduling classification via ML and DL models.

4) Dataset Generation: Extensive experimentation with the PF, RR, PRIO, HTB, and CBQ algorithms was conducted to create a comprehensive dataset capturing channel conditions, user distributions, and resulting QoS metrics. [88], [90], [91].

3.3.1.2 Experimental Network Environment

This study implemented a wireless testbed comprising a router-centric WLAN hosting four Raspberry Pi endpoints on the private IPv4 subnet 192.168.1.0/24, statically assigned the addresses 192.168.1.140-192.168.1.143 (device_1-device_4). Proportional fair and round robin schedulers executed on a controller host within the same WLAN and issued 64-byte ICMP echo probes at 2 Hz (0.5 s inter-probe interval) using a 2 s timeout over

continuous 2-hour measurement sessions. For each scheduling round, the instrumentation recorded per-target latency and jitter, estimated packet loss from probe outcomes, and computed an ICMP-based effective probe rate (bytes delivered per unit time) from the fixed payload size and successful replies. All measurements were timestamped, labelled by device identifier, and written to human-readable logs and CSV datasets to support reproducible QoS characterisation of intra-WLAN paths.

To contextualise local behaviour against upstream reachability, the RL scheduler expanded the target set to include the four LAN endpoints and a fixed set of public DNS resolvers. The RL loop used the same 64-byte payload with a 1 s timeout, therefore sampling both local (WLAN) and WAN paths through the same access router and enabling comparative estimations of latency and success ratio across the two transit regimes.

All schedulers used `icmplib` in unprivileged ICMP mode, maintaining consistent payload sizing and probe cadence across PF, RR, and RL conditions. This uniform instrumentation ensured methodological parity when contrasting intra-WLAN performance with end-to-end connectivity to public resolvers.

The testbed was also utilised to evaluate PRIO, HTB, and CBQ schedulers. The data collected from these experiments were used to characterise their traffic fingerprints and to classify the schemes using ML and DL techniques.

3.3.1.3 Data Collection and Dataset Creation

To evaluate conventional scheduling algorithms using ML and DL models, we developed a dataset derived from an experimental testbed. This testbed was designed to emulate a 5G WSN by leveraging a configurable WLAN infrastructure. The WLAN was set up to reproduce key 5G WSN characteristics, such as specific traffic models and network densities. The resulting dataset captured these realistic operational dynamics, providing a solid foundation for the comparative analysis of the scheduling algorithms [18], [30], [92], and [93].

Following the emulation of the 5G WSN on the WLAN testbed, a systematic data collection process was initiated. Performance data for each scheduling algorithm were meticulously gathered across a spectrum of network conditions and configurations. This raw data was then structured into a comprehensive dataset, where each entry consists of input features, such as the number of users, prevailing channel conditions, and packet lengths, aligned with corresponding target variables. These variables included key performance indicators (KPIs) such as delay, throughput, and packet loss, along with a label identifying the specific scheduling algorithm used. Finally, this curated dataset served as the foundation for training and evaluating our ML models, which were designed to predict performance metrics and classify the optimal scheduling algorithm for a given set of network conditions [18], [30], [92], [93].

3.3.1.4 Rationale for Applying ML Models to Predict and Classify Live Network Data under QoS Schemes

This study employed machine-learning models, including decision trees and EGPR, to predict network key performance indicators (delay, throughput, and packet loss) and to classify scheduling algorithms based on input features derived from simulated data. Below, we explain some of the reasons behind the choice of ML models to classify and predict network traffic and protocols [39], [61], [62]:

1) Capturing Complex, Non-linear Relationships in QoS

a) Traditional methods struggle: QoS schemes often involve intricate interactions between various network parameters (such as latency, throughput, bandwidth, jitter, and packet loss) and application requirements. Additionally, traditional analytical models or rule-based systems may struggle to capture the complex, non-linear relationships that govern these interactions, particularly as network architectures and traffic patterns become increasingly complex [39], [61], [62].

b) ML excels in non-linearity: ML models, such as decision trees, ensemble methods (Random Forest, Gradient Boosting), and Gaussian Processes, are adept at learning and representing this non-linearity directly from data. These models can uncover hidden patterns and dependencies that might be difficult or impossible to define explicitly using traditional methods [39], [61], [62].

2) Handling High Dimensionality and Feature Interaction

a) Many influencing factors: Network performance under QoS mechanisms is influenced by numerous factors, resulting in a high-dimensional feature space. The latter includes network parameters, application-specific characteristics, traffic profiles, and even hardware configurations [39], [61], [62].

b) ML navigates high dimensions: ML models are designed to handle high-dimensional data and automatically learn the relevant feature interactions that impact QoS. ML algorithms can identify which factors are most influential in determining performance and how they combine to produce specific outcomes, providing a comprehensive understanding of QoS behaviour [39], [61], [62].

3) Adaptability to a Dynamic Network Environment

a) Networks are constantly changing: Modern communications networks are highly dynamic, with fluctuating traffic loads, evolving application demands, and changing network topologies. Often based on static assumptions, traditional models can quickly become outdated in such environments [39], [61], [62].

b) ML enables continuous learning: ML models can be trained and updated continuously with new data, allowing them to adapt to the changing network conditions and maintain their accuracy over time. This adaptability is crucial for ensuring that QoS schemes remain effective in the face of evolving network dynamics [39], [61], [62].

4) Accurate Prediction and Classification for Proactive Management

a) Anticipating performance: Accurate network performance prediction under different QoS mechanisms is crucial for proactive network management. This process involves capacity planning, resource allocation, and meeting Service Level Agreement (SLAs) [39], [61], [62].

b) ML empowers prediction: ML algorithms, particularly those that offer uncertainty quantification, such as Gaussian Processes, can provide accurate forecasts of Key Network Performance Indicators (KPIs) such as latency, jitter, packet loss, and throughput, along with a measure of confidence in those predictions. This ability enables network operators to anticipate potential issues and take proactive measures to maintain or improve QoS [39], [61], [62].

c) Precise traffic classification: Accurate network traffic classification into different QoS classes is fundamental for effective traffic prioritisation and resource management [39], [61], [62].

d) ML refines classification: ML models can be trained to accurately classify traffic, considering a wide range of features beyond simple port numbers or IP addresses. This can lead to more granular and effective QoS control, ensuring that different application types receive the appropriate level of service [39], [61], [62].

3.3.1.5 Model Selection Rationale

Table 3.1 summarises the rationale for model selection for all algorithms used on our network traffic datasets. For each model (Decision Tree, Random Forest, SVM, EGPR, CNN, LSTM, and Transformer), we list task fit, a justification, three key hyperparameters, core strengths, and caveats. Detailed rationales and MATLAB/Python workflows are available in the Appendix.

Table 3.1: Model Rational and Fit (Used on Packet/Flow/KPI Datasets).

Model	Best for task(s) in this study	Why used here	Key Knobs	Strengths	Caveats
Decision Tree	Interpretable packet/flow classification; low-latency rules for QoS	Clear per-feature splits over mixed categorical/numeric traffic fields give transparent scheduling rules.	Max depth; Min sample/leaf; Pruning (ccp_alpha)	Human-readable; handles non-linear interactions; fast inference	High variance/overfit if deep; stability improves by pruning/ensembles
Random Forest	Robust classification under noisy bursty traffic	Bagged trees reduce variance and cope well with heterogeneous feature/missing values	n_estimators; Max depth; Max_features	Strong baseline accuracy; out-of-bag estimates; feature importance	Less interpretable; larger models can be compute-heavy
SVM	Small/medium, high-dimensional	Kernel trick separates complex protocol/traffic classes with	C; Kernel (linear/RBF); Gamma	Effective in high-D; flexible kernels	Sensitive to scaling/hyperparameters;

	curated traffic sets	strong small-N generalisation.			Training slows on very large training samples.
EGPR	KPI regression (latency/throughput) with calibrated uncertainty	Produces mean + variance for QoS planning/SLAs; good when labelled data are limited.	Length-scale; Signal variance; Noise σ	Uncertainty quantification; non-parametric; Small-N friendly	$O(N^3)$ training; memory-heavy-needs approximations for big N
CNN	Image-based traffic representations (e.g., traffic-statistics, heatmap/plots)	Learn spatial patterns from transformed traffic images; supports transfer learning.	Learning rate; filters; Batch/Epochs	Strong pattern recognition; GPU-accelerated	Needs careful input transforms/augmentation; higher compute
LSTM	Temporal QoS/traffic sequences; KPI forecasting	Captures short/long-term dependencies in evolving traffic series.	Hidden units; Sequence length; Learning rate	Suitable for variable-length sequences; sequence awareness	Data preparation (windowing) required; tuning can be slow
Transformer	Long-range multi-feature sequences; parallel training	Self-attention models efficiently capture global context across traffic features.	Layers/heads; d_model; Dropout	Scales well; expressive representations; fast parallel training	Many hyperparameters; benefits from GPU and larger datasets

To complement Table 3.1, Table 3.2 maps each model to practical capabilities relevant to network traffic datasets. We use a compact scoring system: Yes (Strong fit), More or less (workable with care), and No (not a natural fit).

Table 3.2: Capabilities Matrix (Network Traffic Dataset).

Model Capability	Decision Tree	Random Forest	SVM	EGPR	CNN	LSTM	Transformer
Interpretability (human-readable)	Yes	More or less	More or less	More or less	No	No	No
Handles non-linear boundaries	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Small-N friendly	Yes	More or less	Yes	Yes	No	More or less	No
Sequence modelling (temporal dependencies)	No	No	More or less	More or less	More or less	Yes	Yes
Native uncertainty estimates	More or less	More or less	More or less	Yes	No	No	No
Real-time inference latency (fast at	Yes	More or less	More or less	More or less	More or less	More or less	More or less

prediction time)							
Compute/GPU demand (training), lower is better	Low	Moderate	Moderate	High	High	Moderate	High

To connect each algorithm to its role in the study, Table 3.2 maps use-cases to data sources, objectives, and models. Detailed MATLAB and Python workflows are provided in Appendix A, B, C, and D (Pages 335-345).

Table 3.3: Model Utilisation in This Study.

Experiment	Data Source	Objective	Model
(1) Comparative Evaluation of Scheduling Disciplines (PF/RR)	Dataset collected from a live WLAN environment.	Collect a comprehensive QoS dataset to compare PF and RR scheduling disciplines. Develop predictive capabilities(classification/regression) based on network behaviour.	ML regression models (EGPR) for comparing and predicting QoS metrics, such as delay and throughput.
(2) Classification and Prediction of QoS Algorithms (PRIO, HTB, CBQ) – Classification Task	Network traffic data captured from an operational network using Wireshark and a Linux router configured with QoS queuing disciplines (PRIO, HTB, CBQ)	Classify and predict scheduling algorithms (PRIO, HTB, CBQ) based on network traffic patterns/signatures.	ML classification models (Decision Trees, SVM)
(3) DL (CNN) for Image-based Traffic Classification	Network traffic data (PRIO, HTB, CBQ, and No_QoS configuration) converted into standardised RGB scatter plots.	Leverage the strengths of CNN architectures to process and understand complex network traffic patterns through image-based representations. Perform classification across four classes.	CNN
(4) DL (CNN-LSTM) for Image-based Traffic Classification	Network traffic data (PRIO, HTB, CBQ, and No_QoS classes) converted into image format.	Classify scheduling algorithms based on an Image-Based representation of network traffic and compare their performance with that of the pure CNN model.	Hybrid CNN-LSTM network (uses CNN layers for spatial feature extraction followed by LSTM units)
(5) Delay Prediction Models (Prediction Task)	Dataset collected from a live WLAN employing a PF scheduling algorithm. Data consists of	Develop models to predict network delay (regression task). Compare the efficacy of temporal sequence models against a non-temporal baseline.	Hybrid Linear-LSTM-Transformer model (sequence-based, window length 10).

	time-stamped QoS metrics.		LSTM model (sequence-based, window length 10). RF Regressor (baseline, non-sequential based).
(6) Transformer-based RL Agent for Network Traffic Scheduling	Live operational network environment	Optimise network traffic scheduling to minimise latency (RTT) and maximise transmission success.	Transformer-based RL Agent (uses policy gradient loss)

3.3.1.6 Common Modelling Pipeline for ML/DL/DRL

1) Data and labels: Assemble packet, flow, or KPIs datasets (live or synthetic); de-duplicate, time-align, and document label sources.

2) Splits: Use 70% of the data to train, 15% for validation, and 15% for testing purposes (or k-fold for small data samples); grouped or stratified by device/session to avoid leakage.

3) Preprocessing: Median impute numeric; z-score scale for SVM and EGPR; one-hot encode categorical; persist train-fit transforms.

4) Traffic-specific transforms: (i) Sequences sliding windows. (ii) Images: convert statistics/heatmaps (0-1 tensors with standard resize).

5) Imbalance handling: Class weights (preferred) or restrained resampling; report macro-average metrics.

6) Hyperparameter tuning: Same search budget across models (Bayesian/Optuna or MATLAB optimiser); early stopping on validation metrics.

7) Training settings: Fix seeds; batch size, epochs, learning rate for DL, and max depth for trees (detailed explanations for training settings are provided in Appendix A)

8) Evaluation metrics: For classification, use accuracy, F1 (macro), and inference latency (ms). For regression tasks, use RMSE, MAE, and R^2 .

9) Selection criteria: Choose by primary metric; tiebreak with inference latency, stability (variance across folds), and interpretability.

10) Reproducibility: Log configurations, seeds, and metrics to CSV/JSON; freeze preprocessing + model artefacts for reuse.

11) Details: Rationale summaries for model selection are presented in Tables 3.1; detailed MATLAB and Python implementation workflows are provided in Appendices A, B, C, and D.

3.3.1.7 Feature Selection

This study identified features relevant to QoS and scheduling algorithms, which significantly affect network performance. These features include the number of users in

the system, channel conditions, packet lengths, traffic load, data rates, delay, throughput and packet loss.

The choice regarding the number of users is fundamentally rooted in its direct impact on the network load and resource contention. As the number of users increases, the demand for bandwidth and processing power also rises. This factor can lead to congestion, increased queuing delays, and potentially higher packet loss, especially if the network infrastructure is not adequately provisioned. For ML models, the number of users is a crucial indicator of the overall network load and can be used to predict performance degradation under different user population scenarios. Decision trees and EGPR can effectively incorporate this feature to model its relationship with QoS metrics [37], [39], [61], [102]. Decision trees can create splits based on the number of users. Meanwhile, EGPR can model the non-linear relationships between user counts and performance metrics, providing uncertainty estimates for different user load scenarios.

Channel conditions represent the quality of the communication medium, which is particularly relevant in wireless networks. Factors such as interference, fading, noise levels, and signal strength directly affect data transmission rates, error rates, and overall link reliability. Additionally, poor channel conditions can result in reduced throughput, increased latency, and higher packet loss. Including channel condition as a feature allows ML models to learn the complex relationships between channel quality and network performance. For instance, decision trees can use this feature to classify traffic based on expected performance under different channel conditions, while EGPR can model how performance metrics vary continuously with changes in signal strength or interference levels. Furthermore, EGPR's ability to handle uncertainty is valuable here, as channel conditions can fluctuate rapidly [37], [39], [61], [102].

Packet lengths are a fundamental characteristic of network traffic, directly influencing bandwidth consumption and processing overhead. Large packets consume more bandwidth per transmission; however, larger packets may be more efficient in terms of header overhead. Scheduling algorithms often take packet lengths into account. Including packet length as a feature enables ML models to learn how different packet sizes affect QoS. Decision trees can create rules based on packet length thresholds to differentiate between traffic types and predict their performance. Furthermore, the EGPR can model the relationship between packet length and metrics such as delay or throughput, providing insights into optimal packet sizes for different applications or network conditions [37], [39], [61], [102].

The traffic load represents the overall demand placed on the network. It is a composite measure expressed in bytes per second, packets per second, or another relevant unit. High traffic load can lead to congestion, increased latency, and packet loss if the network's capacity is exceeded. For ML, traffic load is crucial for predicting network performance under different demand scenarios. Additionally, both models can utilise this feature, with decision trees potentially partitioning the data based on load thresholds. At the same time, EGPR can capture the likely non-linear relationship between traffic load and QoS metrics, quantifying prediction uncertainty across different load levels [37], [39], [61], [102].

The data rates, delay, throughput, and packet loss are not only features but also key performance indicators (KPIs) in their own right. The reason for including these features is to model the relationship between these metrics. For example, past data on data rates, delays, and packet loss can be used to predict future throughput. This is particularly useful for EGPR, which can capture complex dependencies between these variables. Furthermore, using these metrics as features at the per-packet level can enable the ML algorithm to classify or predict them at the per-flow or per-user level [37], [39], [61].

3.3.1.8 Feature creation

For problems involving the analysis of a dynamic network scheduler, effective feature creation is crucial and extends far beyond logging raw measurements. The primary goal is to transform sparse, instantaneous data points into a rich, stateful representation of the network environment and the scheduler behaviour. This approach involves several key strategies. First, raw metrics, such as delay, must be transformed into standardised performance indicators, such as instantaneous throughput, to provide a comparable measure of efficiency. Second, temporal context is vital; features such as exponential moving averages and rolling statistics like jitter are engineered to capture a device's historical performance and stability, preventing models from overreacting to transient fluctuations [37], [39], [61], [102], [103].

Most importantly, the scheduler's internal logic must be exposed by creating features such as the PF metric, which directly represents the value used to make a decision. Finally, the measurement process itself is modelled using indicator features to explicitly handle the structural sparsity, ensuring that a lack of data is interpreted as a "not measured" state rather than missing values. Together, these engineered features create a comprehensive dataset that captures not just what happened, but also how and why [37], [39], [61], [102], [103].

3.3.2 Framework for Designing and Implementing AI Models (ML, DL, DRL) for Classification/Prediction of Scheduling Algorithms with Data Captured from an Operational Network

3.3.2.1 Experimental Setup

This study established a live network environment to test various scheduling disciplines, including Priority-Based Scheduling, Hierarchical Token Bucket (HTB) queuing, and Class-Based Queuing (CBQ). Routers, switches, and servers were configured to support these algorithms, and Wireshark was used to capture network data that was subsequently used to create datasets for analysis [2], [41], [44], [62], [70], [91].

3.3.2.2 Scheduling Algorithms

1) Priority-based scheduling (PRIO): This algorithm schedules packets based on predefined priority levels. Each packet is assigned a priority, and higher-priority packets are transmitted before lower-priority ones. This algorithm aims to ensure QoS by giving time-sensitive or critical traffic preferential treatment [44].

2) Hierarchical Token Bucket (HTB): HTB is a classful queuing discipline that allows hierarchical configuration of classes with different priorities and rate limits. This

algorithm utilises token buckets to regulate the transmission rate of packets, ensuring that each class receives a guaranteed share of the available bandwidth while allowing borrowing of unused bandwidth from other classes [2].

3) Class-Based Queuing (CBQ): Another classful queuing discipline, CBQ divides network traffic into classes based on predefined rules (e.g., source/destination IP addresses, protocols, or ports). Each class is assigned a configurable share of the available bandwidth, and packets are scheduled according to class priority and weight factors [91].

3.3.2.3 Wireless sensor network traffic generation

The methodology employed in this study involved implementing a wireless sensor network testbed using Raspberry Pi devices. More precisely, one Raspberry Pi was designated as a router, functioning as the sink node, while the remaining three devices were configured as sensor nodes, acting as client devices. This setup was designed to emulate a realistic wireless sensor network environment, providing a controlled platform for subsequent experimentation and analysis [1], [2], [41].

Furthermore, to achieve this, the Raspberry Pi devices were configured to communicate wirelessly, establishing a network infrastructure that mirrored the operational characteristics of a typical wireless sensor network. Additionally, the sensor nodes were programmed to generate and transmit various types of network traffic. This approach generated data transmission patterns commonly observed in real-world wireless sensor applications, ensuring the testbed's relevance and applicability to practical scenarios [1], [2], [41].

The following traffic scenarios were implemented to generate realistic network traffic patterns:

1) Video streaming: YouTube videos were streamed to client devices to simulate video traffic, which typically involves large data transfers and varying bit rates.

2) Network diagnostics: Ping and traceroute commands were executed from client devices to generate ICMP diagnostic traffic, which helps characterise the network's responsiveness and routing behaviour.

3) Social media: Browsing activities on popular platforms were conducted on the client's devices to mimic real-world social media traffic patterns.

4) Email: Clients were installed on the client machines to send and receive emails, simulating email traffic with varying message sizes and attachment loads.

5) The following techniques can be applied to diversify the network conditions:

a) Varied traffic intensity: The number of concurrent traffic flows, such as video streams, social media sessions, ping requests, and traceroute requests, was varied to achieve a range of network load conditions from low to high.

b) Controlled packet loss: Packet loss was simulated by configuring queuing disciplines to mimic real-world network congestion or link-quality degradation.

c) Network topology changes: The network topology was dynamically reconfigured by connecting and disconnecting client machines to simulate dynamic scenarios such as client mobility, network expansion or contraction.

d) Store the captured network traffic data and the corresponding ground truth labels for the active scheduling algorithms for further preprocessing and model training.

3.3.2.4 Data collection in Live Network Environment

Network traffic data was captured using the graphical network protocol analyser Wireshark and the command-line packet analyser tcpdump. These tools were employed to monitor and record network communications within the implemented network infrastructure testbed. Specifically, Wireshark provided a user-friendly interface for detailed analysis and visualisation of the captured traffic, while tcpdump was utilised for its efficiency in capturing raw packet data directly from the network interface [6], [18].

The data captured focused on characterising the behaviour of different scheduling algorithms implemented within the network. Additionally, a range of relevant features and attributes was extracted from the captured network traffic. These included, but were not limited to, packet arrival times, packet sizes, source and destination IP addresses, and port numbers. Collecting these attributes was crucial for a thorough understanding of the performance and characteristics of each scheduling algorithm. Furthermore, data were collected over extended periods and across multiple experimental runs to ensure the generation of a representative, diverse dataset, mitigating the impact of transient network conditions and providing a robust basis for subsequent analysis [6], [18], [21].

3.3.2.5 Data preprocessing

The collected network traffic data underwent rigorous preprocessing to ensure its suitability for subsequent analysis and model development. Initially, the raw data underwent a cleaning process to address common data quality issues. This involved identifying and handling missing values, either imputing them using appropriate statistical methods or removing them if deemed non-informative. Outliers, which could potentially skew analysis or model training, were detected and mitigated using techniques such as standardisation. This meticulous cleaning process was crucial for ensuring the integrity and reliability of the dataset [62], [102], [103].

Furthermore, after the cleaning stage, feature engineering was performed to enhance the data's discriminative power. This entailed transforming existing features or extracting new ones from the raw network traffic data. For instance, packet inter-arrival times can be calculated from packet arrival timestamps, or statistical measures such as the mean and variance of packet sizes can be derived. The selection of appropriate features was guided by domain knowledge and aimed at capturing the most relevant aspects of network behaviour. The pre-processed dataset was partitioned into distinct training, validation, and test sets. This partitioning followed established best practices for ML and DL model development, ensuring that the models were trained on one subset of the data, validated on another to tune parameters, and evaluated on a held-out test set to assess their generalisation performance. This rigorous preprocessing methodology ensured the

creation of a high-quality dataset suitable for robust and reliable analysis [62], [102], [103].

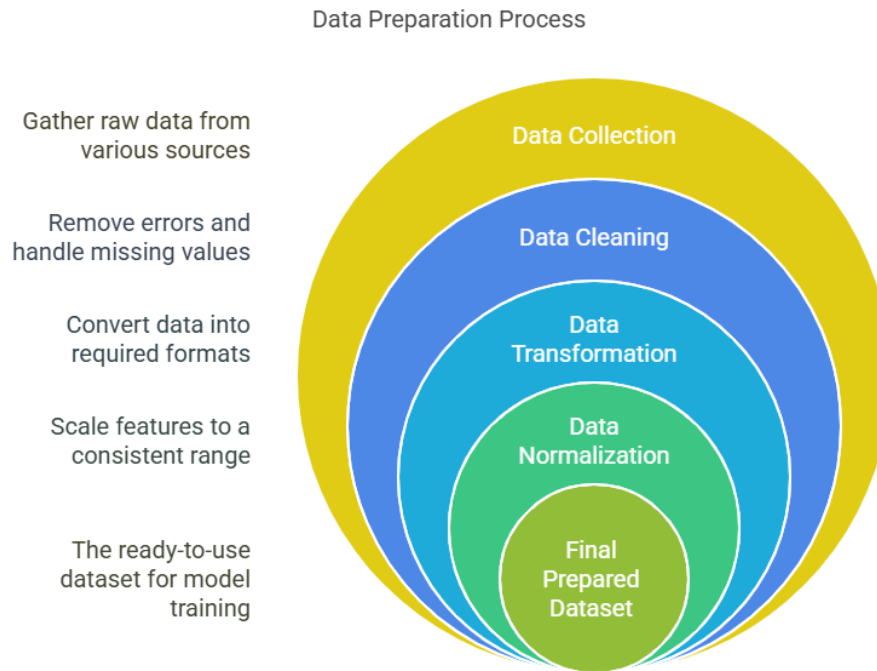


Figure 3.1: The figure illustrates the design workflow to achieve the objective.

Figure 3.1 illustrates the various stages involved in preparing data for training ML and DL models. It shows the sequential nature of data preparation, where each stage builds upon the previous one to refine and prepare the data for its intended use [62], [102], [103].

3.3.2.6 Network traffic data conversion for CNN Model

The following describes the data preprocessing steps for the CNN DL model used in this study:

- 1) Network traffic data was collected and stored in separate datasets for each scheduling algorithm implemented in the live network environment.
- 2) MATLAB statistical tools were employed to analyse features in each dataset, such as packet lengths by source or destination IP address.
- 3) Using MATLAB's plotting capabilities, the analysis results were stored as graphical representations (e.g., scatter plots, histograms, or other statistical charts).
- 4) These graphical representations effectively captured the statistical properties and patterns of network traffic data for each scheduling algorithm.
- 5) These generated graphical representations were stored as image files, with each image representing the network traffic characteristics of a specific scheduling algorithm.

6) Separated folders were created for each scheduling algorithm, containing the corresponding image files generated from the statistical analysis of its network traffic data.

This approach of converting network traffic data into images using statistical analysis and visualisation tools could represent an innovative way to leverage the pattern recognition capabilities of DL models. Additionally, by representing the data as images, it is possible to leverage the strengths of CNNs or other DL architectures designed for image processing tasks [62], [102], [103].

The generated images capture the statistical properties and patterns of network traffic data, enabling the distinction between different scheduling algorithms. Using MATLAB's statistical and plotting capabilities ensures a systematic, reproducible process for data conversion and image generation.

This approach enables the use of transfer learning techniques or pre-trained models for image classification, potentially improving model performance and reducing training time compared to training from scratch.

3.3.2.7 Rationale for Using DL in Scheduling Algorithm Classification and Prediction

DL models provide a powerful means of capturing the complexity of communication network scheduling algorithms. Scheduling algorithms operate under dynamic conditions, where network variables such as data traffic loads, channel conditions, and user demands can change rapidly. Traditional analytical approaches or simpler ML methods may struggle to capture such intricate dependencies, particularly in high-dimensional settings involving multiple scheduling policies and network parameters. By contrast, DL models excel at recognising complex patterns in large-scale data, making them well-suited for tasks such as classification and scheduling decisions, as well as for predicting scheduling outcomes in real time [25], [26], [31].

Another vital rationale lies in the representational power of deep neural networks. Complex scheduling algorithms often involve convoluted interactions between inputs such as traffic arrival rates, buffer states, and user QoS requirements. Deep architectures—especially those employing multiple layers of transformations—can automatically learn feature hierarchies, eliminating the need for handcrafted features. This adaptability means that once a DL model is trained on representative data, it can generalise across a wide range of network conditions, even those not explicitly encountered during training [25], [26], [31].

Furthermore, DL approaches are supported by a well-developed ecosystem of tools and methodologies that facilitate their integration into real-world network systems. Additionally, modern frameworks enable distributed, accelerated training, allowing the processing of vast amounts of data generated by network logs and performance metrics. Advancements in transfer learning and model compression can help deploy these models efficiently on edge devices and network components, ensuring minimal latency overhead. Altogether, these capabilities position DL as a compelling choice for analysing,

classifying, and predicting scheduling algorithms in communications networks, leading to more robust and adaptive network performance.

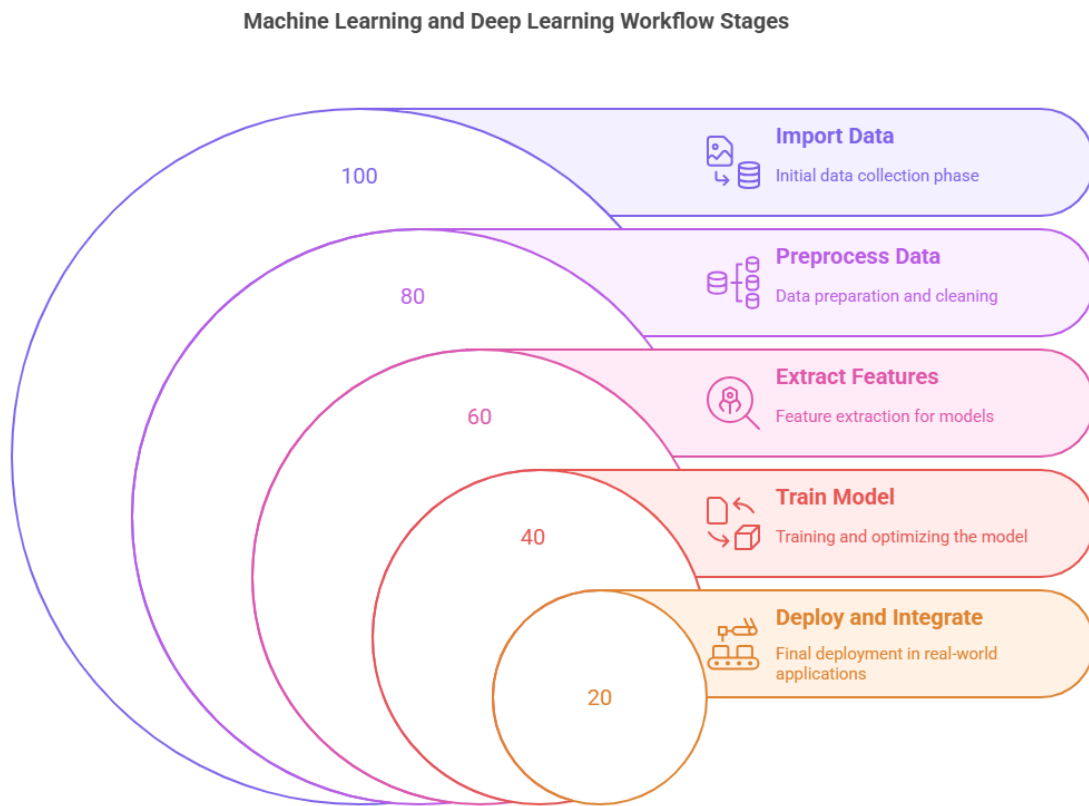


Figure 3.2: Design workflow for ML/DL Models.

Figure 3.2 illustrates the stages of ML and DL workflows. The initial stages include data collection, preprocessing, and feature extraction. Subsequently, the model is trained and optimised before its final deployment and integration into real-world applications [38], [39].

Deep Learning Workflow: From Data to Inference

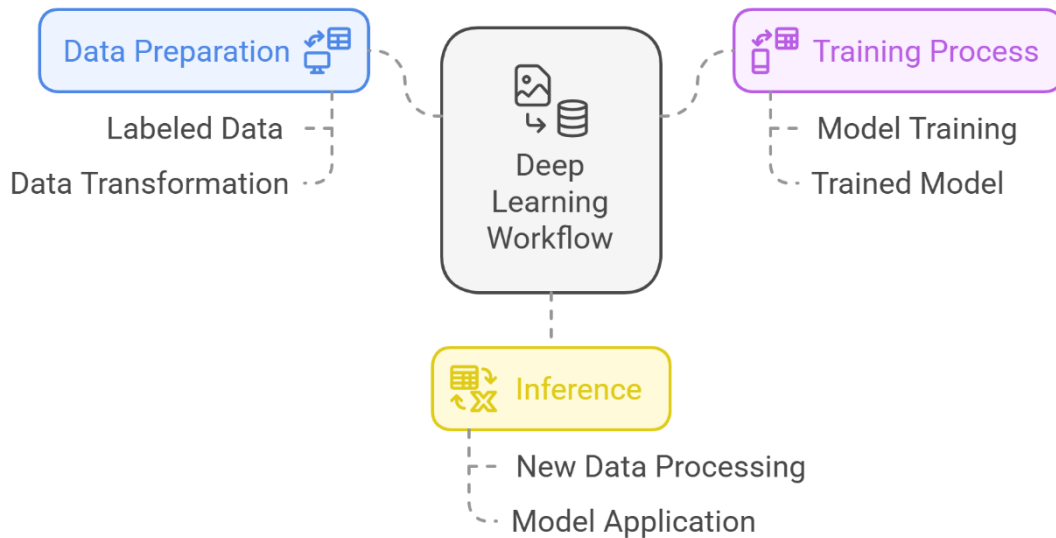


Figure 3.3: Design workflow to train neural network models.

Figure 3.3 provides a clear overview of the DL workflow, illustrating key stages from data preparation to model training, evaluation and application. It emphasises the iterative process of refining models and optimising performance, highlighting how raw data is transformed into actionable insights through neural networks [38], [39].

3.3.3 Validation and Evaluation Methods

In the development of AI models, ML, DL, and DRL, for network traffic and protocol analysis, classification, prediction, resource allocation, and prioritisation schemes, the adoption of rigorous validation methods is crucial. Validation methods, such as cross-validation, holdout, and bootstrapping, are foundational in ensuring that models accurately capture the underlying patterns and generalise well to unseen data. Validation techniques are crucial in mitigating overfitting, optimising model parameters, and ultimately selecting the model that offers the best blend of accuracy and generalisability. Considering the highly dynamic and complex nature of network environments, where traffic patterns and protocols continually evolve, employing diverse and robust validation methods is crucial for developing reliable models that can adapt to changing conditions and effectively manage network resources [40], [53].

After rigorous model validation, comprehensive evaluation methods are employed to assess their performance in real-world network settings. Evaluation metrics such as accuracy, precision, recall, F1 score, MAE, MSE, and the AUC-ROC provide detailed insights into the model's strengths and limitations. These metrics help understand how often the model is correct (accuracy), how precise its predictions are (precision), how well

it captures relevant instances (recall), and how it balances precision and recall (F1 score), among other aspects. In network traffic and protocol analysis, classification and prediction, these evaluation methods are indispensable for determining the model's effectiveness in identifying traffic patterns, predicting network conditions, and facilitating decision-making for optimal network management. By leveraging a comprehensive suite of evaluation metrics, developers and network administrators can ensure that the deployed models are technically sound and practically valuable in enhancing network performance and security [62], [63].

1) Validation methods

a) Holdout method: The holdout method is a straightforward validation technique commonly used to assess the performance of ML models, including those designed for network traffic and protocol classification and prediction. This approach splits the dataset into two sets: a training set and a testing set. The ML model is trained on the training set, where it learns to recognise patterns and make predictions, and then evaluated on the testing set to measure how well it generalises unseen data. This method helps assess a model's effectiveness and robustness in network environments, where it is essential to ensure that predictive models perform well not only on historical data but also on future data and potentially novel traffic patterns. The simplicity of the holdout method makes it an attractive first step in the validation process, providing an initial indication of model performance before more complex validation techniques are employed [64], [67].

b) Cross-Validation: Cross-validation is a robust validation technique widely used for evaluating the generalisation ability of ML models, including the models designed to classify or predict network traffic and protocols. It involves partitioning the data into multiple subsets and training the model multiple times, using each subset as the test set and the remaining subsets as the training sets. This process is repeated until each subset has been used as the testing set. The most common form of cross-validation is k-fold cross-validation, where the data is divided into k equally sized subsets. In addition, cross-validation enables a more comprehensive assessment of model performance across different data segments, thereby reducing bias and variance in the evaluation. Furthermore, this method is valuable in network settings, ensuring that models are effective on a specific portion of the data and robust and reliable across various network conditions and traffic patterns [71], [73].

c) Bootstrapping: Bootstrapping is a statistical resampling method commonly used for validating machine-learning models, including those aimed at network traffic and protocol classification and prediction. It involves repeatedly sampling, with replacement, from the original dataset to create numerous "bootstrap" samples. These samples are then used to train and evaluate the model, providing estimates of accuracy, variance and bias. This method is advantageous in scenarios where the dataset is limited or highly variable, as is common in network environments with dynamic, unpredictable traffic patterns. By leveraging Bootstrap, insights can be gained into how well a model will likely perform in real-world network conditions, ensuring that decisions based on its prediction are robust and well-founded [74], [92].

2) Evaluation metrics

a) Accuracy: Accuracy is a vital metric for evaluating the overall effectiveness of classification models, particularly those used for network traffic and protocol classification and prediction. This metric quantifies the proportion of correct predictions, encompassing both true positives and true negatives. This simplicity makes accuracy an intuitive measure of model performance, which is valuable in network scenarios where the straightforward assessment of how often the model correctly identifies or predicts traffic types and protocols is needed. Nevertheless, while accuracy provides a clear metric for model evaluation, its utility can be limited in imbalanced datasets where true negatives significantly outnumber true positives, potentially inflating the model's perceived effectiveness. Accuracy remains a critical starting point for understanding model performance in network management and security applications [94], [95].

b) Precision: Precision is vital for evaluating ML classification models. This metric is particularly relevant for network traffic and protocol classification and prediction tasks. It measures the proportion of true positive predictions relative to the total number of positive predictions made by the model, thus indicating its accuracy in identifying relevant instances amidst all instances it claims as relevant. In network management use cases, where false positives (such as incorrectly flagged traffic as malicious) can lead to unnecessary or disruptive actions, maintaining high precision is crucial. It ensures that the network's security and operational protocols are triggered only by genuinely concerning traffic, minimising the risk of mistakenly penalising or blocking legitimate activities. Precision plays a vital role in fine-tuning model performance to uphold network efficiency and integrity [96], [97].

c) Recall: Recall, also known as sensitivity or true positive rate, is an essential metric for evaluating machine-learning models for classification tasks, particularly in network traffic and protocol classification and prediction. This metric measures the proportion of actual positive cases the model correctly identifies, emphasising the model's effectiveness at capturing all relevant instances in the dataset. For example, in network security and management use cases, high recall is desired to ensure that no potential threats or significant events are missed, such as accurately detecting all instances of malicious or abnormal network traffic. A model with high recall minimises the risk of false negatives, thus enhancing the network's overall security posture and operational reliability by ensuring comprehensive monitoring and detection capabilities [94], [97].

d) F1 Score: The F1 Score is a vital metric used to evaluate the performance of classification models, particularly in situations where the data is imbalanced or when the cost of false positives and false negatives is significantly different. It provides a single metric combining precision and recall, offering a balanced view of model performance. This metric is handy in scenarios where precision and recall are crucial to the application. For instance, in a network security context, precision might accurately represent the model's capacity to identify real threats (minimising false alarms); on the other hand, recall represents the model's capacity to detect as many threats as possible (minimising

missed threats). An optimal balance between these two measures ensures that the model is accurate and comprehensive, making the F1-score an ideal evaluation metric [98], [99].

e) MAE: The MAE is an essential metric for evaluating regression models, including those used in network traffic and protocol prediction. This metric measures the average magnitude of errors between predicted and actual values, providing a straightforward, interpretable assessment of model accuracy regardless of direction (overestimation or underestimation). MAE is particularly useful in telecommunications networks where precise quantification of traffic volumes, latency, throughput and other continuous variables is essential for optimising resource allocation and scheduling decisions. Its importance lies in its ability to provide clear insights into the model's performance in practical, real-world tasks, helping network administrators and algorithms to minimise discrepancies in traffic predictions and thus enhance network efficiency and reliability [100], [101].

f) MSE: The MSE is an essential metric for assessing the performance of regression models in tasks such as network traffic and protocol classification and prediction. This metric calculates the average of the squares of the errors between the actual and predicted values, emphasising more significant errors by squaring them, making it particularly sensitive to outliers. This sensitivity is vital in network scenarios, where significant prediction errors can significantly affect resource allocation, scheduling decisions, and overall network performance. By prioritising the reduction of more significant errors, MSE helps ensure that models are accurate on average and minimises the risk of catastrophic prediction failures, making it an indispensable tool for developing robust network management systems [95], [108].

g) AUC-ROC: The ROC curve is an essential metric for evaluating the performance of classification models, including the ones applied to network traffic and protocol classification and prediction. These metrics measure the model's ability to discriminate among classes across all possible thresholds, providing a comprehensive indicator of performance independent of any specific classification threshold. This ability makes AUC-ROC particularly valuable in network scenarios where the balance between True Positive Rates (TPRs, sensitivity) and false positive rates (specificity) is crucial, such as detecting malicious traffic or anomalies. A high AUC-ROC score indicates that the model can effectively distinguish between different types of network traffic or accurately predict protocols, enabling more effective and reliable network management and security measures [108], [109].

h) RMSE: A widely used evaluation metric quantifies the average magnitude of the errors between predicted and actual values in a regression model. It is calculated by taking the square root of the average squared difference between predictions and true values. Squaring errors ensures that both positive and negative errors contribute to the overall error magnitude, and the square root returns the error to the original unit of measurement, making it more interpretable. A lower RMSE indicates a better-performing model, as it signifies that the model's predictions are, on average, closer to the true values. RMSE is

particularly sensitive to large errors due to the squaring operation, making it a helpful metric when large errors are undesirable [108], [109].

i) Success Rate: The success rate serves as a crucial and directly interpretable metric for evaluating the performance and reliability of systems, particularly in the context of scheduling and packet transmission. Quantifying the proportion of successful attempts relative to the total attempts indicates how consistently a system achieves its intended goals. A high success rate means a reliable system that effectively completes tasks or delivers accurate data, while a low success rate suggests potential issues and reduces dependability. Consequently, success rate is frequently employed as a primary performance measure, allowing for straightforward assessment and comparison of different system configurations or operational conditions and guiding efforts to enhance overall system effectiveness and robustness [90], [91].

3.3.4 Summary

Beyond the core model development, this methodology chapter underscores the critical importance of rigorous documentation, validation, and dissemination practices. This study established clear guidelines for documenting all aspects of the experimental setup, data collection, model training, and evaluation processes to ensure the reproducibility and reliability of our findings. This includes detailed descriptions of the simulation environment, the parameters used to collect synthetic and live network data, the specific ML, DL, and RL algorithms implemented, and the evaluation metrics employed. This meticulous documentation enables other researchers to replicate our work and build upon our findings.

Furthermore, the chapter emphasises the necessity for robust validation and dissemination strategies. To ensure the validity of our results, we advocate peer review and external validation processes, in which experts in the field scrutinise our methodologies and findings. This rigorous evaluation helps to identify potential limitations and strengthens the credibility of our research. Ultimately, we emphasise the importance of disseminating our research findings through publications in reputable academic journals. This ensures that our contributions reach a wider audience, fostering collaboration and advancing the state of the art in applying ML, DL, and RL to enhance QoS in 5G networks.

Chapter 4: Design and Implementation of Experimental Systems

4.1 Introduction

This chapter discusses a comprehensive series of experiments and implementations designed to address the aims and objectives established for this investigative study. The experiments outlined and discussed in this chapter form the core of this research study, leveraging various ML and DL techniques to enhance network traffic management and QoS prediction.

This research study's experimental approach is multifaceted, progressing from fundamental comparisons of scheduling algorithms to advanced DL models and RL techniques. This progression enabled this study to thoroughly explore the potential of AI to optimise 5G network performance.

The chapter is structured as follows:

First, this study conducted a comparative evaluation of scheduling algorithms (Proportional Fair and Round Robin) using statistical analyses and ML regression models. The dataset for these ML models was created from data captured in a live network environment, where PF and RR, implemented in Python, were deployed on hardware devices.

Subsequently, this study broadened its scope to additional real-world scenarios, focusing on classifying and predicting the performance of three scheduling algorithms: Priority (Prio), HTB, and CBQ. For this phase, ML and DL models were utilised, trained on comprehensive network traffic data captured from the operational network. This expansion to different types of real-world data and more complex tasks offers crucial insights into the practical applicability and versatility of our methods.

This study builds on these foundations to explore innovative approaches for representing network traffic data. This study designed and implemented a CNN model for image-based representation of network traffic datasets, followed by an LSTM model for the same purpose. These experiments aim to leverage the strengths of different neural network architectures to process and understand complex network traffic patterns.

Finally, this study pushes the boundaries of AI-driven network management by designing and implementing a Transformer-based RL agent for network traffic scheduling. This experiment represents a significant step towards autonomous and adaptive traffic management in 5G WSNs.

Throughout this chapter, this study will detail the methodologies employed for each experiment, including the design of the models, preparation of the dataset, and implementation of our algorithms. This chapter also describes the technical specifications of our experimental setup, the model parameters, and the evaluation metrics. This chapter focuses on the rigorous experimental procedures and implementations we have developed

to investigate AI-driven approaches for improving QoS in 5G networks. By providing a comprehensive account of our experimental framework, we lay the groundwork for the subsequent chapters, where we will present our results and conduct in-depth discussions of our findings.

Below is the end-to-end pipeline we use to study scheduling on network traffic datasets – from live capture to deployment. It standardises data collection and preprocessing, maps model choice to the task, and unifies training and tuning in MATLAB and Python, validation on unseen data, and evaluation with QoS-relevant metrics and visuals. We then interpret results against latency, throughput, jitter, loss targets and, where applicable, deploy models for real-time behaviour and online adaptation. Detailed MATLAB and Python workflows are provided in Appendix A. The pipeline includes the following steps:

1) Data collection

- Network traffic data captured using tools like Wireshark, tcpdump, and Python scripts.
- Traffic generated under different scheduling algorithms (PF, RR, PRIO, HTB, and CBQ).
- Data saved in PCAP format and converted to CSV for analysis.

2) Data Preprocessing

- Handle missing values and outliers.
- Normalise and standardise numerical features, encode categorical features, and create sequences for time-series models.
- Perform feature engineering (e.g., creating temporal features, aggregating metrics).
- Split data into training, validation, and test sets.

3) Model Selection

- Choose models based on task type:
 - ✓ **Classification:** Decision Tree, SVM, Random Forest, CNN, LSTM.
 - ✓ **Regression:** EGPR, Random Forest, LSTM, Hybrid Linear-LSTM-Transformer.
 - ✓ **RL:** Transformer-based RL agent.

4) Model Training

- Train ML models using MATLAB (Classification Learner, Regression Learner) or Python libraries (PyTorch, scikit-learn, Optuna).
- Train DL models using MATLAB's DL Designer or Python frameworks (PyTorch, TensorFlow).
- Apply hyperparameter tuning (e.g., grid search or Optuna), early stopping, and learning rate scheduling to DL models.

5) Model Validation and Testing

- Validate models using techniques such as k-fold cross-validation, the holdout method, or bootstrapping.
- Test models on unseen data to assess generalisation.
- Evaluate performance using metrics such as accuracy, precision, recall, F1-score, RMSE, MAE, R^2 , and AUC-ROC.
- Generate visualisations such as confusion matrices, ROC curves, precision-recall curves, and scatter plots.

6) Results Analysis and Interpretation

- Interpret model outputs and visualisations (e.g., feature importance, loss curves, bar charts).
- Compare scheduling algorithms and assess QoS metrics (delay, throughput, jitter, and packet loss).
- Discuss insights, strengths, limitations, and implications for 5G networks and emerging applications.

7) Deployment

- Deploy trained models in real-time network environments.
- Evaluate performance under dynamic conditions.
- Implement online learning for adaptive scheduling.

Table 4.1 contrasts our approach with prior models across key dimensions, including data sources, representation, learning objectives, real-time/latency constraints, tuning and evaluation protocols, and deployment and online adaptation. It highlights our central departures: live network datasets, multi-modal inputs (tabular, image, sequence), uncertainty KPI regression (EGPR), Transformer-based RL for prioritisation, and explicit latency-bounded inference.

Table 4.1: How Our Approaches Diverge From Existing Models.

Our Approach	Divergence from Existing Models
Comparative Evaluation of Scheduling Disciplines (PF, RR, PRIO, HTB, and CBQ) using ML/DL Classification/Regression Models	Diverges from traditional approaches that analyse scheduler performance primarily through direct observation or fundamental statistical analyses. Our approach explicitly uses ML classification to learn the unique “QoS” signatures produced by different scheduling algorithms, enabling them to be distinguished. This contrasts with model-based optimisation approaches or DRL solutions (like DEMUX [79] or TD3-OLLA [72]), whose goal is to replace or tune the scheduler parameters to find an optimal decision, rather than to classify or predict the behaviour of existing, complex queuing disciplines.
	Diverges by converting the network traffic monitoring and classification problem into an

Design and Implementation of a CNN Model for Image-based Representation of Network Traffic Dataset	image processing/computer vision task. DL and DRL-based resource allocation algorithms typically rely on processing structured numerical inputs, such as state vectors composed of metrics such as CQI, instantaneous throughput, delay, jitter, or channel conditions. Our method employs a CNN architecture designed for feature extraction from visual representations (scatter plots) of network traffic patterns (CBQ, HTB, PRIO, no_QoS).
Design and Implementation of a Hybrid CNN-LSTM Model for Image-based Network Traffic Analysis	This is an unconventional approach in which we applied an LSTM network to an image-based dataset of network traffic. This hybrid architecture first uses convolutional layers for spatial feature extraction, and then feeds the flattened output into an LSTM layer. This diverges from models such as DRLLA or PW-DRL, which use LSTMs or GRUs exclusively to capture temporal correlations and time dependencies within sequential numerical state features. In our approach, the LSTM is utilised to process features extracted from a spatially represented image dataset.
Hybrid Linear-LSTM-Transformer Model for Delay Prediction	Diverges from pure recurrent or simple DRL architectures by combining LSTM (for recurrent memory) and Transformer Encoder (for multi-head self-attention) in a sequence regression model. This stacked approach is designed to capture long-range or contextual relationships that an LSTM alone might miss. While DRL platforms like PW-DRL [15] use LSTMs in their prediction networks, integrating a self-attention mechanism into the core sequential prediction logic itself represents an advanced technique compared to the simpler GRU/FC layers often found in foundational DRL action/critic networks.
Transformer-based RL Agent for Network Traffic Scheduling	Pushes the boundaries for AI-driven network management by integrating the complex Transformer architecture into the core RL policy network. This approach leverages self-attention mechanisms to weigh the importance of elements within the structured network state sequence (including traffic demands, channel conditions and QoS requirements). This diverges significantly from many existing DRL methods for resource allocation, such as Deep Q-Learning (DQL), which is suitable only for discrete, small action spaces. DDPG or policy gradient-based actor-critic learning (PGACL), which are traditionally built on

	simpler fully connected networks or standard RNNs to handle continuous action domains.
Delay Prediction Models implemented using a Device-wise Split Strategy.	To ensure generalisation capabilities for new devices, the dataset was split using a group-wise shuffle split so that the entire subset of devices (~20%) was held out for validation/testing, ensuring no device appears in both the training and test sets. This strategy is crucial for assessing how well the model generalises to unseen devices, distinguishing it from simpler training splits in which data points from the same device might appear in both the training and test sets.
Implementation using Real-Time Live Network Data and Online Training	Our Transformer-based RL agent is trained online using real-time network data from a live operational environment. This is distinct from many simulation-based or offline training approaches discussed in the sources. Online training eliminates reliance on potentially inaccurate network models or simulations and enables the scheduler to continuously adapt to dynamic, non-stationary network conditions, such as fluctuating traffic loads and varying resource availability.

4.2 Comparison Evaluation of Scheduling Disciplines Using ML Regression Models

This experimental work aims to investigate the performance of the PF and the RR scheduling algorithms in a real-world WLAN environment. The core objective is to collect a comprehensive QoS dataset by actively probing four target machines (Raspberry Pi devices) from a central server. This dataset is intended for training and evaluating ML models to compare the referred scheduling disciplines. The corresponding experimental results and performance metrics are discussed in *Section 5.2*, and the interpretation of the findings and contributions to scheduling research are discussed in *Section 6.2*.

Furthermore, the experiment is implemented in Python, and a central scheduler program (PF or RR) orchestrates periodic network pings to each Raspberry Pi device over the WLAN. We utilised the *icmplib* library, a modern Python ICMP implementation, to send ICMP echo requests (ping) and measure responses. Each ping cycle, the scheduler selects one target device (based on the scheduling strategy), sends a ping packet (64-byte payload), and records the outcome. This includes the RTT as latency, any packet loss, and timing information needed to compute jitter and throughput. The code runs in a loop with a fixed interval (e.g., 0.5 s between scheduling decisions), simulating a time-slot scheduler. All devices are on the same Wi-Fi network (via a common access point), each with a static IP address. A logging module records the metrics for every ping in a CSV file. Crucially, the CSV has a well-defined structure (one row per device per timeslot)

with labelled columns for timestamp, device ID, metric, etc., making it immediately usable for data analysis or ML training. The entire setup thus mimics a multi-client WLAN environment, allowing us to enforce different scheduling algorithms and observe the QoS delivered to each client.

4.2.1 Scheduling Algorithms

1) PF Scheduling: Proportional Fair is a compromise-based scheduling algorithm that aims to balance total network throughput with fairness among users. In our implementation, each device maintains an exponentially weighted moving average of its throughput. For each scheduling interval, the PF scheduler computes a priority metric for each device as the ratio of the device's instantaneous achievable throughput to its average throughput. The device with the highest ratio is selected to ping next. Intuitively, this favours devices that have high current throughput relative to their past average, for instance, devices that are momentarily in good condition or that have been underserved recently. If a device has not been served yet, we initialise its PF metric to a high value to ensure that new or previously idle devices get a chance. Over time, PF "rotates" through all devices but allocates more opportunities to those that can best utilise the channel at a given moment. This approach leverages time-varying channel quality while ensuring each device receives a minimum level of service. The PF scheduler tends to improve overall throughput (by exploiting favourable link conditions) without starving any device.

2) RR Scheduling: Round Robin is a simple, cyclic scheduling algorithm where each device is served in turn, in a fixed order. The scheduler maintains a circular list of devices and selects the next device in the list each cycle, wrapping around to the start after reaching the last device. This ensures each node gets an equal share of ping opportunities in strict rotation. RR scheduling is straightforward to implement and inherently fair in the short term – every device is served regardless of its channel condition. There are no priority differences; no device can monopolise the channel. Nevertheless, RR does not account for differences in instantaneous channel quality or throughput potential. A device experiencing poor wireless conditions is still assigned the same time slot as others, which can reduce overall network throughput if that slot is less productive. Despite this, RR guarantees starvation-free access (every client gets regular service) and is often used as a baseline scheduler due to its simplicity and fairness.

In summary, PF scheduling dynamically prioritises devices based on real-time performance relative to their history (seeking a balance between efficiency and fairness), whereas RR scheduling treats all devices uniformly in a fixed cycle. Our experiment implements both scheduling schemes under identical conditions to collect comparative data on their performance impact.

4.2.2 Experimental Configuration and Parameters

To ensure a controlled and fair comparison, this study employed the same WLAN setup for both the PF and RR trials, varying only the scheduling logic. Table 4.2 summarises the key experimental parameters:

Table 4.2: Experimental Configuration for Scheduling Trials.

Parameter	Description/Value
Client Devices	4 x Raspberry Pi (wireless clients)
Device IPs	192.168.1.140, 192.168.1.141, 192.168.1.142, 192.168.1.143
Wi-Fi Network	802.11 WLAN via AP (2.4 GHz band)
Scheduler Host	1 x PC (laptop) running Python scheduler
Scheduling Interval	0.5 s between scheduling decisions (ping cycles)
Experiment Duration	7200 s (2 hours) per scheduler run
Ping Packet Size	64 bytes ICMP payload
Ping Timeout	2s (declares packet lost if no reply)
Scheduling Algorithms	Proportional Fair (PF) and Round Robin (RR) run in separate trials

All Raspberry Pi devices were connected to the same wireless access point, placed in the same physical area to minimise external variability. During a test run, the scheduler continuously looped through ping cycles for the specified duration (2 hours), either cycling through devices (RR case) or picking devices based on the PF metric (PF case). The 0.5 s transmission interval indicates that the scheduler attempts two pings per second, allocating roughly one ping timeslot to a device at a time. This interval is a trade-off between obtaining fine-grained temporal data and not overloading the network with measurement traffic. A 2-second ping timeout was set so that if a device fails to respond (due to delay or loss), the scheduler records a loss event for that ping and moves on. Each run (PF and RR) produces a timestamped CSV file (named with the scheduler type) containing the logged metrics.

4.2.3 Collected QoS Metrics

Throughout the experiment, we collected several QoS metrics to evaluate network performance under each scheduling discipline. These metrics, recorded for each ping and each device, are defined as follows (see Table 4.3).

Table 4.3: Collected Data Output for Experiment 2.

CSV Header	Description	Unit/Type
timestamp	ISO 8601 timestamp of measurement.	String
device_name	Name of the target device (e.g., “device_1”).	String

device_ip	IP address of the target device.	String
round_id	Identifier for the scheduling round.	Integer
delay_ms	Round-trip ping time for the selected device.	Milliseconds
jitter_ms	Jitter (stddev of recent delays) for the selected device.	Milliseconds
inter_arrival_s	Time since the last ping to this specific device.	Seconds
packets_sent	Cumulative packets sent to this device.	Integer
packets_received	Cumulative packets received from this device.	Integer
packets_lost	Cumulative packet loss for this device.	
loss_rate	Packet loss rate for the selected device	Float (0.0-1.0)
inst_throughput_bps	Instantaneous throughput for the device. (Calculated if selected and a successful ping is received, else 0.) Last known value if not selected in the current round.	Bits per second
avg_throughput_bps	Exponential moving average throughput for the device. (Updated if selected, else last known value).	Bits per second
pf_metric	Proportional Fair metric for the device. (Updated if selected, else last known value).	Float
is_selected	1 if the device was selected in this round, 0 otherwise.	Integer (0 or 1)

For the RR scheduling dataset, the columns will match those in Table 4.3, except for `pf_metric`, which is not part of the RR data collection.

This structured approach allowed us to collect robust datasets for PF and RR schedulers, enabling a comprehensive comparison using ML models.

4.2.4 Data Logging and ML-Ready Output for Enhanced Scheduler Comparison

A primary motivation for this experimental work, beyond immediate performance observation, is the generation of a high-quality dataset suitable for advanced ML analysis. This ML-driven approach aims to:

- 1) Corroborate and Augment Statistical Findings:** Verify if ML model outcomes align with traditional statistical analyses of scheduler performance.
- 2) Uncover Deeper Insights:** Utilise ML to identify complex patterns, dependencies, or conditions under which one scheduling discipline might outperform another, potentially revealing insights not obvious through direct observation or basic statistics.
- 3) Develop Predictive Capabilities:** Build models that can predict QoS metrics or even identify the active scheduling policy based on network behaviour.

To achieve this, the data logging strategy has been meticulously designed:

a) Comprehensive and Structured Data Points: For every scheduling interval (transmission_interval), a record is generated for each of the four target devices, regardless of whether it was selected for probing in that specific round. This ensures a continuous view of the perceived state (such as average throughput and PF metric) of all devices. The CSV output includes a rich set of fields, as detailed in Table 4.3 (e.g., timestamp, device_name, delay_ms, jitter_ms, loss_rate, inst_throughput_bps, and avg_throughput_bps).

b) Explicit Selection Indicator: The is_selected binary flag (1 or 0) is crucial. It unequivocally identifies which device was actively pinged in a given round. This allows ML models to directly associate the observed QoS metrics (such as delay_ms from a successful ping) with the scheduler's action for that specific device. For non-selected devices in a round, delay_ms and jitter_ms would be set to 0, but their historical avg_throughput_bps and current pf_metric (for PF) are still logged, providing valuable state information.

c) Scheduler-Specific Context:

- **Proportional Fair (PF):** The calculated PF_metric for each device is logged in every round. This provides direct visibility into the PF scheduler's decision-making input, allowing ML models to learn the relationship between this metric, the selection, and the resulting QoS.
- **Round Robin (RR):** While the RR selection index is implicit in the cycle, the combination of round_id and the is_selected flag for each device traces the sequential selection pattern, which ML algorithms can easily learn.

4) Suitability for ML Tasks: The collected dataset, with its temporal nature and per-device granularity, is well-suited for a variety of ML tasks:

a) Classification: The collected dataset is suitable to train a classifier to predict which scheduling algorithm (PF or RR) was active based on a sequence of QoS metrics from all devices. This tests if the scheduler produces distinguishable "QoS signatures". A

classifier can also be trained to predict if a device will experience high delay or packet loss in the next interval based on current and past metrics.

b) Regression: Our dataset allows us to develop models to estimate or forecast key QoS metrics such as delay or throughput for a specific device under a given scheduler. It also provides for creating a regression model to quantify the impact of selecting (or not selecting) a device on its evolving average throughput in bits per second (bps).

5) Data Volume and Granularity

a) A 2-hour experiment run with a transmission_interval of 0.5 seconds generates $(7200 \text{ seconds} / 0.5 \text{ seconds/round}) = 14,400$ rounds. Since data is logged for all four devices each round, this results in $14,400 * 4 = 57,600$ data rows per experiment.

b) This substantial volume provides sufficient data for training robust ML models, while the 0.5-second granularity captures relatively fine-grained network dynamics.

In summary, the experimental design and data logging methodology ensure the creation of a valuable asset for exploring scheduler performance through an ML lens, promising a deeper understanding beyond what is immediately observable. The structure directly supports feature engineering and model training to investigate the nuanced behaviours of PF and RR scheduling in a live WLAN environment.

Furthermore, by structuring the experiment and data as described, we ensure that results are not only directly interpretable but also reusable for advanced data-driven evaluation of QoS in WLAN scheduling contexts.

4.3 Classification/Prediction of Scheduling Algorithms Using ML/DL Models with Data Captured from Operational Network

4.3.1 Experimental Setup and Scenario Design

A specific experimental scenario was designed and implemented to collect network traffic data in a live network environment. This scenario aims to capture a wide range of network traffic patterns and behaviours, build datasets for analysing network traffic, and apply ML and DL models to classify, predict, and compare network-scheduling algorithms. Building on the experimental scenarios and QoS queuing implementations described in this Section, a classification-based performance analysis using ML models is presented in *Section 5.3*, focusing on PRIO, HTB, and CBQ schedulers. The experimental procedures for ML-based classification using operational traffic datasets presented in this Section form the basis for the synthesis and recommendations discussed in *Section 6.3*.

As illustrated in Figure 4.1, the setup includes a range of devices connected through a wireless modem to a network that comprises:

- 1) A Linux router configured with the tc (traffic control) command.
- 2) Multiple QoS configurations, including prio, HTB, and CBQ queuing disciplines.
- 3) A direct connection to the Internet via an ISP modem, enabling realistic testing of network traffic conditions.

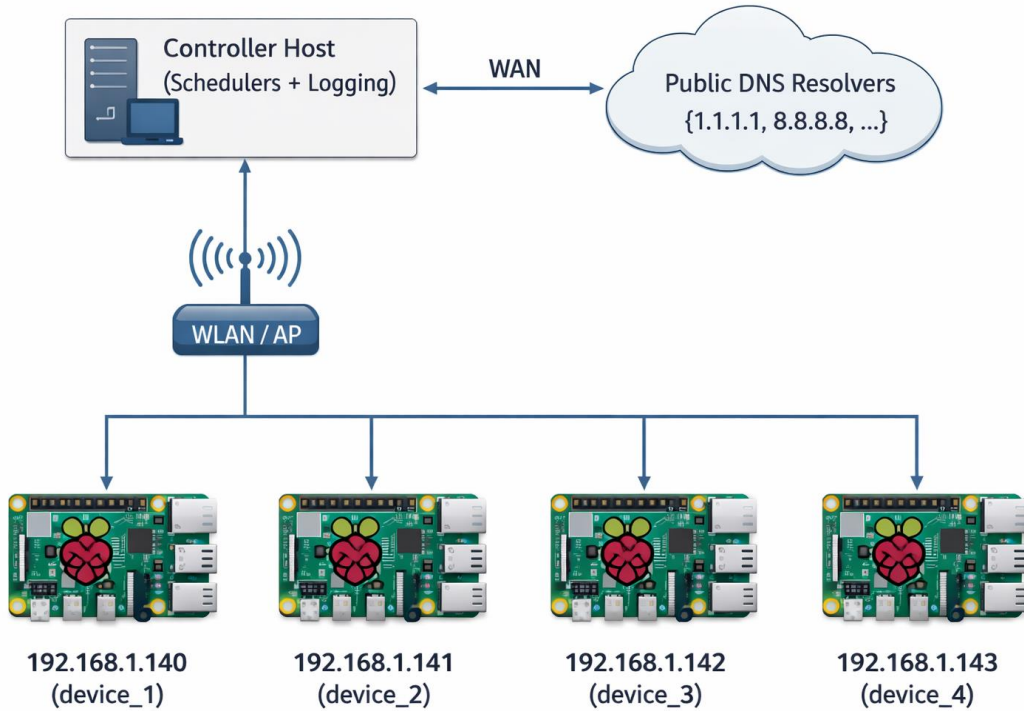


Figure 4.1: Illustration of the Experimental Network Scenario.

This setup simulates a typical WLAN-to-Internet connection scenario, allowing data from different devices to be captured after being queued according to priority classes defined by QoS policies.

4.3.2 QoS implementation using Prio Queuing Discipline

This study implemented a priority queuing discipline (PRIO) on a Linux-based system configured as a router to analyse traffic scheduling and prioritisation schemes. Figure 4.2 illustrates that the router interface has been configured with the prio queuing discipline. This qdisc organises packets into different classes based on priority levels. The bmon (bandwidth monitor) and Wireshark are utilised to capture real-time data flow statistics for analysis, and the collected data is used to create datasets for further study with ML and DL models.

This experimental setup involved:

1) Network interface configuration: The network interface is configured with a priority queuing discipline (qdisc), enabling packets to be classified into separate queues based on priority levels.

2) Priority Queuing Discipline (qdisc 1: (prio)): The prio queuing discipline (qdisc) in Linux systems organises packets into multiple classes according to assigned priority levels. Three priority classes (1:1, 1:2, and 1:3) are established in this configuration. Class 1:1 has the highest priority and is usually reserved for latency-sensitive traffic to ensure minimal delay and jitter. Class 1:2 (medium priority) is assigned to web browsing and general application traffic. The class 1:3, which is assigned a low priority, is typically used for bulk downloads, backups, or large file transfers that can tolerate higher latency.

```

linux-router@pc1: ~
lo
Interfaces
>lo
qdisc none (noqueue)
enp0s3
qdisc none (fq_codel)
enp0s8
qdisc 1: (prio)
  cls none (u32)
  cls 8000: (u32)
  cls 8000:800 (u32)
  cls 8010: (u32)
  cls 8010:800 (u32)
  class 1:1 (prio)
  class 1:2 (prio)
  class 1:3 (prio)
  
```

	RX bps	pps	%	TX bps	pps
lo	0	0		0	0
enp0s3	235.83KiB	0		14.33KiB	88
enp0s8	10.69KiB	0		385.00KiB	198
qdisc 1: (prio)	0	0		235.77KiB	198
cls none (u32)	0	0		0	0
cls 8000: (u32)	0	0		0	0
cls 8000:800 (u32)	0	0		0	0
cls 8010: (u32)	0	0		0	0
cls 8010:800 (u32)	0	0		0	0
class 1:1 (prio)	0	0		156.58KiB	117
class 1:2 (prio)	0	0		79.19KiB	81
class 1:3 (prio)	0	0		0	0

Increase screen height to see graphical statistics
 Press d to enable detailed statistics
 Press i to enable additional information
 Thu May 4 10:31:36 2023 Press ? for help

Figure 4.2: PRIO queueing discipline configuration on a Linux router.

This setup allowed the collection of prioritised traffic data, which was logged for subsequent analysis to assess the effectiveness of prio in managing traffic under various network loads.

4.3.3 QoS Implementation using Hierarchical Token Bucket Queuing Discipline

The Hierarchical Token Bucket (HTB) is a traffic-shaping mechanism in Linux that controls bandwidth allocation on network interfaces. This mechanism defines a hierarchy of traffic classes, each with its bandwidth constraints. It also enables fine-grained control over how network resources are distributed across different traffic types. As illustrated in Figure 4.3, HTB has been configured on the router's interface, and this qdisc implements a three-tier traffic-shaping configuration that divides bandwidth among different traffic classes based on traffic priorities. The data collected in this experiment was used to create a dataset for analysing traffic patterns with ML and DL models under this scheduling discipline. Below is a detailed explanation of how the HTB qdisc is configured:

1) Root Class (qdisc 1: htb): The root class is the main entry point for the HTB queueing discipline, managing the overall bandwidth allocation. Additionally, it establishes the hierarchical structure for all child classes.

2) Default Class (class 1: htb): This parent class contains all child classes and enforces the total bandwidth limits. This class acts as a container for more specific traffic classifications.

3) Service Classes (1:10 and 1:20): These leaf classes handle actual traffic shaping for different traffic priorities. Each class can have its guaranteed bandwidth (rate) and ceiling (maximum burst rate).

4) CLS (Classifiers) entries: The cls entries (such as cls 8000, 8000:800, 8000:801) are filters that determine which package goes into which service class. Further, they use matching criteria to classify and direct traffic to the appropriate classes.

5) HTB Structure (1:, 1:10 and 1:20): This represents a three-tier hierarchy where network traffic hits the root qdisc, gets classified into the default class, and is distributed to appropriate service classes. This three-tier hierarchy enables fine-grained control over bandwidth allocation and traffic prioritisation.

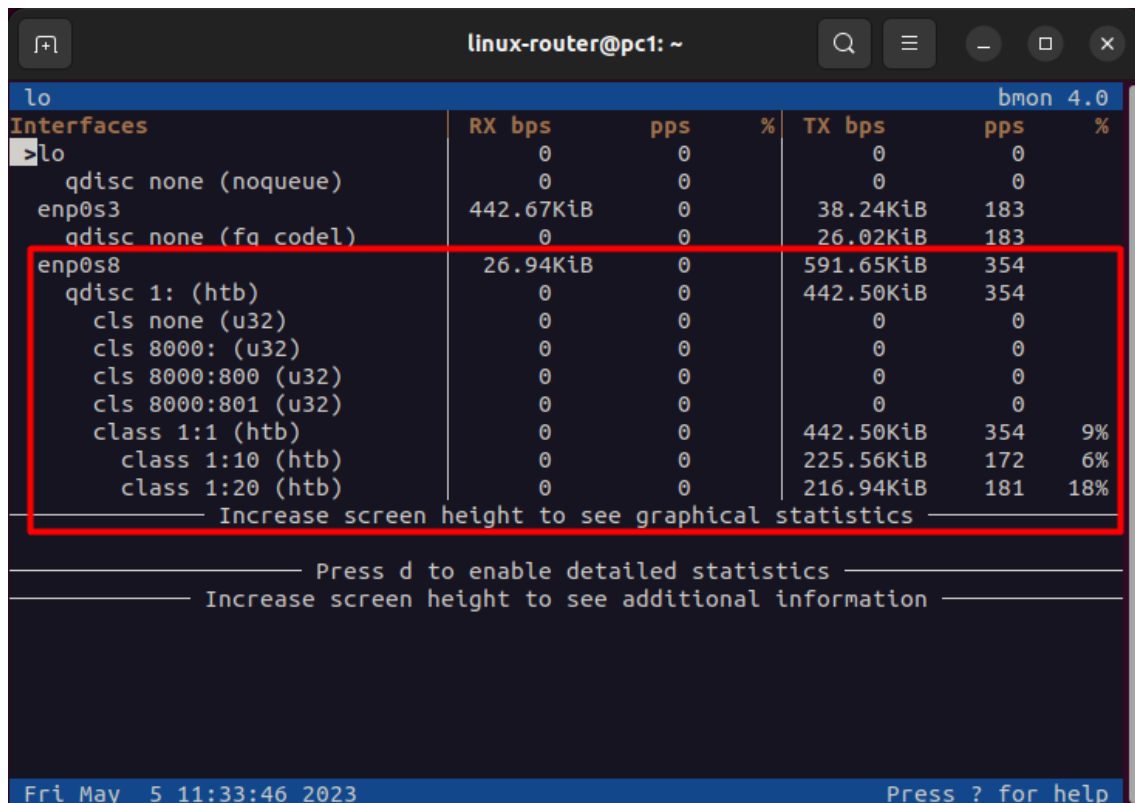


Figure 4.3: Illustration of the HTB queueing discipline configuration on a Linux router.

Figure 4.3 illustrates the router interface showing active bandwidth usage with transmission and reception rates. The TX (transmission) rate is quite significant at 591.65 KiB/s, with 354 packets per second, while the RX (receiving) rate is lower at 26.94 KiB/s; this indicates that, at the time the screenshot was taken, the interface was predominantly used for outbound traffic.

The HTB classes in Figure 4.3 demonstrate how the bandwidth is distributed across different traffic categories. Class 1:10 utilises 225.56 KiB/s, which represents 6% of its capacity, while class 1:20 uses 216.94 KiB/s, 18% of its capacity. The root class (class 1:htb) manages a total shaped traffic of 442.50 KiB/s, showing effective bandwidth management through the hierarchical structure.

All the CLS (classifier) entries, including cls 8000, cls 8000:800, and cls 8000:801, show zero traffic. It suggests that, although these classifiers are correctly configured in the system, they do not actively match any traffic at the time this screenshot was taken.

The HTB implementation demonstrates a functional traffic-shaping setup with effective bandwidth management, facilitated by its hierarchical structure. This experimental setup successfully captured network traffic patterns shaped by the HTB queueing mechanism.

4.3.4 QoS Implementation Using Hybrid HTB-PRIO Queueing Disciplines

This study employed a hybrid approach utilising HTB and PRIO queueing disciplines (qdiscs) in the experiment. The implementation follows a hierarchical structure, with qdisc 3 (prio) serving as the root qdisc and using PRIO scheduling. Under qdisc 3, there are multiple queueing disciplines organised hierarchically. The qdisc 1 implements HTB, divided into classes (cls) with identifiers such as 8000:0, 8000:800, and 8000:801. Within the HTB structure, nested classes (1:10, 1:20, 1:30) further organise traffic. The implementation also includes several PRIO qdiscs (qdisc 2, qdisc 3, and qdisc 4), each configured with three priority bands (shown as classes 2:1, 2:2, 2:3, 3:1, 3:2, 3:3, and 4:1, 4:2, 4:3, respectively). This hybrid implementation combines the traffic-shaping capabilities of HTB with the priority-based scheduling of PRIO, enabling both bandwidth control and hierarchical traffic prioritisation.

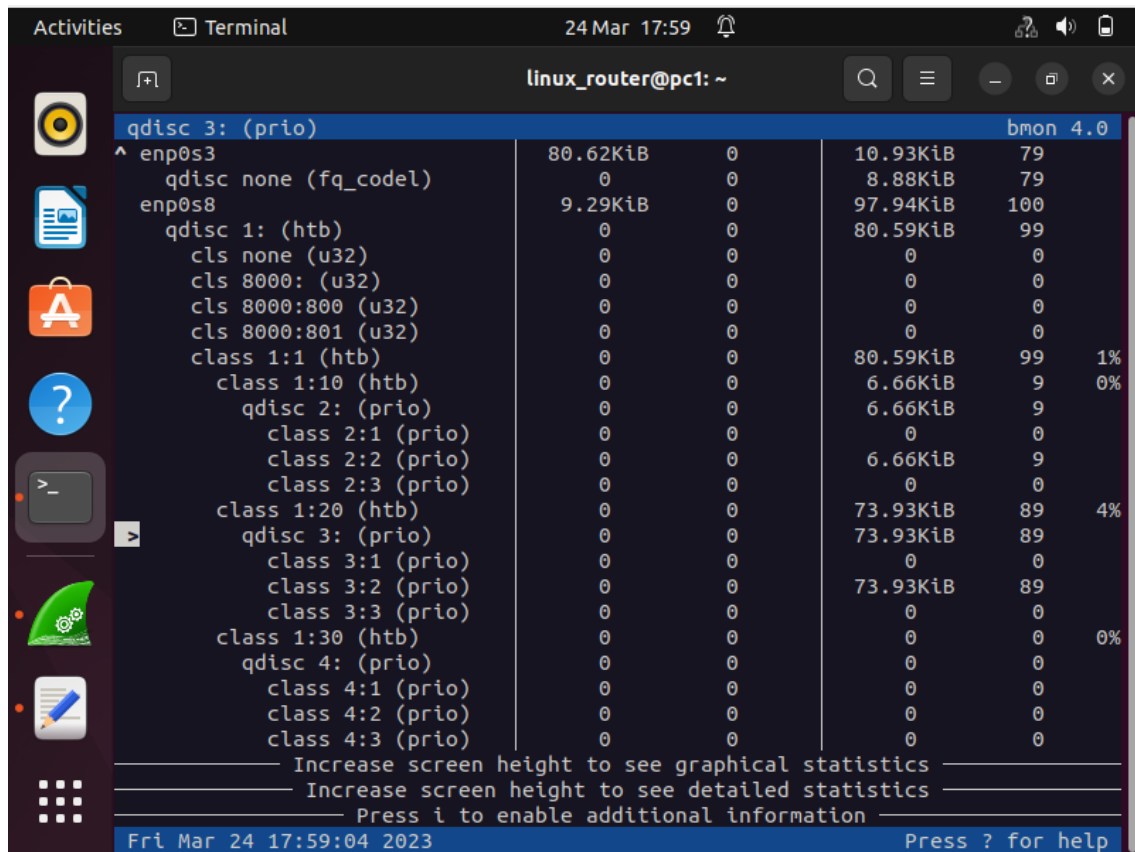


Figure 4.4: Illustration of a hybrid HTB-PRIO queueing discipline Implementation.

As shown in Figure 4.4, the traffic results for the QoS structure indicate that the highest activity is observed in class 1:1 (htb), with a transmission rate of 80.59 KiB/s.

Additionally, there is moderate traffic in class 1:20 at 73.93 KiB/s, while class 2:2 (prio) exhibits minimal activity at 6.66 KiB/s.

Furthermore, the root qdisc (qdisc 3 prio) shows a total bandwidth usage of 80.62 KiB, with one of the router's interfaces handling 9.29 KiB. Most other classes, particularly those under qdisc 4 and several PRIO classes, show no active traffic (0 KiB), suggesting that traffic was not flowing through this qdisc and classes at the time of the screenshot.

This hybrid implementation, utilising HTB and PRIO queueing disciplines, demonstrates a sophisticated approach to traffic management. By combining HTB's bandwidth control capabilities with PRIO's priority-based scheduling, the system can limit overall bandwidth usage and prioritise specific traffic types through the nested class structure. This hierarchical organisation enables granular control, allowing network administrators to allocate and shape bandwidth according to the needs of different traffic flows. The robust QoS solution's strength lies in its flexibility for managing both total capacity and the relative priorities of diverse traffic classes.

4.3.5 QoS Implementation Using Class-Based Queueing Discipline

Class-Based Queueing (CBQ) is a traffic control mechanism that divides network bandwidth into hierarchical classes. Each class can have specific bandwidth limits and priorities, allowing for controlled traffic shaping and prioritisation. CBQ helps manage network congestion by allocating bandwidth to various qdisc classes based on priority and traffic type, ensuring fair distribution and efficiency. The experimental data were used to create a dataset of traffic patterns under this queueing discipline for analysis with ML and DL models. Below is a detailed explanation of this experimental setup:

1) Interface and queueing discipline (qdisc): The Linux router is configured with the CBQ discipline.

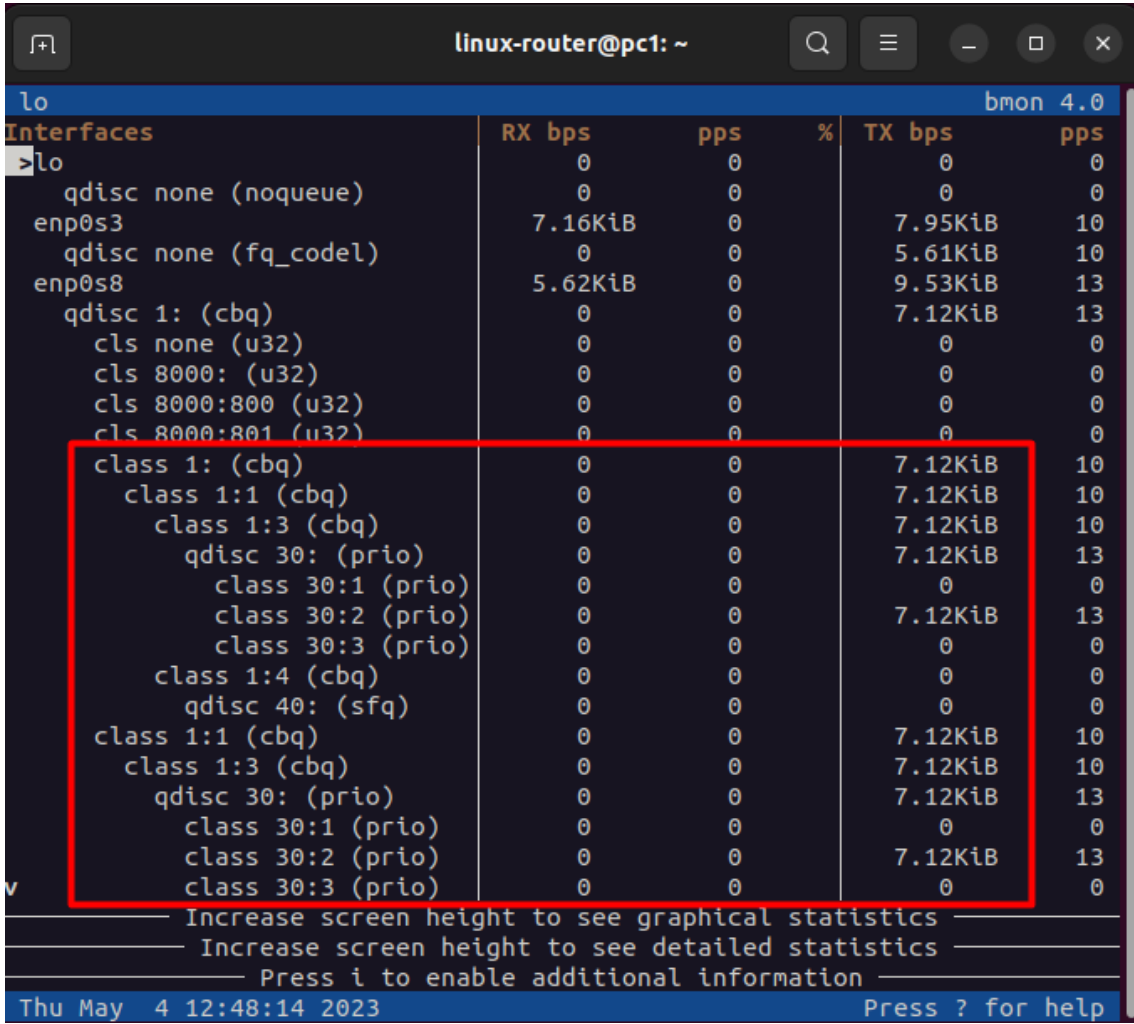
2) Class 1: cbq (main root class): This is the root CBQ class, defining the primary set of rules and bandwidth limits for all traffic this class manages.

3) Sub-Classes under Class 1 (class 1:1, 1:3, 1:4): These are child classes under the root, designed to handle different traffic flows, each with unique rate limits or priorities.

4) Prioritisation (qdisc prio) in Class 1:3: The qdisc 30 (prio) sets up three priority classes (class 30:1, class 30:2, and class 30:3), allowing for prioritisation within class 1:3 to ensure that certain traffic types are prioritised.

5) Stochastic Fair Queueing (qdisc sfq) in Class 1:4: The qdisc 40 (sfq), under class 1:4, uses Stochastic Fair Queueing (SFQ) to distribute bandwidth among flows, reducing congestion and ensuring fairness.

6) Hierarchical Structure: The nested structure of CBQ with priority queues allows hierarchical traffic shaping and prioritisation, with each child class inheriting and refining the rules of its parent class.



```

linux-router@pc1: ~
lo
bmon 4.0
Interfaces
>lo
  qdisc none (noqueue)
  enp0s3
  qdisc none (fq_codel)
  enp0s8
  qdisc 1: (cbq)
    cls none (u32)
    cls 8000: (u32)
    cls 8000:800 (u32)
    cls 8000:801 (u32)
    class 1: (cbq)
      class 1:1 (cbq)
        class 1:3 (cbq)
          qdisc 30: (prio)
            class 30:1 (prio)
            class 30:2 (prio)
            class 30:3 (prio)
          class 1:4 (cbq)
            qdisc 40: (sfq)
          class 1:1 (cbq)
          class 1:3 (cbq)
            qdisc 30: (prio)
              class 30:1 (prio)
              class 30:2 (prio)
              class 30:3 (prio)
  v
  Increase screen height to see graphical statistics
  Increase screen height to see detailed statistics
  Press i to enable additional information
Thu May  4 12:48:14 2023
Press ? for help

```

Interface	RX bps	pps	%	TX bps	pps
lo	0	0		0	0
enp0s3	7.16KiB	0		7.95KiB	10
enp0s8	5.62KiB	0		9.53KiB	13
qdisc 1: (cbq)	0	0		7.12KiB	13
cls none (u32)	0	0		0	0
cls 8000: (u32)	0	0		0	0
cls 8000:800 (u32)	0	0		0	0
cls 8000:801 (u32)	0	0		0	0
class 1: (cbq)	0	0		7.12KiB	10
class 1:1 (cbq)	0	0		7.12KiB	10
class 1:3 (cbq)	0	0		7.12KiB	10
qdisc 30: (prio)	0	0		7.12KiB	13
class 30:1 (prio)	0	0		0	0
class 30:2 (prio)	0	0		7.12KiB	13
class 30:3 (prio)	0	0		0	0
class 1:4 (cbq)	0	0		0	0
qdisc 40: (sfq)	0	0		0	0
class 1:1 (cbq)	0	0		7.12KiB	10
class 1:3 (cbq)	0	0		7.12KiB	10
qdisc 30: (prio)	0	0		7.12KiB	13
class 30:1 (prio)	0	0		0	0
class 30:2 (prio)	0	0		7.12KiB	13
class 30:3 (prio)	0	0		0	0

Figure 4.5: Illustration of experiments using the CBQ queue to gather data for analysis.

Figure 4.5 displays traffic data traversing the Linux router's interface, with transmitted (TX) and received (RX) data rates shown in kilobytes per second (KiB) and packets per second (pps). Each class (Class 1, Class 1:1, Class 1:3, and Class 1:4) shows a transmission rate of 7.12 KiB/s and a packet rate of 10-30 packets per second, while the receiving rate remains at zero across all classes, indicating no traffic at the time of the snapshot. This uniform transmission rate across all qdisc classes in the CBQ structure reflects consistent traffic shaping and bandwidth distribution, effectively managing traffic according to the configured hierarchy and ensuring stable prioritisation across different qdisc classes.

This experimental setup enabled detailed control over traffic, balancing prioritisation and fairness to manage network resources effectively, and gathered traffic data patterns shaped by CBQ queueing discipline for subsequent analysis with AI tools.

4.3.6 QoS Implementation using Class-Based Queueing (CBQ) and Hierarchical Queueing Disciplines

In this experimental setup, this study used Class-Based Queueing (CBQ) as the primary queueing discipline on the Linux router's interface (enp0s8), creating a hierarchical structure of traffic classes. Below is a description of this QoS scheme setup:

1) Setting up the root CBQ: The CBQ is configured as the root queueing discipline on the Linux router's interface. This queueing discipline is well-suited to hierarchical QoS, as it allows multiple classes with specific bandwidth constraints and priorities. Thus, this root CBQ is a top-level controller that manages various subclasses.

2) Creating traffic classes: Within the root CBQ, several traffic classes are defined (such as cls 8000 and cls 8010). Each of these classes corresponds to a specific traffic category or application that requires control, enabling bandwidth allocation and priority setting. These classes have unique identifiers that help organise and isolate traffic flows.

3) Subdividing classes with additional queueing disciplines: Additional queueing disciplines were configured within certain qdisc classes, which was done to add more control and optimise bandwidth usage, effectively subdividing them:

a) Priority FIFO (pfifo): A pfifo queueing discipline was configured in one subclass, which operates on a simple first-in-first-out basis. This approach is suitable for lower-priority traffic that does not require fair sharing but needs orderly handling.

b) Stochastic fair queueing (sfq): In another subclass, the SFQ ensures that bandwidth is fairly distributed among different flows. This scheme helps prevent a single flow from dominating bandwidth within a class and promotes fairness across applications or users.

4) Configuring FQ-CoDel on enp0s3: The fq_codel (Fair Queueing Controlled Delay) was implemented on the enp0s3 interface. This queueing discipline is designed to minimise latency by dynamically managing queue lengths, therefore mitigating buffer bloat. This queueing mechanism works well for controlling delay-sensitive traffic and complements the CBQ structure by improving responsiveness on another network interface.

5) Assigning traffic to classes: Each traffic flow was mapped to a specific class based on its requirements. For instance, real-time or high-priority traffic was directed to classes with higher priority or dedicated bandwidth, while less critical traffic was assigned to classes managed by pfifo or sfq. By carefully mapping traffic flows, this experiment ensured that priority traffic received adequate resources without compromising overall network performance.

6) Testing and adjusting parameters: This setup was tested under various network loads to verify that the bandwidth allocation and priority settings met the desired QoS objectives. This experiment enabled the collection of network traffic data for subsequent analysis using AI tools.

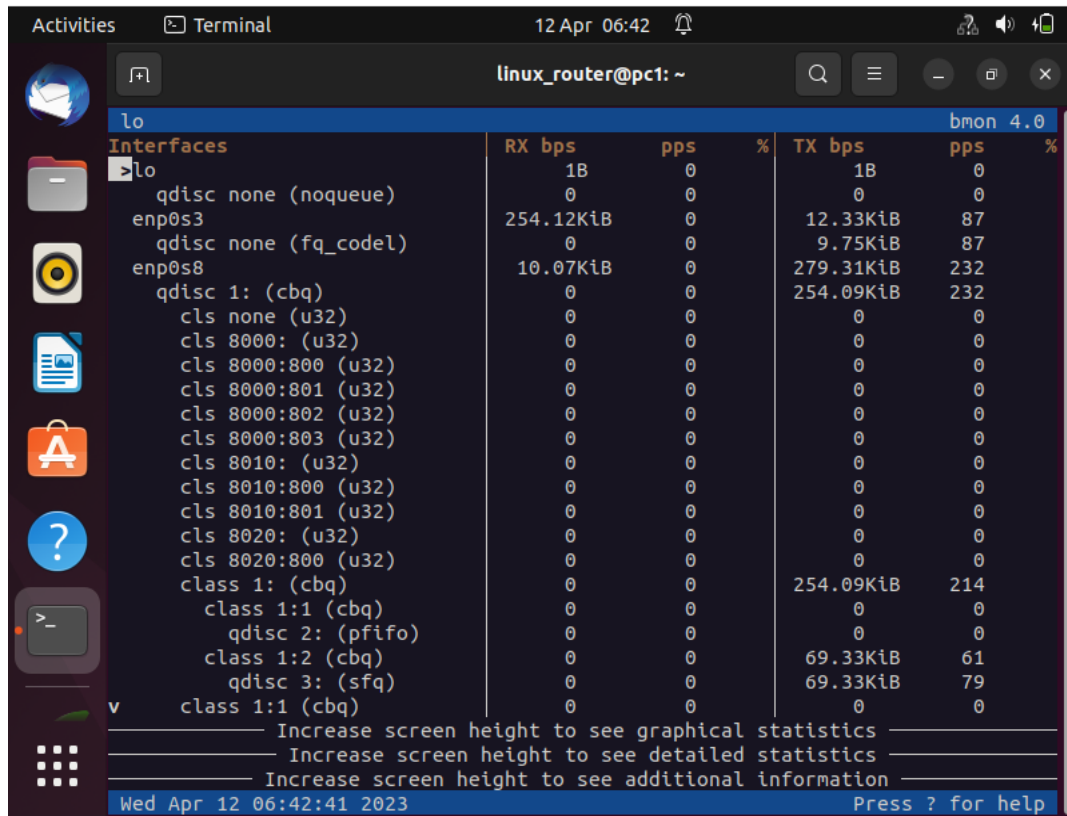


Figure 4.6: Illustration of CBQ configured with three classes (cbq, pfifo, and sfq).

Figure 4.6 shows significant data flows primarily on two interfaces, enp0s3 and enp0s8. The enp0s3 interface shows asymmetric traffic: it receives approximately 254.12 KiB at zero packets per second (pps) while transmitting 12.33 KiB at 87 pps. Similarly, enp0s8 shows distinct traffic characteristics, receiving 10.07 KiB at zero packets per second and transmitting 279.31 KiB at 232 packets per second. Within the qdisc and class hierarchy, notable traffic is concentrated in two specific classes: class 1:1 (cbq), handling 254.09 KiB at 214 packets per second, and class 1:2 (cbq), processing 69.33 KiB at 61 and 79 packets per second. All other configured qdiscs and classes in the system remain inactive, showing no traffic during the observed period or at the time the screenshot was taken.

This implementation leverages CBQ's hierarchical structure to enforce QoS policies, with pfifo and sfq providing additional control within classes and fq_codel reducing latency on a separate interface. This multi-layered approach enables granular control over traffic behaviour, ensuring that critical applications meet their QoS requirements while maintaining fair bandwidth allocation across all traffic types.

4.3.7 Real-time data collection

The bandwidth monitor (bmon) tool served as a crucial real-time network traffic visualisation utility in the experimental setups, providing comprehensive statistics about network interface throughput and packet flow patterns. As demonstrated in Figure 4.6, bmon offers a detailed hierarchical view of the traffic management system, displaying both RX (receive) and TX (transmit) statistics in terms of bytes per second (bps) and packets per second (pps) for each network interface and its associated queueing

disciplines (qdiscs) and traffic classes. The bmon's capability to present live traffic data in an organised, terminal-based interface makes it particularly valuable for monitoring and debugging traffic control mechanisms, allowing the observation of the immediate effects of traffic shaping policies and verifying the proper distribution of network flows across configured classes and qdiscs in real time.

In the experimental setup, Wireshark, a comprehensive network protocol analyser, provided detailed packet-level traffic inspection on the enp0s8 interface. As shown in Figure 4.7, Wireshark captures and displays essential packet information across multiple fields, such as packet number (N^o), timestamp (Time), source and destination IP addresses (notably the communication between 74.125.105.102 and 172.16.1.3), protocol type (predominantly TLSv1.2 and TCP), and packet length in bytes. The ability of Wireshark to capture and decode network traffic in real-time makes it invaluable for analysing protocol behaviour, verifying network configurations, and troubleshooting connectivity issues. Additionally, the hexadecimal packet dump displayed at the bottom of the interface enables a deeper inspection of packet contents, providing researchers with detailed information at the byte level for thorough protocol analysis and verification of traffic patterns.

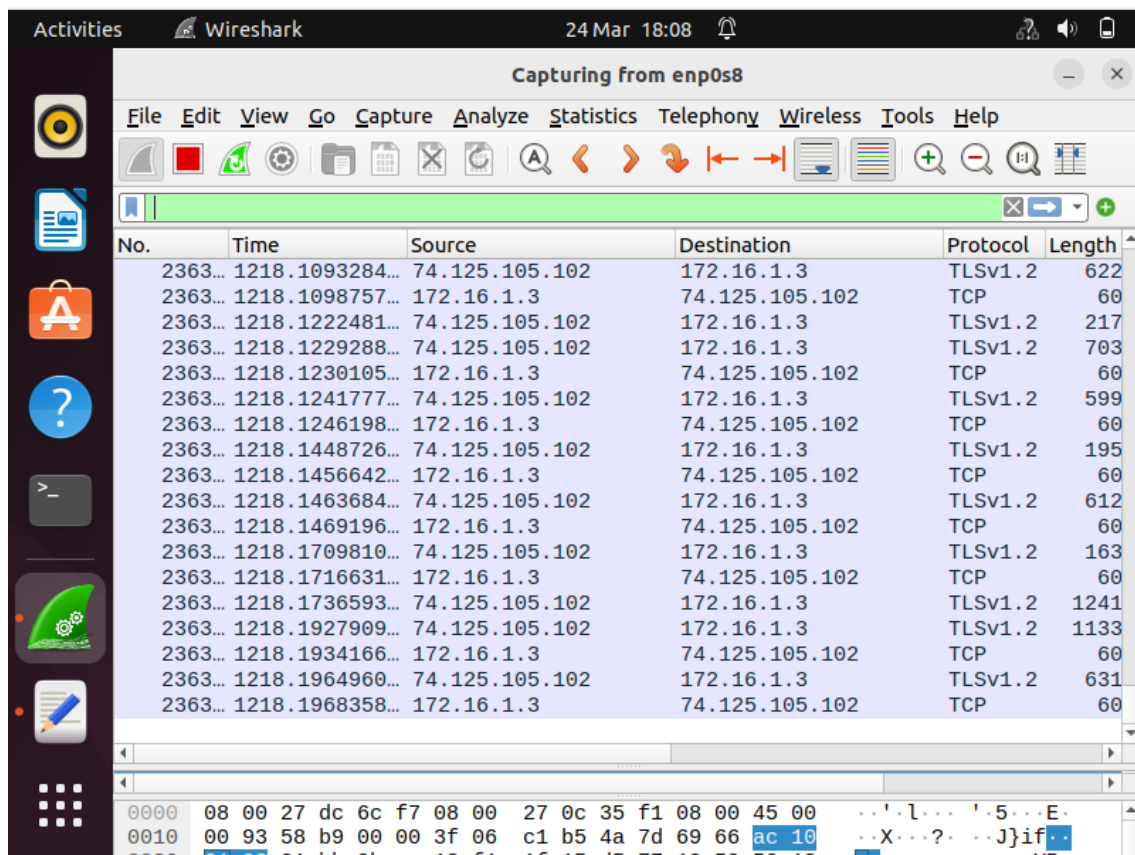


Figure 4.7: Illustration of Wireshark Traffic Capturing From Linux Router Interface.

4.4 Design and Implementation of ML Algorithms for Traffic Scheduling Analyses

This section discusses the design and implementation of ML algorithms to analyse the network traffic data gathered through the experiments described in Section 4.3. This section outlines the implementation steps for ML algorithms, including Decision Trees, SVMs, Ensembles, and EGPR. The design and implementation of ML models for a QoS dataset analysis involves a systematic approach, as presented in Table 4.4. The ML models implemented in *Sections 4.4.1 and 4.4.2* are further assessed in Section 5.3.4, where their classification performance is compared across datasets.

Table 4.4: Framework for Implementing ML Models for QoS Dataset Analysis.

Phase	Category	Metric/Task	Description
Experimental Setup	Network/QoS Configuration	Topology details, Traffic Generation, QoS parameters	Define the network environment and QoS configuration.
Data Collection and Preprocessing	Data Capture	Wireshark is used to collect network traffic data under different QoS schemes.	This process ensures the collection of high-quality data for model training.
	Feature Extraction	Extract features such as packet size, inter-arrival times, and flow duration.	Provides relevant data for model training
	Data Preprocessing	Handles missing data, normalises or scales features, and addresses class imbalance.	Prepares data for practical model training and prevents bias.
Feature Engineering	Feature Selection	Exploratory data analysis to identify relevant features.	Improves model performance by focusing on essential features.
Model Selection and Tuning	Model Choice (Decision Trees, SVM, Random Forest)	Choose models based on their strengths for the dataset.	Model selection and tuning leverage the strengths of different models for better performance.
	Hyperparameter Tuning	Use grid search or random search for tuning.	Optimises model performance and prevents overfitting.
Model Evaluation	Performance Metrics	Evaluate models using accuracy, precision, recall,	Model evaluation provides a comprehensive understanding of model performance.

		F1-score, and AUC-ROC.	
	Confusion Matrices	Visualise model performance.	It helps in understanding misclassification and model accuracy.
	Cross-Validation	Use k-fold cross-validation for generalisation.	Ensures model reliability across different data subsets
Sensitivity Analysis	Parameter Variation	Analysis of how changes in QoS parameters affect model performance.	Sensitivity analysis assesses a model's robustness under varying conditions.
Fairness Evaluation	Fairness Metric	Measure the fairness of resource allocation among traffic flows.	Fairness evaluation assesses equity in resource distribution.
Data Visualisation	Visualisation Techniques	Use time series, histograms, and box plots to understand traffic patterns.	Data visualisation provides intuitive insights into data and model behaviour.
Results and Discussion	Real-World Implications	Discussion on practical network deployment based on findings.	Results and discussions facilitate the application of research in real networks.
	Potential Pitfalls and Limitations	Ensure the dataset represents a diverse range of network conditions and addresses potential biases.	Enhance the reliability and generalisability of results.

Conceptual Framework

The core idea of this framework is to treat the network traffic generated by each scheduling algorithm as a unique signature. ML classifiers can then be used to learn these signatures and determine how accurately they can distinguish between them. This approach can provide us with insights into each algorithm's distinct behaviour and, potentially, which one is most effective for specific traffic types or applications. Table 4.5 summarises the key ideas of this conceptual framework.

Table 4.5: Conceptual Framework for Comparing Scheduling Algorithms Using ML Models.

Category	Metric	Description	Interpretation in Scheduling Context	Potential Insights
Classification Performance	Accuracy	The proportion of correctly classified instances among all instances.	It helps us analyse how well the ML model can differentiate between the traffic patterns produced by PRIO, HTB, and CBQ.	Higher accuracy suggests that the algorithms create distinct traffic patterns, making it easier to identify and predict their behaviour.
	Precision	The proportion of correctly predicted positive instances among all predicted positive instances.	Precision helps us understand how accurately the model predicts a specific scheduling algorithm (e.g., when predicting HTB, how often it is actually HTB).	High precision indicates a low false positive rate for a specific algorithm, meaning the model is reliable when it identifies that algorithm.
	Recall (Sensitivity)	Proportion of correctly predicted positive instances out of all actual positive instances.	Recall helps us understand how well the model can identify all instances of a specific scheduling algorithm.	High recall indicates a low false negative rate, ensuring the model captures most instances of a specific algorithm.
	F1-Score	F1-score is the harmonic mean of precision and recall, providing a balanced measure.	It is a general measure of the model's ability to correctly identify each scheduling algorithm.	A higher F1-score suggests a better balance between precision and recall, indicating overall good

				performance in identifying each algorithm.
	ROC Curve(AUC)	Plot the True positive rate (recall) against the False positive rate at various thresholds.	ROC curve visualises the trade-off between sensitivity and specificity (or 1-specificity). AUC (Area Under the Curve) provides an overall performance metric.	A larger AUC suggests better discrimination by the model across various thresholds.
Latency and Jitter	Mean Latency	The average delay experienced by packets.	Each algorithm introduces an average delay.	Lower mean latency is generally desirable for real-time applications.
	Latency Variance/Std.Dev.	Spread of latency values around the mean.	It is the consistency of delay introduced by each algorithm.	Lower variance indicates a more predictable latency; this is important for applications sensitive to delay fluctuations.
	Jitter	Variation in packet delay.	A measure of latency fluctuations introduced by each algorithm.	Lower jitter is crucial for voice and video call applications requiring smooth data transmission.
Throughput and Loss	Mean Throughput	Average data rate achieved	The average data transfer rate facilitated	Higher throughput is desirable for

			by each algorithm.	efficient data transfer.
	Throughput Variability	Fluctuations in the data rate.	It refers to the stability of the data transfer rate provided by each algorithm.	Lower variability suggests more stable data transfer.
	Packet Loss Rate	It is the percentage of packets lost during transmission.	It is a measure of each algorithm's data transmission reliability.	A lower packet loss rate is crucial for maintaining data integrity and ensuring optimal application performance.
Resource Utilisation	CPU Usage	Percentage of CPU resources consumed by each algorithm.	It measures each algorithm's efficiency in terms of processing power usage.	Lower CPU usage implies better resource efficiency.
	Memory Usage	Amount of memory utilised by each algorithm.	It is the memory footprint of each algorithm.	Lower memory usage is desirable for efficient resource utilisation, especially in resource-constrained environments.
Fairness	Fairness Index	A measure of how equally resources are allocated among different traffic flows.	An analysis of how fairly each algorithm treats different flows competing for resources.	A higher fairness index indicates a more equitable allocation of resources among different traffic flows.

4.4.1 ML Classification Models

1) Generate training data: The experimental setup described in Section 4.3 enabled the generation of network traffic data patterns influenced by the specified QoS schemes. Table 4.3 describes the types of traffic captured with Wireshark and stored as PCAP files to create datasets. Different queueing disciplines were tested to gather diverse, representative data and improve QoS. The following scenarios were central to this study:

- No QoS settings configured
- PQ settings are configured on the router
- HTB settings are configured on the router
- CBQ settings are configured on the router

Table 4.6 presents six categories of network traffic in the datasets to be used with the ML algorithms for comparison, classification, or prediction of scheduling algorithms. These categories encompass Audio, which includes voice calls and streaming audio; Video, representing streaming content and video conferencing; Chat, covering instant messaging and real-time text communications; Email, electronic mail traffic; File transfer, representing various forms of data transmission between network nodes; and ICMP, which handles error reporting and network diagnostics through ping requests and replies. Each traffic type exhibits distinct patterns, bandwidth requirements, and temporal characteristics, making them suitable features for ML models to classify and predict network behaviour. ICMP traffic, in particular, provides valuable insights into network health and connectivity status, while the other categories represent different aspects of user-generated content and communications.

By analysing these diverse traffic categories using ML models, it is possible to classify and predict key QoS metrics. This analysis enables network administrators to optimise resource allocation, implement QoS measures, detect potential security anomalies, and maintain robust network performance through comprehensive traffic monitoring. The Decision Tree algorithm, in particular, excels at interpreting these network traffic patterns and establishing classification rules based on various performance indicators such as latency, packet loss, throughput, and jitter. This systematic approach to traffic analysis enhances network security, ensures efficient bandwidth utilisation, and improves service delivery across different traffic types.

Table 4.6: Network Traffic Data for Datasets

SI. No	Network Traffic
1	Audio
2	Video
3	Chat
4	Email
5	File transfer
6	ICMP

Table 4.7 presents six widely used network applications selected for traffic analysis and classification/prediction experiments using ML techniques. These applications represent diverse network usage patterns: Skype generates both real-time audio/video conferencing and instant messaging traffic; Netflix and YouTube represent high-bandwidth video streaming services with different quality adaptation mechanisms; Facebook produces mixed traffic patterns including text, images, and short video content; Spotify represents audio streaming with variable bit rate requirements; and Gmail generates email traffic with varying attachment sizes and access patterns. Each application exhibits unique network behaviour signatures, including distinct packet sizes, timing patterns, and bandwidth consumption characteristics.

Furthermore, these applications were specifically chosen for their widespread use and diverse traffic patterns, making them ideal candidates for training ML models to accurately classify and predict network traffic behaviour. Understanding the traffic patterns of these popular applications is vital for network administrators to implement adequate QoS policies and maintain optimal network performance.

Table 4.7: Applications Associated with the Datasets

SI No	Application
1	Skype
2	Netflix
3	YouTube
4	Facebook
5	Spotify
6	Gmail

2) Data collection and preparation: This process begins with the network traffic data collection and preprocessing to ensure it is clean and relevant for classification or regression. Removing outliers or noisy data points that might skew results is vital, as is converting non-numeric features to numerical values if necessary. Additionally, it is crucial to split the data into training and testing sets to validate the performance of the Decision Tree algorithm.

3) Feature selection: Analyses of what features of the network data (such as latency, throughput, or packet loss) are most relevant to predicting the scheduling algorithm or decisions. Identify and retain only the most critical features to reduce computational complexity and improve model accuracy. Correlation analysis or feature importance methods are best practices for finalising the feature set for the ML model.

4) Model initialisation and parameter selection: The ML classification model can be initialised in MATLAB's Classification Learner app, which offers a user-friendly interface for exploring different configurations. Principal Component Analysis (PCA)

reduces dimensionality by transforming the original features into smaller components that explain a significant portion of the total variance (e.g., 95%). This approach simplifies the feature space, reduces computational complexity, and helps mitigate overfitting. The application's hyperparameter optimisation features are used to fine-tune parameters such as maximum tree depth, split criterion, or minimum leaf size. This experimentation helps find an optimal balance between model complexity and performance, with MATLAB's interface providing immediate feedback on validation accuracy. Figure 4.8 illustrates the initialisation and parameter selection for the ML model in MATLAB's classification learner application.

As shown in Figure 4.8, the PRIO dataset is loaded into MATLAB's Classification Learner to initialise the classifier by selecting the predictor set, applying PCA for dimensionality reduction, and using the built-in hyperparameter optimisation tools. The interface summarises the training setup (e.g., cross-validation and response classes) and reports validation accuracy across multiple candidate models, enabling rapid comparison and selection of an appropriate configuration.

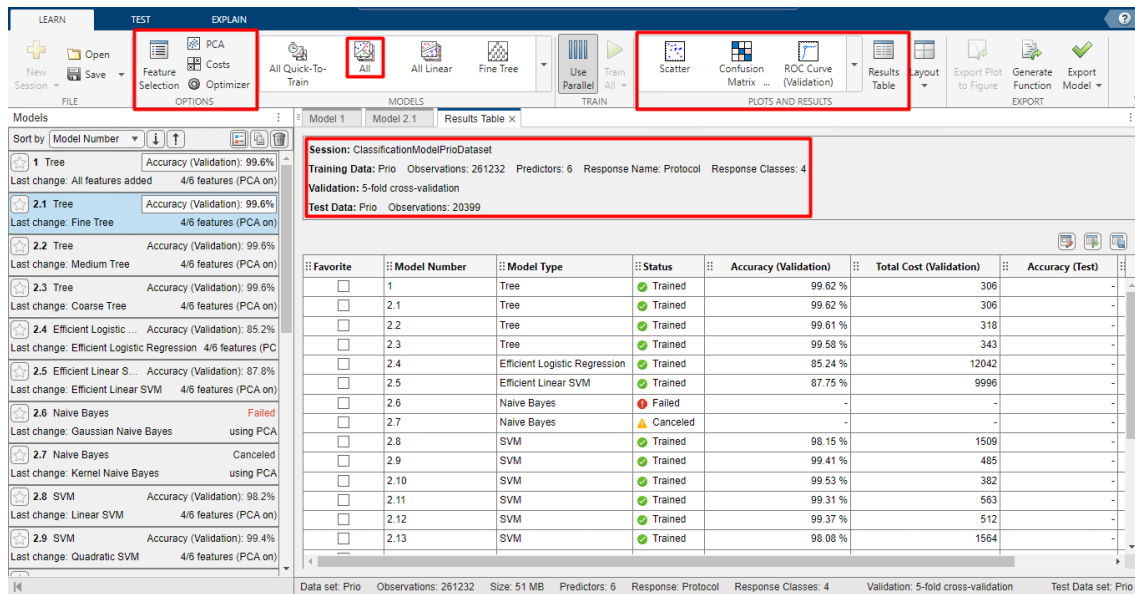


Figure 4.8: Illustration of ML Models Training with PRIO Dataset in MATLAB's Classification Learner App.

Table 4.8 provides a structural outline of the steps involved in implementing ML algorithms in MATLAB to classify network traffic data obtained from a live network capture. The ML workflow included applying various algorithms, such as Decision Trees, Logistic Regression, SVMs, and Ensemble Models.

Table 4.8: Implementation of ML models for network traffic classification in MATLAB.

Step	Description
Step 1: Dataset Preparation	- Collected network traffic data from live capture. - Dataset size for PRIO (73.9 MB)

	- Dataset size for HTB (66.0 MB) - Dataset size for CBQ (40.2 MB) - Six predictors were identified and classified into four response classes.
Step 2: Preprocessing	- Applied dimensionality reduction using Principal Component Analysis (PCA) to retain key features and reduce computational complexity - Ensured data was balanced and pre-processed for model compatibility.
Step 3: Model Selection	- Tested a variety of supervised ML models, including: - Decision Tree models (Fine, Medium, Coarse) - Logistic Regression (Efficient, Linear) - SVM - Naïve Bayes classifiers
Step 4: Training	- A five-fold cross-validation strategy was used to train the models and ensure generalisability. - Evaluated each model for accuracy and total validation cost.
Step 5: Performance Evaluation	- Performance evaluation measured validation accuracy and the total cost of each model. - “Fine Tree” is highlighted as the top-performing model with 99.6% validation accuracy and the lowest total cost (306). - Evaluated SVM and Logistic Regression models for comparison.
Step 6: Visualisation Metrics	- Generated plots for analysis, including: - Scatter plots to visualise data relationships. - Confusion matrices to assess classification accuracy across response classes. - ROC curves are used to analyse the trade-off between true-positive and false-positive rates.
Step 7: Testing	- Reserve the test dataset for the final evaluation.

	- Compare the test accuracy and other metrics with the validation results to ensure consistency.
Step 8: Result Analysis	- Determined the Fine Tree model to be the most effective based on accuracy and cost. - Documented the performance of other models, such as SVM and Logistic Regression, which performed comparatively lower.
Step 9: Reporting	- Reporting summarises findings regarding model performance, computational efficiency, and overall classification accuracy. - It also highlights areas for future optimisation and extensions of the implementation.

4.4.2 ML Regression Models for Live Network Traffic Data

This experiment compares the performance of scheduling algorithms such as PRIO, HTB, and CBQ by leveraging ML regression models. Network traffic data was collected from a live network using Wireshark to achieve the experiment's goal. The captured traffic, saved as PCAP files, was converted into CSV format for analysis. The dataset includes all fields encountered in the Wireshark capture, such as source and destination IP addresses, ports, packet lengths, Time-to-Live (TTL) values, protocols, and timestamps. This study experimented with different response variables (dependent variables) to predict. Table 4.9 describes the main components of a regression model:

Table 4.9: Components of a regression model.

Attribute	Description
Response variable	The response variable is the dependent variable that the model is trying to predict.
Predictor variables	These are the independent variables used to predict the response variable.
Coefficient/Parameter	The coefficient parameter is a numerical measure of the relationship between the predictor and response variables.

An error term	An error term relates to a measure of the deviation between the actual and predicted values.
Model fit statistics	Model fit statistics are measures of how well a model fits the data.

1) Data Preprocessing: This process involves handling missing or incomplete values, normalising numerical fields, and encoding categorical fields such as protocols into numerical formats. Careful feature selection is crucial to ensure that only the most relevant fields are included for model training, as irrelevant fields could introduce noise and reduce model accuracy. Splitting the dataset into training and validation subsets (e.g., 80% for training and 20% for validation) is recommended to ensure the model's performance can be reliably evaluated.

2) Model Selection and Training: The prepared dataset is imported using MATLAB's Regression Learner App, and the response and predictor variables are selected. The app enables the exploration of various ML regression models, including linear regression, Tree-Based Models (e.g., Fine Tree, Medium Tree), SVM, Ensemble Methods (e.g., Boosted or Bagged Trees), and Gaussian Process Regression. Additionally, this study trained multiple models to assess their suitability for the dataset. The initial training focuses on default models to establish a baseline for comparison.

3) Model Evaluation: Evaluating the performance of ML regression models is vital to understanding their accuracy and reliability. Key metrics, such as RMSE, R-squared (R^2 , also known as the Coefficient of Determination), and MAE, are used to assess the quality of predictions. The prediction plots in MATLAB's Regression Learner app can visually compare the actual and predicted values, helping identify patterns or inconsistencies. Additionally, residual plots can reveal systematic errors in the model's predictions.

4) Hyperparameter Tuning and Cross-Validation: After training baseline models, further optimisation is performed through hyperparameter tuning. For instance, adjusting the tree depth, experimenting with different optimisation algorithms, kernel functions for SVMs, or the number of estimators in ensemble models can significantly improve performance. This study ensured robust evaluation and minimised overfitting by employing K-fold cross-validation (e.g., 5-fold).

5) Model Comparison and Export: The best-performing models are identified based on metrics such as RMSE or R^2 after training and evaluating various models. The MATLAB Regression Learner app provides a Model comparison view that simplifies this process. The selected model can then be exported to the MATLAB workspace for further testing on new or unseen data. For instance, further experiments can be made using the exported model in MATLAB scripts, facilitating a deeper analysis of scheduling discipline performance.

6) Testing and Deployment: The exported model should be tested on a separate test dataset to ensure its generalisation capabilities. This step is crucial in live network environments where traffic patterns differ from those in the training data. If the model performs satisfactorily, it can be integrated into applications for real-time analysis or automated decision-making. By providing deployment options, such as generating standalone executables or integrating with larger systems, MATLAB enables the practical applications of the trained models.

7) Iterative Improvements: Iterative refinement is a vital step; based on the results, it is crucial to revisit earlier steps, such as data preprocessing, feature engineering, or model selection, to improve accuracy. This study also experimented with various response variables (delay, throughput, and packet loss) to gain deeper insights into the impact of scheduling disciplines on network performance.

4.5 Design and Implementation of a DL (CNN) Model for Image-based Representation of Network Traffic Dataset

Sections 4.5 to 4.7 detail the design and implementation of a CNN for image-based traffic classification, a hybrid CNN-LSTM model for spatiotemporal feature learning using the exact image-based representation of network traffic, and a standalone LSTM model for sequential input data. A comprehensive analysis of these models is presented in *Section 5.4*, and a synthesis of the findings, contributions and recommendations is presented in *Section 6.4*.

4.5.1 CNN-Based Network Traffic Classification

1) Data Preparation and Preprocessing: The network traffic data for this experiment was collected using the methodology previously described in sub-sections 4.3.1, 4.3.2, 4.3.3, and 4.3.5. For this experiment, an additional traffic class was introduced by capturing data from the same network scenario without any QoS scheme configured. Thus, the study utilised four datasets corresponding to different QoS configurations: CBQ, HTB, Prio, and no QoS configuration. The image dataset for each class was generated using MATLAB's plotting capabilities. Specifically, six distinct scatter plots were created for each class by plotting packet lengths against traffic features such as timestamps and source/destination IP addresses, resulting in 24 images.

These scatter plots were converted into standardised Red, Green, and Blue (RGB) images of size 227x227 pixels. This resolution was chosen to strike a balance between the need for detailed pattern representation and computational efficiency during training.

This investigative study applied the following augmentation technique to augment the limited dataset and enhance the model's generalisation capabilities:

a) Random rotation: Images were rotated randomly within a range of (+15 to -15) degrees to simulate different orientations.

b) Horizontal Flips: Images were flipped horizontally with a 50% probability to introduce mirror-image variations.

c) Random Translations: Images were shifted horizontally and vertically by 10 pixels to simulate positional variations.

These augmentation techniques expanded the dataset and introduced variability, helping the model learn more robust features than it could from the unaugmented dataset alone.

An example of the figures used for this experiment is presented in Figure 4.9. This Figure presents a scatter plot of the original PRIO dataset, showing Response (Length) versus Time. Most observations are densely concentrated at lower response lengths, forming clear horizontal bands that indicate repeated or quantised length values across the measurement period. A smaller number of points appear as sporadic high-length spikes (outliers) reaching the upper range of the plot, highlighting occasional bursts amid otherwise stable, low-to-moderate response lengths over time.

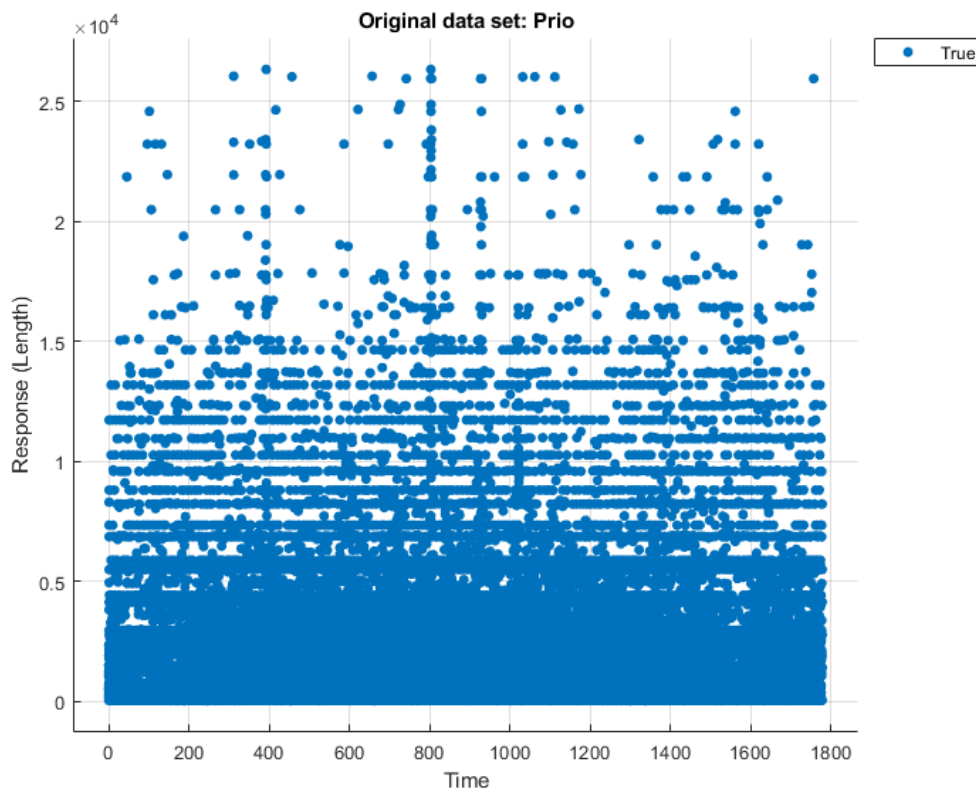


Figure 4.9: Extract of the Image Dataset for CNN DL Model.

2) CNN Architecture Design: The CNN model was implemented using MATLAB's DL Toolbox. It consists of an input layer, seven repeated feature extraction blocks, and final classification layers. Each layer's type, parameters, and output shape are specified in Table 4.10.

Table 4.10: Structure of CNN DL Model for Image Dataset of Network Traffic.

Layer Number	Layer Type	Parameters	Output Shape
1	Input Layer	Input Size [227 x 227 x3]	227 x 227 x 3
2	Convolutional Layer	Filters: 32, Kernel Size: 3x3, Padding: "Same"	227 x 227 x 32
3	Batch Normalisation	-	227 x 227 x 32
4	ReLU Activation	-	227 x 227 x 32
5	Max Pooling	Pool Size: 5x5, Stride: 1, Padding: "Same"	227 x 227 x 32
6	Convolutional Layer	Filters: 32, Kernel Size: 3x3, Padding: "Same"	227 x 227 x 32
7	Batch Normalisation	-	227 x 227 x 32
8	ReLU Activation	-	227 x 227 x 32
9	Max Pooling	Pool Size: 5x5, Stride: 1, Padding: "Same"	227 x 227 x 32
...	...	...	...
30	Average Pooling	Pool Size: 5x5, Stride: 1, Padding: "Same"	227 x 227 x 32
31	Fully Connected Layer	Number of Units: 10	1 x 1 x 10
32	Softmax Layer	-	1 x 1 x 10
33	Classification Layer	-	-

Figure 4.10 illustrates how all the components described in Table 4.10 are organised in the model.



Figure 4.10: The Proposed CNN Architecture.

3) Distribution of the Training Data: The distribution reflects a balanced allocation across the four classes, each contributing an equal number of observations (4). This balance ensures that the model does not become biased towards any class during training, which is crucial for fair performance across all categories. The dataset consists of 6 images per class (24 images in total); however, the dataset was splatted into separate subsets for training and validation, which is a common standard practice for DL to ensure the model is not only trained but also validated and tested on unseen data, which provides a better measure of its generalisation capability.

Despite the small size of the training set, the equal representation of each class is a strong positive aspect of the dataset's composition. Additionally, balanced training data helps reduce the likelihood of model bias, ensuring that the patterns and features of each class are equally represented during training. However, the limited dataset presented a challenge, as DL models typically require larger datasets to capture intricate patterns effectively and avoid overfitting. Data augmentation was applied to overcome this limitation, effectively expanding the training dataset and improving model generalisation.

This is evident in the training progress, where the validation accuracy reached a commendable 87.50%, and the validation loss stabilised, indicating effective learning.

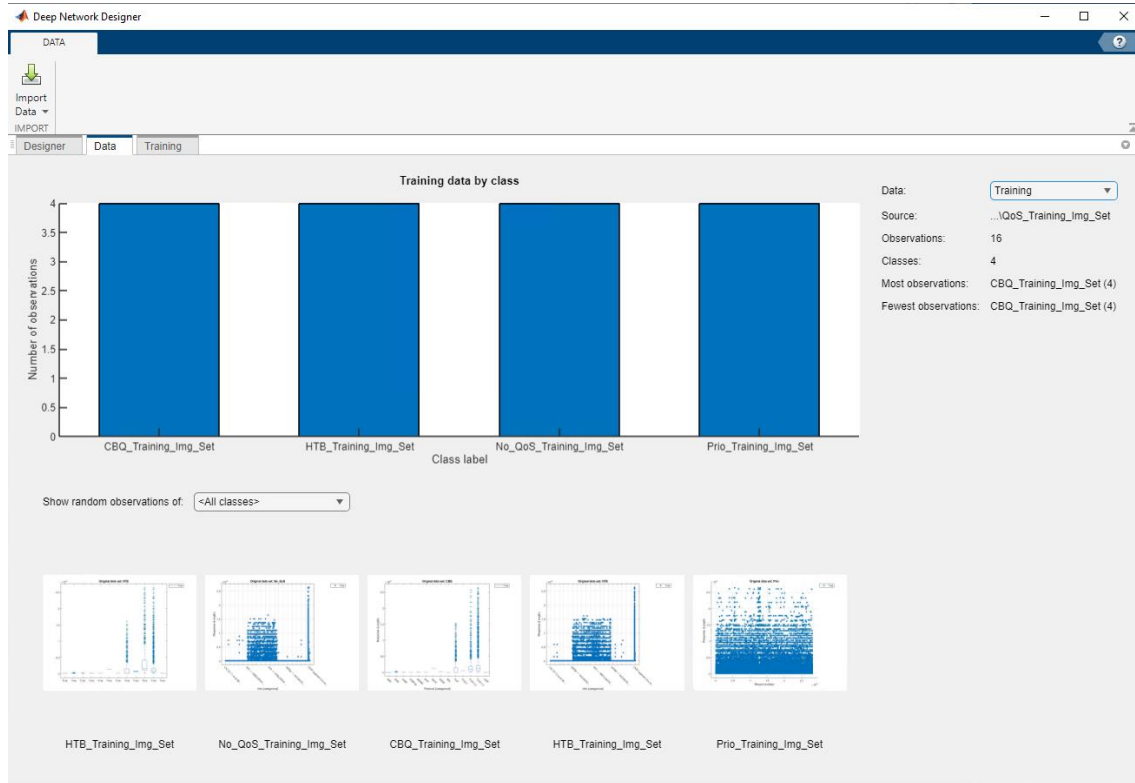


Figure 4.11: Distribution of the training data for DL Classification Models.

While the balanced distribution across classes (as presented in Figure 4.11) is ideal for training a CNN model, the small number of total observations in the training set needed careful handling. Data augmentation has mitigated the risks associated with a limited dataset, improving the model's robustness and generalisation.

4) Training Process Summary: Table 4.11 delineates the sequential phases of the CNN training process. It meticulously records key hyperparameters and performance metrics across distinct experimental phases, including the number of completed epochs, training duration, learning rate adjustments, validation frequency, and hardware resources. The Table provides a quantitative overview of the network's iterative refinement and facilitates analysis of training efficacy across varying configurations.

Table 4.11: CNN Training Process.

Experiment Phase	Epochs Completed	Training Time (s)	Learning Rate	Validation Frequency	Hardware Resource
Phase 1	30	141	0.01	50	Single CPU
Phase 2	50	9	0.001	50	Single CPU
Phase 3	80	15	0.001	50	Single CPU
Phase 4	100	18	0.001	50	Single CPU
Phase 5	200	32	0.001	50	Single CPU

Figure 4.12 presents a screenshot of the “Training Options” interface in the MATLAB DL Designer app, detailing the configuration used during Phase 5 of the CNN training process. The interface is systematically organised into distinct categories: “Solver”, “Basic”, “Sequence”, and “Advanced.” Key parameters displayed include the “Adam” solver, an initial learning rate of 0.001, a validation frequency of 50, a maximum epoch count of 200, and a minibatch size of 128. Figure 4.12 shows settings related to sequence handling, regularisation techniques, gradient threshold, validation patience, checkpoint, learning rate scheduling, and learning rate drop factor. This screenshot provides valuable insights into the hyperparameter tuning and optimisation strategies implemented during this training phase.

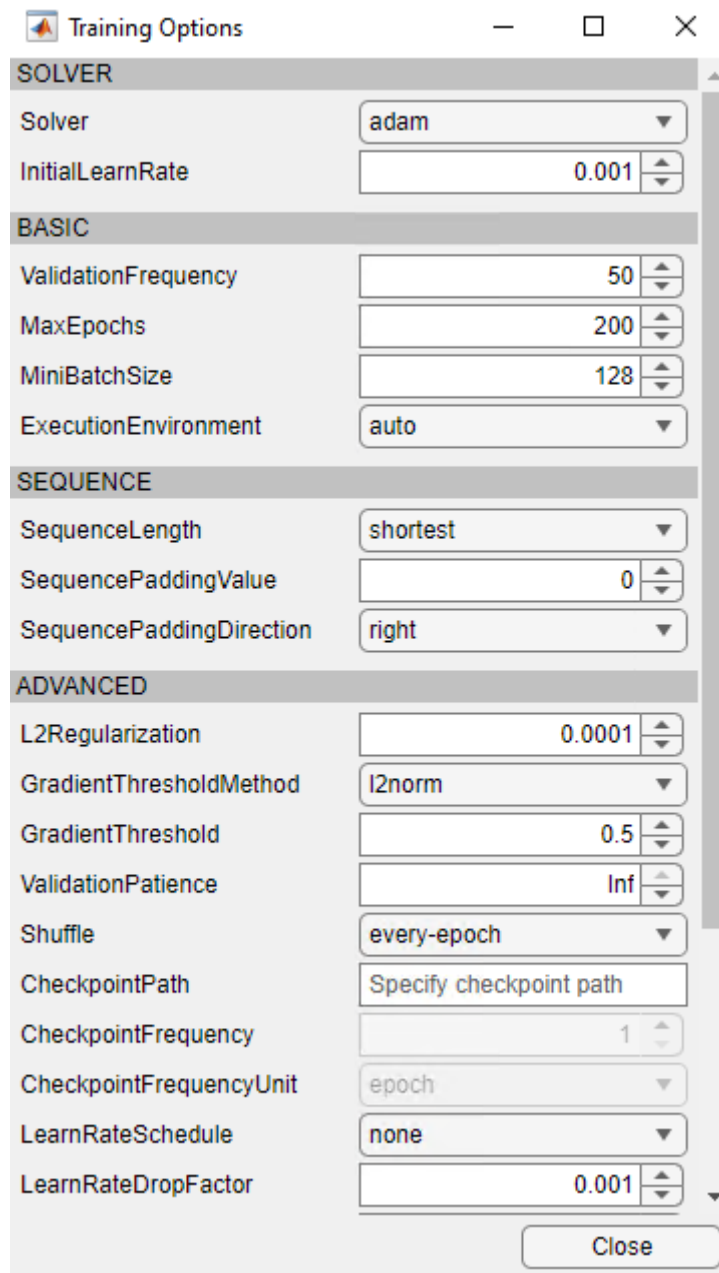


Figure 4.12: Illustration of Training Options Interface in the MATLAB DL App.

4.6 Design and Implementation of DL (CNN-LSTM) Model for Image-based Representation of Network Traffic Dataset

This Section discusses implementing an unconventional approach: applying an LSTM network to an image-based dataset of network traffic. The aim is to compare its performance with a CNN model, shedding light on the potential benefits and limitations of using LSTMs in this context.

4.6.1 CNN-LSTM Model for Network Traffic Classification

1) Data import and preparation: Implementing the LSTM model in MATLAB begins with importing the dataset using an image datastore, which reads images from a specific directory, including subfolders. Each subfolder represents a different class of network traffic (PRIO, HTB, and CBQ), and the labels are derived from the folder names. The dataset is split into training and validation sets at 70% and 30%, respectively, ensuring a robust evaluation setup.

2) Data augmentation: Data augmentation is applied to the training set to enhance the model's robustness and generalisability. This includes horizontal reflection and small translations, which artificially expand the dataset. The validation set remains unaltered to provide an unbiased performance measure.

3) Network architecture: The model architecture includes an image input layer for 28x28 pixel images with three colour channels. It features two convolutional layers, each followed by batch normalisation, ReLU activation, and max pooling, which are standard for extracting spatial features from images. A flattened layer converts these 2D features into a 1D vector, which is then processed by an LSTM layer with 200 units. The architecture concludes with fully connected layers, which are followed by a softmax classifier for multi-class classification.

4) Training options: The model is trained using the Adam optimiser, with a piecewise learning rate schedule that drops the rate by 0.001 every five epochs. The training process runs for up to 400 epochs, using a mini-batch size of 128. Data shuffling is performed every epoch to improve convergence.

```
%Import training and validation data.
imdsTrain =
imageDatastore("C:\Users\OrlandoNa\Desktop\QoS_Training_Img_Set","IncludeSub
folders",true,"LabelSource","foldernames");
[imdsTrain, imdsValidation] = splitEachLabel(imdsTrain,0.7);
% Define data augmentation for training and validation
trainAugmenter = imageDataAugmenter(...
    "RandXReflection",true,...
    "RandXTranslation",[-5 5],...
    "RandYTranslation",[-5 5]);
valAugmenter = imageDataAugmenter();
augimdsTrain = augmentedImageDatastore([28 28 3], imdsTrain, ...
    'DataAugmentation', trainAugmenter);
augimdsValidation = augmentedImageDatastore([28 28 3], imdsValidation, ...
```

```

    'DataAugmentation', valAugmenter);
% Set Training Options
opts = trainingOptions("adam",...
    "InitialLearnRate",0.001,...
    "LearnRateSchedule", "piecewise",...
    "LearnRateDropFactor",0.001,...
    "LearnRateDropPeriod",5,...
    "MaxEpochs",400,...
    "MiniBatchSize",128,...
    "Shuffle", "every-epoch",...
    "Plots", "training-progress",...
    "ValidationData",augimdsValidation);
layers = [
    imageInputLayer([28 28 3], "Name", "input")
    % Convolutional Layers
    convolution2dLayer(3,16, 'Padding', 'same', 'Name', 'conv1')
    batchNormalizationLayer('Name','batchnorm1')
    reluLayer('Name','relu1')
    maxPooling2dLayer(2, 'Stride',2, 'Name', 'maxpool1')
    convolution2dLayer(3,32, 'Padding', 'same', 'Name', 'conv2')
    batchNormalizationLayer('Name','batchnorm2')
    reluLayer('Name','relu2')
    maxPooling2dLayer(2, 'Stride',2, 'Name', 'maxpool2')
    % LSTM Layers
    flattenLayer("Name","flatten")
    lstmLayer(200,'OutputMode','last','Name','lstm1')
    % Fully Connected Layers
    fullyConnectedLayer(64,'Name','fc1')
    batchNormalizationLayer('Name','batchnorm3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(4,'Name','fc2')
    softmaxLayer('Name','softmax')
    classificationLayer('Name', 'classoutput')
];
% Train Network
% Train the network using the specified options and training data.
[net, traininfo] = trainNetwork(augimdsTrain, layers, opts);

```

4.7 Design and Implementation of LSTM Model for Scheduling Classification and Prediction

This study proposed a sequence-based scheduling classifier built on a two-layer LSTM that ingests mixed numeric and categorical features to distinguish among three scheduling algorithms (PRIO, HTB, and CBQ). To handle heterogeneous inputs, we first apply custom preprocessing: numeric features are mean-imputed and standardised, while categorical features are collapsed to N categories, imputed, and one-hot encoded. The LSTM is trained with class-weighted cross-entropy to mitigate label imbalance, and we monitor validation loss to trigger early stopping. After training, we evaluate performance

using accuracy, F1, ROC-AUC, and precision-recall metrics, and provide both quantitative reports and visualisations. This proposed approach aims to deliver a robust, interpretable classification of scheduling disciplines from tabular data by combining rigorous preprocessing, sequence modelling, and thorough evaluation.

1) Implementation Details

Table 4.12: Implementation Steps for LSTM-Based Model for Scheduling Algorithm Classification.

Step	Component	Description
Phase 1	Data Ingestion and Labelling	Load datasets (PRIO, HTB, and CBQ); verify if files exist; concatenate into one Dataframe; add a “scheduling” label and encode it via LabelEncoder.
Phase 2	Preprocessing Pipeline	Use a ColumnTransformer combining: (a) Numeric pipeline – mean imputation + StandardScaler; (b) Categorical pipeline – TopNCategoriesTransformer (N=10) → frequent-value imputation → OneHotEncoder.
Phase 3	Model Architecture	Define LSTMModel with input_dim equal to post-preprocessing feature count, two stacked LSTM layers (hidden_dim = 128, dropout = 0.5), and a final linear layer for three classes.
Phase 4	Training Procedure and Early Stopping	Train for up to 30 epochs using Adam (learning rate = 0.001) and weighted CrossEntropyLoss. Compute class weights from the training labels. Apply early stopping with patience (5), saving the best checkpoint.
Phase 5	Evaluation and Visualisation	On the held-out test set, compute accuracy, precision, recall, F1, and generate a confusion matrix, per-class ROC curves, and precision-recall curves (average precision).

In summary, the implementation presented in Table 4.12 demonstrates how to integrate custom preprocessing for mixed feature types with a lightweight LSTM-based classifier in PyTorch. By using class weighting and early stopping, our approach addresses both imbalance and overfitting. The comprehensive evaluation suite, encompassing standard

metrics, ROC, and precision-recall analyses, provides clear insights into model performance. Additionally, our approach provides a reproducible template for multiclass classification on tabular datasets using sequential modelling techniques.

4.8 Experimental Design and Implementation of Delay Prediction Models

This Section presents the experimental design and implementation of ML and DL models for predicting network delay using real-world traffic data. The objective is to evaluate and compare the performance of different model architectures in capturing delay dynamics under varying QoS conditions in 5G WSNs. Specifically, we explored a hybrid model combining Linear, LSTM, and Transformer components, a standalone LSTM model, and an RF baseline. The Section also describes the dataset used, the preprocessing steps, the tools employed, and the measures taken to ensure reproducibility. The model implementations introduced in this Section serve as the foundation for the comparative evaluation and results presented in *Section 5.5*, and are further analysed in *Section 6.5*, with emphasis on predictive accuracy and real-time applicability in URLLC scenarios.

4.8.1 Dataset and Preprocessing

This experimental study utilises a dataset collected from a live wireless network employing a Proportional Fair (PF) scheduling algorithm. The data consists of time-stamped records of quality-of-service (QoS) metrics for multiple devices, with each record labelled by the measured network delay in milliseconds (`delay_ms`). In addition, to prepare the data for modelling, several preprocessing steps were applied:

1) Data cleaning: Entries missing the target delay value were removed, and the remaining records were sorted by device and timestamp to preserve chronological order within each device's time series. Non-predictive fields, such as timestamps, device identifiers (including IP addresses and names), and experimental round IDs, were dropped from the feature set. The features retained include various QoS metrics (e.g., packet loss count, loss rate, instantaneous and average throughput in bits per second, and a scheduler selection flag). An internal PF scheduler metric (`pf_metric`) was recorded in the dataset but was excluded from the feature by default to assess model performance without direct scheduler information.

2) Leakage-safe splitting: To prevent information leakage, the dataset was split on a per-device basis. A group-wise shuffle split was performed, so that all data from a subset of devices (~20%) was held out for validation and testing, while the remaining devices' data was used for training. This device-wise split ensures that no device appears in both training and test sets, forcing the generalisation to unseen devices.

3) Feature engineering: A column transformer pipeline was applied to handle different data types. Continuous numeric features were imputed (median strategy for any remaining missing values) and then standardised through z-score normalisation. The categorical device identifier was ordinal-encoded as an integer, with a special code for unknown devices to accommodate new devices in the test set. All features were then

aggregated into a flat feature matrix for each record. For the two DL models, the pre-processed data were further organised into temporal sequences. Using a sliding window of length 10 times the number of steps, consecutive records for each device were grouped into input sequences (of length 10) with the delay value of the last time step as the prediction target. Overlapping windows were used (for a device with N records, this yields up to $N-9$ sequences) to maximise the number of training samples while maintaining chronological order. Notably, each sequence is drawn from a single device's timeline to avoid mixing device contexts. In contrast, for the RF baseline, no sequence assembly was performed – each record's feature vector (from a single timestamp) was used independently as a flat input, with no temporal context beyond what is implicitly captured by metrics like moving averages.

4.8.2 Implementation Tools and Reproducibility

All models were implemented in Python using well-established ML frameworks. The DL models were built with PyTorch (enabling custom neural network modules and GPU acceleration), while the RF used scikit-learn's implementation of an ensemble regressor. Data preprocessing utilised pandas for manipulation and scikit-learn pipelines for feature transformation (imputation, scaling, and encoding).

To ensure a fair model comparison and robust hyperparameter selection, we utilised Optuna for automated hyperparameter tuning. Each model's key hyperparameters were optimised by running multiple training and validation trials. For instance, the hybrid model's search space included the LSTM/Transformer hidden dimensions, number of transformer layers (1-2), number of attention heads, dropout rate, and learning rate, while the LSTM-only model explored different hidden sizes (e.g., 32, 64, 128, and 256) and number of LSTM layers (1-3). The Random Forest's search space covered the number of trees, maximum depth, and minimum samples per split/leaf. Optuna's trials were evaluated based on the validation MSE, and the best hyperparameter configuration for each model was selected for final training.

We paid special attention to reproducibility in the experimental setup. A fixed random seed (42) was used throughout the pipeline. This seed was applied to Python's random module, NumPy, and PyTorch to ensure consistent random initialisations. The data splitting (*GroupShuffleSplit*) and the RF also used the same fixed *random_state* to ensure consistent splits and model behaviour. Model training was performed on a single computing device at a time (using the CPU). We also implemented early stopping with a patience of 8 epochs – if the validation loss did not improve for 8 consecutive epochs, training was halted to prevent overfitting. The learning rate scheduler (*ReduceLROnPlateau*) was used to reduce the optimiser's learning rate when the validation loss plateaued, which helped stabilise and improve convergence. Throughout training, the primary loss function was MSE, which is appropriate for the regression task of predicting delay. We also tracked metrics such as MAE and R-squared on the validation set for comprehensive evaluation.

4.8.3 Hybrid Linear-LSTM-Transformer Model

1) Architectural design: The architectural design of the hybrid model combines a linear projection, an LSTM, and a Transformer encoder in sequence, forming a deep sequence model tailored for time-series regression. Its key components are:

a) Feature projection layer: Each input time step's feature vector (all QoS metrics for that timestamp) is passed through a linear layer that projects it to a latent dimension d_{model} (e.g., 32 or 64). A ReLU activation is applied, providing a learned representation of the raw features. This acts as an embedding layer that can emphasise relevant feature combinations before temporal modelling.

b) LSTM sequence encoder: The sequence of the projected feature vectors (of length 10) is then fed into an LSTM network. We utilised an LSTM layer (with a hidden state of 64, as determined through tuning) to capture temporal dependencies. The LSTM processes the sequence in order, maintaining an internal state that summarises information from previous time steps. For deeper variants, dropout was applied between LSTM layers to regularise, but the final chosen architecture kept a single LSTM layer for simplicity and to avoid overparameterisation.

c) Transformer encoder layer: The output sequence from the LSTM is next passed into a Transformer encoder. The Transformer layer utilises multi-head self-attention (e.g., 4 heads) to enable each time step in the sequence to attend to other time steps, thereby capturing long-range or contextual relationships that the LSTM might not fully exploit. In our implementation, we used a single Transformer encoder layer (with an internal feed-forward dimension of $2 \times \text{hidden\_size}$) on top of the LSTM outputs. This hybrid stacking (LSTM + Transformer) combines the strengths of recurrent memory with self-attention, potentially improving sequence representation for prediction. The Transformer's output sequence has the same length (10) and hidden dimensionality as its input, with each time step's representation now enriched by attention to the whole sequence.

d) Regression head: Finally, the model predicts the delay using the Transformer's output at the last time step of the sequence. This last-step representation (corresponding to the most recent time slot, after incorporating context from previous slots) is fed into a fully connected regression head. The head consists of a dense layer (mapping from the hidden size to an intermediate size, e.g., 64), a ReLU activation, a dropout layer, and a final linear layer that outputs a single scalar. This structure enables a nonlinear transformation of the learned features before the final prediction. The use of dropout in the head (and in the LSTM, as noted) provides regularisation to reduce overfitting.

2) Implementation and training: The hybrid model was implemented as a PyTorch *nn.Module* class that enables end-to-end backpropagation training. Input sequences (of shape [batch_size, 10, num_features]) are fed through the network to produce a batch of outputs (shape [batch_size, 1]). We trained the model using the Adam optimiser with an initial learning rate in the range of $1e-3$ to $1e-5$ (the exact value optimised via Optuna). Training was performed in mini-batches (batch size: 64-256, also tuned) using a DataLoader that shuffles sequences within each epoch. The loss function was MSE, and

gradients were computed using backpropagation through the LSTM, Transformer, and linear layers. We employed early stopping as described and used a learning rate scheduler to halve the learning rate upon stagnation of the validation loss. The typical training duration was on the order of tens of epochs (a maximum of 80), after which the best model (lowest validation MSE) was saved for evaluation. Due to the fixed random seed and consistent train and validation splits, training results were reproducible across runs.

3) Inference mode: Using the trained hybrid model to predict network delay works as follows. For a given device at the current time, we assemble the feature values from the last 10 time steps (e.g., the past 10 reporting intervals, including the current time’s features) into the same format as the training sequences. This 10-step feature sequence is fed through the model, which processes it with the linear projection, LSTM, and self-attention layers, and outputs a single predicted delay for the last time step. In essence, at inference, the model performs a one-step-ahead prediction (where “ahead” refers to predicting the present delay given the present metrics). The hybrid model runs efficiently at inference, as sequence lengths are fixed and modest, and all operations operate on vectorised representations. It can readily be used for real-time delay prediction by sliding its input window forward as new metrics arrive, outputting the predicted delay for each new time step.

4.8.4 LSTM Model

1) Architectural design: The LSTM-based model is a pure RNN approach for sequence regression. In this model, the input is again a sequence of 10 time steps of features (the same pre-processed feature set per time step, including the delay). The model consists of an LSTM sequence encoder followed by a dense output head:

a) We employed an LSTM with one or more layers (up to 3 layers were explored) and a hidden state size in the range 64-256. The final chosen architecture used a single LSTM layer with a hidden size of 64 for the best generalisation. The LSTM reads the input sequence of feature vectors sequentially and produces a sequence of hidden states (one per time step). Internally, the LSTM gates learn to retain relevant temporal information while filtering out noise, making it well-suited to capture patterns such as periodicity and trends in the delay data.

b) The regression head takes the LSTM’s output at the final time step and maps it to the delay prediction. Similar to the hybrid model, the head is a small feed-forward network: a linear layer transforms the last hidden state (size 64) to an intermediate size (64), followed by a ReLU and dropout, then a final linear layer to output the single predicted value. This head structure (identical in form to the hybrid’s head) allows the model to combine the information in the last LSTM state into a nonlinear prediction. In effect, the LSTM’s last hidden state serves as a summary of the entire sequence’s information, and the dense layers learn the mapping from that summary to the target delay.

2) Implementation and training: The LSTM-based model was also implemented in PyTorch as a custom module (essentially a simplified variant of the hybrid model without the Transformer). During the training process, we created the same sequence datasets and

DataLoader pipeline for feeding sequences of 10. The training phase mirrored that of the hybrid model: using the Adam optimiser, minimising MSE loss, and applying early stopping and learning rate scheduling. We tuned the LSTM hyperparameters with Optuna, which indicated that a single-layer LSTM (without additional stacked layers) was sufficient, and that a moderate dropout rate (around 20-30%) on the recurrent layers and head was optimal. The model training typically converged faster than the hybrid model, given its smaller size. With a hidden state of 64 and a single layer, the LSTM model has approximately 2×10^4 trainable parameters (plus another $\sim 4K$ from the head), which is relatively lightweight. This made training feasible within a few dozen epochs. The final model chosen had its weights from the epoch with the lowest validation MSE.

3) Inference mode: Using the LSTM-based model for prediction is analogous to the hybrid model's procedure (minus the Transformer step). For a new data point (a device at the current time), we use the device's last 10 time steps of features and feed them into the LSTM network. The model processes the sequence and directly outputs the predicted delay for the current time step. Because the LSTM model depends only on the sequence of recent data, it can be deployed online: at each time interval, update the input window with the latest metrics and use the model to predict the current delay. Compared to the hybrid model, the pure LSTM has a simpler architecture and fewer computations, potentially resulting in faster inference. However, both are easily capable of real-time performance for the window size used.

4.8.5 Random Forest Baseline

1) Architectural design: As a baseline, we used an RF regressor, which is a non-neural model that operates on static feature vectors without sequence context. The RF is an ensemble of decision trees, where each tree independently learns a series of thresholding rules on the input features to predict the target, and the ensemble's prediction is the average of all trees' outputs. For our problem, the input to the RF at time t is the set of QoS features for that device at that instant (e.g., throughput, loss, etc.), and the output is the predicted delay at time t . Unlike the LSTM and hybrid models, the RF does not explicitly consider time history (i.e., it does not have sequential input). We did not augment the RF model with lag features or rolling statistics beyond what was already provided (such as the average throughput feature) to maintain a straightforward baseline. Key RF parameters include the number of trees (estimators) and the depth of each tree, which control the model's capacity. A larger number of trees and greater depth enable the model to capture more complex relationships, albeit at the risk of overfitting. We tuned these parameters using Optuna: the final model, for instance, used on the order of a hundred trees, each up to tens of levels deep (Optuna suggested an optimal balance between depth and leaf criteria). Because the RF can handle nonlinear feature interactions, it provides a strong benchmark on tabular data, though we expect it to be limited by its lack of temporal awareness.

2) Implementation and training: We used the scikit-learn implementation of *RandomForestRegressor* for training and validation. The model was trained on the pre-processed training set. This means that each training sample for the RF was a feature

vector corresponding to a single timestamp of a single device, with features scaled and encoded as described earlier (including the device indicator as an ordinal value). Training the RF involved fitting the ensemble to minimise MSE on the target delay across all training samples. There is no epoch-based training in tree models; instead, the algorithm builds each decision tree by recursively splitting the data based on feature thresholds to reduce delay variance. Additionally, we utilised the optimal hyperparameters identified by Optuna’s search (including settings for *min_sample_split*, *min_sample_leaf*, and *max_features* for splits) and set the ensemble’s *random_state* to 42 for consistency. The resulting model can be seen as learning an average mapping from instantaneous QoS metrics to delay, aggregated over the training devices.

3) Inference mode: Deploying the RF model is straightforward: for a new observation (device and its metrics at time t), the model directly takes the current feature vector and outputs a predicted delay. Each of the trees in the forest produces a prediction by following the series of splits corresponding to that feature vector, and the forest’s output is the mean of all outputs. This process is fast, as it involves simple conditional traversal and averaging. Nevertheless, because the RF does not explicitly use past information, it cannot adjust its prediction based on recent trends or patterns beyond what the current feature values reflect. This means the baseline might miss temporal dynamics that the LSTM or hybrid model could capture. Nonetheless, it serves as a valuable benchmark for quantifying the benefit of sequence-based modelling. The Random Forest’s performance on the held-out devices provides a reference point for how well a memory-less regression can do on this task, and any improvement by the LSTM or hybrid model can be attributed to their ability to leverage temporal context.

4.8.6 Comparative Summary of Model Architecture

Table 4.13 summarises the principal characteristics of the three models, highlighting their different structures and capacities:

Table 4.13: Key Architectural Features of the Three Models Compared.

Model	Sequence-based Input	Architecture Components	Approx. Trainable Parameters
Hybrid (Linear+LSTM+Transformer)	Yes (window of 10 time steps)	Linear projection (per time step) $\rightarrow$ 1-layer LSTM (hidden size ~ 64) $\rightarrow$ 1 Transformer Encoder layer (4 heads, hidden size ~ 64) $\rightarrow$ Dense regression head ($64 \rightarrow 1$ with dropout)	70K
LSTM Model	Yes (window of 10 time steps)	1-layer LSTM (hidden size ~ 64) $\rightarrow$ Dense regression	20K

		head (64→1 with dropout)	
RF (Baseline)	No (single time step only)	Ensemble of ~100 decision trees (depth up to ~20; features: throughput, loss, etc., plus device ID)	N/A (non-neural model)

The “Sequence-Based Input” indicates whether the model uses a sequence of past time steps (and the number of time steps) for each prediction. The column for “Architecture Components” describes the layers or structures in the model (for neural networks) or the composition of the ensemble (for the Random Forest). The column for “Approx. Trainable Parameters” gives the order-of-magnitude of tunable weights in each model, not applicable to the Random Forest, which instead has complexity determined by the number and depth of trees. Each neural model’s design (layers, dimensions, etc.) was optimised through hyperparameter tuning (Optuna) to balance model complexity and performance. The hybrid model incorporates both recurrent and self-attention mechanisms; the LSTM model uses only recurrence, and the RF uses no sequential modelling, relying purely on instantaneous features.

4.9 Design and Implementation of a Transformer-based RL Agent for Network Traffic Scheduling

This section discusses the design and implementation of a transformer-based RL algorithm for network scheduling in 5G WSNs. This approach leverages the advanced sequence modelling capabilities of the Transformer architecture, enabling efficient handling of complex data sequences. It integrates it within a deep reinforcement-learning framework to optimise network traffic scheduling and enhance QoS in communications networks. The accompanying Table 4.14 provides a comprehensive overview of the essential mathematical symbols and their corresponding definitions, employed consistently across the design and implementation stages of the proposed Transformer-based DRL model. The model’s performance results and experimental trials are discussed in Section 5.6 and revisited in Section 6.6, where its broader contributions and alignment with industrial benchmarks are discussed.

Table 4.14: Mathematical notations and definitions.

Notation	Definition
S	A set of all possible states in the network environment
A	A set of all possible actions the scheduler can take
P	Transition probability function: $P: S \times A \times S \rightarrow [0,1]$
R	Reward function: $R: S \times A \rightarrow \mathbb{R}$
γ	Discount factor for future rewards: $\gamma \in [0,1]$

s_t	State of the network at time t
a_t	Action taken at time t
r_t	The reward received at time t
v_t	Encoded state vector produced by the Transformer
θ	Parameters of the policy network
ϕ	Parameters of the value network (when used)
$\pi(a_t s_t; \theta)$	Policy function: the probability of taking action a_t at a given state s_t
$V(s_t; \phi)$	Value function: expected return from state s_t
$O^{(l)}$	Output of layer l in the Transformer
X	Input to a Transformer layer
$J(\theta)$	The objective function for policy optimisation
α	The learning rate for parameter updates
ε	Exploration rate in ε -greedy strategy
$E[\cdot]$	Expectation operator
∇_{θ}	Gradient with respect to θ
LayerNorm($\cdot$)	Layer normalisation operation
MultiHead($\cdot$)	Multi-head attention operation in the Transformer
Softmax($\cdot$)	Softmax function for the action probability distribution

4.9.1 Problem Formulation

The network traffic-scheduling problem in communication networks is an RL problem, where the network state encompasses information on traffic demands (including traffic volume and data transmission characteristics, network topology and infrastructure, current traffic flows across different services, QoS requirements, dynamic traffic patterns, and user behaviour). The actions correspond to dynamic scheduling decisions that optimise network performance, including intelligent routing of data packets, adaptive resource allocation across network segments, prioritisation of critical traffic types, dynamic bandwidth management and real-time traffic flow adjustments. The objective is to maximise a reward function that reflects network performance metrics, aiming to minimise latency, maximise throughput, and ensure optimal QoS across diverse network communications demands. Additionally, the problem can be mathematically formulated as a Markov Decision Process (MDP), as defined by a tuple (S, A, P, R, γ) , where:

$$MDP = (S, A, P, R, \gamma) \quad (1)$$

4.9.2 State Representation

A Transformer encodes the network state into a high-dimensional vector representation by modelling the current state of the network, including traffic patterns, resource utilisation/prioritisation, and topology as structured input sequences. This allows the Transformer to capture complex dependencies and interactions within the network, enabling a detailed and dynamic understanding of its operational state for effective traffic scheduling and resource allocation. The state representation can be mathematically represented as follows:

$$v_t = \text{Transformer}(s_t) \quad (2)$$

4.9.3 Action Space Definition

A well-defined set of potential actions that the algorithm can take in each state is essential. These actions may include allocating bandwidth, optimising routing paths, or scheduling traffic flows. This set of actions constitutes the action space A , which is a critical component of the Markov Decision Process (MDP) framework that enables the algorithm to interact with and adapt to the network environment effectively.

4.9.4 Reward Function Design

A reward function quantifies the network's performance at each scheduling decision by evaluating key metrics such as latency, packet loss rate, and throughput. This function serves as the objective criterion within the Markov Decision Process (MDP), guiding the model to optimise traffic scheduling or resource allocation by balancing trade-offs between these performance metrics. The reward at time t for acting a_t in state s_t is defined as:

$$r_t = R(s_t, a_t) \quad (3)$$

4.9.5 Transformer Architecture Setup

The Transformer architecture, well-suited for network traffic scheduling in 5G and beyond, leverages a multi-layered structure. It employs self-attention mechanisms, allowing the model to weigh the importance of elements within the network's state representation, such as queue lengths, channel conditions, and application demands. This dynamic weighting enables the Transformer to capture intricate dependencies and relationships among these elements, such as the interplay among varying traffic patterns, diverse QoS requirements of emerging applications (such as autonomous driving, remote surgery, and AR/VR), and fluctuating radio resource availability. Following the self-attention layers, feedforward neural networks process this contextualised information to generate optimal scheduling decisions, dynamically allocating resources to minimise latency, maximise throughput, and ensure fairness among users and applications. Transformer architecture can be mathematically expressed as:

$$Z^{(l)} = \text{LayerNorm}[X^{(l-1)} + \text{MultiHead}(X^{(l-1)}, X^{(l-1)}, X^{(l-1)})] \quad (4)$$

$$X^{(l)} = \text{LayerNorm}(Z^{(l)} + \text{FFN}(Z^{(l)})) \quad (5)$$

Where the feedforward network is defined as:

$$\text{FFN}(Z^{(l)}) = \sigma(Z^{(l)} W_1 + b_1) W_2 + b_2 \quad (6)$$

Explanation

1) Input ($X^{(l-1)}$): This is the output of the previous layer or the initial embedding layer.

2) Self-Attention Sub-Layer:

- Multi-Head Attention: Computes attention scores to capture relationships between different positions in the input sequence.
- Residual Connection: Adds the input $X^{(l-1)}$ to the multi-head attention output to improve gradient flow.
- Layer Normalisation: Normalises the summed output to stabilise and accelerate training.

3) Intermediate Output ($Z^{(l)}$): This is the result obtained after the self-attention sub-layer, including the residual connection and layer normalisation, and it serves as the input to the feedforward network sub-layer.

4) Feedforward Network Sub-Layer:

- First Linear Transformation: The first linear transformation projects ($Z^{(l)}$) into a higher-dimensional space.
- Activation Function (σ): The activation function applies a nonlinear activation to capture complex patterns.
- Second Linear Transformation: Projects back to the original dimensionality.

5) Second Residual Connection and Layer Normalisation:

- Residual Connection: It adds $Z^{(l)}$ to the feedforward network's output.
- Layer Normalisation: This layer normalises the result to produce the final output for the layer ($X^{(l)}$).

The Transformer Encoder block processes the input sequence through self-attention and feedforward sub-layers, followed by residual connections and layer normalisation. This architecture allows the model to capture local and global dependencies efficiently. A stack of such encoder blocks forms the complete Transformer Encoder, enabling the creation of deep representations of the input data.

4.9.6 Policy Network Definition

The Transformer is used as part of the policy network, and in the context of network traffic scheduling for delay-sensitive applications, a Transformer-based policy network acts as an intelligent controller. It inputs an encoded representation of the network state (including queue lengths, arrival rates, traffic types, channel conditions, and packet deadlines). The Transformer's attention mechanism enables it to weigh the importance of different state features and capture complex dependencies, allowing it to output a probability distribution over possible scheduling actions, such as prioritising specific queues, dropping packets, or adjusting transmission rates. The policy network can be mathematically defined as:

$$\pi(a_t|s_t; \theta) = \text{Softmax}(\text{Transformer}(s_t; \theta)) \quad (7)$$

This policy network is trained using RL, in which it interacts with the environment, receives rewards based on performance (e.g., low delay, high throughput), and iteratively improves its scheduling decisions. This approach benefits from the Transformer's ability to handle long-range dependencies and contextual information, leading to more efficient and adaptive traffic management, especially in dynamic network conditions where prioritising critical traffic is crucial.

4.9.7 Value Network Construction

Constructing a value network is optional in this Transformer-based RL approach for network traffic scheduling. This design choice offers flexibility, mainly as our implementation relies on a policy network that directly learns the probability distribution over actions. This enables the model to address many reinforcement-learning problems without explicitly estimating state values. Furthermore, omitting the value network reduces computational overhead, a critical factor for real-time network scheduling. The resulting simpler architecture, consisting solely of a policy network, facilitates easier deployment and maintenance, especially in scenarios with clear reward signals where the policy network demonstrates strong performance. The mathematical expression of the value network is:

$$V(s_t; \phi) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t; \phi] \quad (8)$$

4.9.8 Training Loop Initialisation

At the start of the training loop for the Transformer-based RL scheduler, designed for 5G network traffic and emerging applications, the DRL algorithm begins interacting with either the live network or a simulated environment. This interaction is crucial for gathering data; the agent observes the network state, takes scheduling actions and receives feedback in the form of rewards or penalties that reflect the effectiveness of those actions.

This process is iterative and occurs over several discrete time steps or episodes. Through these repeated interactions, the agent accumulates experience, allowing it to learn the relationship between its actions and their consequences. This learned experience then guides the agent in developing optimal scheduling policies for efficient, responsive network traffic management in dynamic 5G and future network environments.

4.9.9 Exploration vs. Exploitation

To effectively balance the trade-off between discovering new actions and leveraging known optimal ones, an exploration strategy such as ϵ -greedy is employed. This balance is critical for enabling the model to adapt and learn a robust scheduling policy that can perform efficiently across diverse and dynamic network conditions. By systematically exploring untested actions while capitalising on established knowledge, the scheduler optimises performance in both familiar and unforeseen scenarios. The strategy can be mathematically expressed as:

Choose a random action with probability ϵ and the best-known actions with probability $1 - \epsilon$

4.9.10 Gradient Update and Learning

During training, the Transformer model learns to optimise network traffic scheduling by iteratively refining its policy and value function. After processing each batch of network scenarios and scheduling decisions, the algorithm calculates the gradients of the objective function. These gradients quantify how changes in the Transformer's parameters would affect the expected cumulative reward, which is determined by comparing the predicted scheduling performance with the observed performance (e.g., reduced latency or improved throughput). The core of the learning process is backpropagation, which efficiently propagates gradients through the Transformer network, enabling adjustment of its internal weights. Specifically, the policy network (which directly selects scheduling actions) and the value network (which estimates the long-term value of states) are updated via gradient ascent. This update rule can be mathematically expressed as:

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta} J(\theta) \quad (9)$$

The update rule for policy parameters θ , analogously for value function parameters ϕ , adjusts the parameters in the direction that maximises the objective function $J(\theta)$, leading to improved scheduling decisions over time. The learning rate α controls the step size of these updates, balancing the learning speed with the risk of overshooting the optimal parameter values.

4.9.11 Policy Evaluation and Adjustment

Continuous performance evaluation is vital for ensuring the effectiveness and robustness of the Transformer-based RL scheduler. Throughout the training process, the policy network's performance is regularly monitored using relevant metrics such as average latency, throughput, and packet loss rate. This ongoing assessment provides valuable

insights into learning progress and identifies potential areas for improvement. Based on the observed performance, adjustments to the learning strategy may be necessary. These adjustments could involve fine-tuning parameters such as the learning rate or discount factor, modifying the reward function to align with the desired scheduling objectives, or altering the exploration-exploitation balance to encourage more diverse or refined scheduling decisions. This iterative cycle of evaluation and refinement ensures that the model learns effectively, adapts to changing network conditions, and optimises its performance towards the desired goals, as reflected in the objective function.

4.9.12 Deployment and Real-Time Adaptation

Deployment of the trained DRL model involves integrating it into the live 5G network environment for real-time traffic scheduling. Once deployed, the model does not remain static; instead, it continues to learn and adapt through online training. This means that the Transformer-based scheduler interacts directly with the network, observing the consequences of its scheduling decisions and receiving immediate feedback in the form of rewards or penalties. This continuous stream of real-time data enables the model to dynamically adjust its policy, ensuring it remains effective even as network conditions fluctuate due to changes in traffic demand, resource availability, or interference patterns. This online, adaptive learning approach is crucial for maintaining optimal performance in the dynamic and often unpredictable environment of a 5G network.

Furthermore, leveraging a Transformer-based architecture, this DRL algorithm demonstrates exceptional suitability for traffic scheduling in complex environments, such as 5G WSNs. Its ability to process and extract insights from high-dimensional sequential data enables the optimisation of real-time scheduling decisions, addressing the dynamic and multifaceted requirements of such networks.

The mathematical expressions presented above establish the theoretical foundations of the DRL system employing a Transformer architecture and highlight the intricate mathematical relationships that define interactions among its components.

4.9.13 Experimental Setup in a Live Network

1) Network Configuration: This subsection details the experimental setup implemented in a live network to evaluate the performance of the proposed Transformer-based RL scheduler. The setup is meticulously designed to optimise the scheduling of ping requests to various devices and sites, aiming to minimise latency (as measured by RTT) while ensuring a high success rate for transmitted pings.

The experimental network topology consists of a router, four Raspberry Pi devices, a smart TV, two iPhones, a Samsung Galaxy tablet, and an external site. The software stack includes essential firmware and network management tools for real-time data acquisition and analysis.

This small-scale topology provides a controlled environment representative of a segment of a 5G WSN comprising 10 nodes. The router serves as the central hub for managing network communications. Four Raspberry Pi devices are deployed, one operating the

proposed model and the remaining three functioning as sensor nodes. Additional endpoints include a smart TV, two iPhones, a Samsung Galaxy tablet, and an external site (public DNS resolver). Each node fulfils a specific role, such as handling network traffic, collecting sensor data, or serving as a communication target, facilitating comprehensive testing and evaluation of the model and the network's performance.

All devices in the network were actively generating traffic under practical conditions. The smart TV streamed YouTube videos, the iPhones browsed social media, and the tablet played online games. Notably, the three Raspberry Pi devices targeted for ping requests also streamed YouTube video, simulating a realistic sensor node under load. These four nodes and the public DNS resolver were monitored explicitly during the experiment. Meanwhile, the non-targeted devices contributed additional background traffic, creating a dynamic and realistic network environment representative of typical real-world usage. Table 4.15 provides the specifications of the devices in the network.

Table 4.15: Specification of Devices in the Network Architecture.

Device	Model Number	Operating System	Communication Range	Sensor Type
Router (ZTE Home Gateway)	ZXHN H298A V1.0	V1.0.25_HOP.1T11	Wi-Fi IEEE 802.11, up to 45 m indoors	N/A
Raspberry Pi 4	Model B Rev 1.4	Ubuntu 22.04.4 LTS 64-bit	Wi-Fi IEEE 802.11b/g, up to 45 m indoors	N/A
Raspberry Pi 4	Model B Rev 1.4	Ubuntu 22.04.4 LTS 64-bit	Wi-Fi IEEE 802.11b/g, up to 45 m indoors	DHT22 Temperature Sensor
Raspberry Pi 3	Model B Rev 1.2	Raspbian GNU/Linux 11 (bullseye)	Wi-Fi IEEE 802.11b/g, up to 45 m indoors	PIR Motion Sensor
Raspberry Pi 2	Model B Rev 1.1	Raspbian GNU/Linux 11 (bullseye)	Wi-Fi IEEE 802.11b/g, up to 45 m indoors	N/A
Smart TV (LG webOS TV)	OLED65A16LA	WebOS TV Version 6.4.0-21 (kisscurl-kluane)	Wi-Fi IEEE 802.11b/g, up to 45 m indoors	Built-in-Sensor
iPhone 14 Plus	MQ513ZD/A	iOS 17.5.1	Wi-Fi 6, IEEE 802.11ax, up to 30 m indoors	Built-in-Sensors

iPhone 6s Plus	NKU12ZD/A	iOS 15.7.2	Wi-Fi IEEE 802.11ac, up to 30 m indoors	Built-in-Sensor
Galaxy Tab A7 Lite	SM-T220	Android Version 11	Wi-Fi IEEE 802.11ac, up to 30 m indoors	Built-in-Sensors
External Site (Public DNS Resolver)	N/A	Server OS	Global	N/A

The logical network topology and traffic model shown in Figure 4.13 provides a simplified yet realistic environment for testing the Transformer-based RL scheduling algorithm, allowing for observing and fine-tuning the model's behaviour. This controlled setting is essential for validating the core principles of the proposed approach before scaling up to more complex and extensive networks. This study enables a more targeted assessment of the Transformer-based RL scheduler by incorporating six active nodes into this experimental configuration.

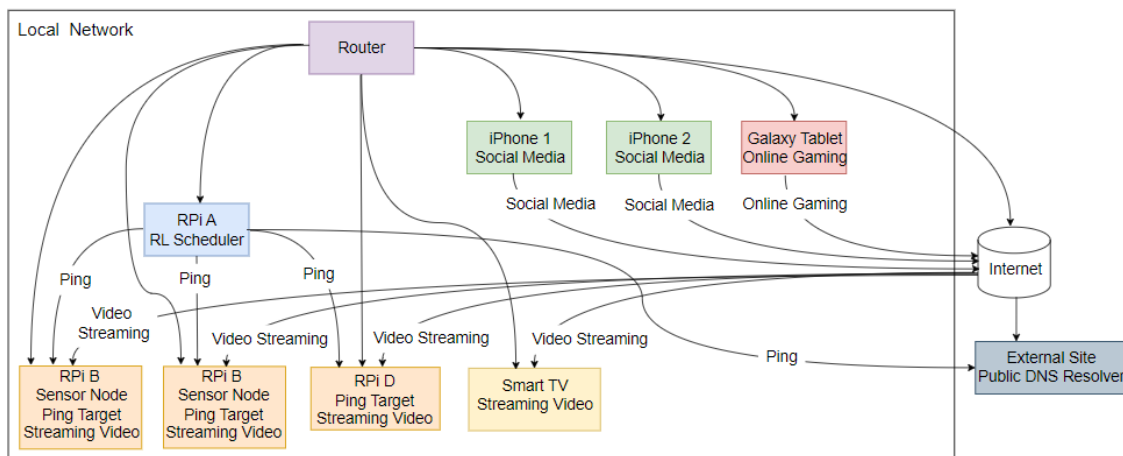


Figure 4.13: Network Topology and Traffic Model.

Although simplified, this experimental setup reflects a typical 5G WSN scenario in which distributed nodes manage real-time data communication with minimal latency. Bringing together a router, four Raspberry Pi devices (one running the proposed model and three attached to sensors), and a public Domain Name System (DNS) resolver creates a manageable yet representative testbed for evaluating the model's performance under realistic conditions. After its effectiveness is confirmed, this configuration can be expanded to include more nodes and increasingly complex network topologies.

The experimental setup evaluates the scheduler's performance under various conditions to ensure a comprehensive assessment, which includes fluctuating network loads and changing latency and reliability requirements. This approach is consistent with the

methodological frameworks in the field’s literature, which often start with smaller-scale experiments to validate new algorithms before scaling them up.

The hyperparameters of the Transformer-based RL scheduler are optimised using Optuna, a hyperparameter optimisation framework. The learning rate is searched from 1e-6 to 1e-3 to identify the optimal parameter that balances model complexity and performance. Initial experiments and benchmarks suggest that a hidden dimension size of 128 is suitable for the task.

Table 4.16 summarises the model configuration and parameters, including the ping targets, number of training epochs, the Transformer hidden layer size, and the learning rate range.

Table 4.16: Transformer-based RL Model Configuration and Parameters.

Component	Parameter	Value
General Configuration	Sites/IP addresses	[“192.168.1.140”, “192.168.1.141”, “192.168.1.142”, “1.1.1.1”]
	Number of sites/IP addresses (NUM_SITES)	4
	Number of Epochs (NUM_EPOCHS)	250; 350; 500
	Hidden Dimension (HIDDEN_DIM)	128
	Training Steps per Epoch	100
Transformer Scheduler Model	Embedding Layer	- Input Dimension: NUM_SITE + 1 - Output Dimension: HIDDEN_DIM (128)
	Transformer Encoder	- Encoder Layer: 2 - Model Dimension (d_model): HIDDEN_DIM (128) - Number of Heads (nhead): 4
	Decoder Layer	- Input Dimension: HIDDEN_DIM (128) - Output Dimension: NUM_SITES (4)
	Activation Function	Softmax (applied over the output of the decoder)
Training Configuration	Optimizer	Adam optimizer

	Learning Rate (LEARNING_RATE)	Suggested by Optuna (log-uniform distribution between 1e-6 and 1e-3)
	Loss Function	Policy Gradient Loss (-log_prob * reward)
	Reward Calculation	1000 – RTT (RTT: from ping_site)
	State Update Mechanism	The next state is set to the current action (state = action)
Hyperparameter Optimization	Optimization Library	Optuna
	Objective	Maximise the scheduling success ratio.
	Hyperparameter Tuned	Learning Rate (LEARNING_RATE)
	Search Space	The log-uniform distribution between 1e-6 and 1e-3
	Number of Trials	10
Performance Metrics	Success Criteria	RTT not equal to 1000.0 (This is used to identify successful pings)
	Success Ratio Calculation	Successes / (NUM_EPOCHS * Training Steps per Epoch)
	Execution Time Measurement	Total time taken to complete all epochs and trials (time.time () – start_time)
Visualization Components	Latency per Site	Bar chart showing RTT for each site (axis [0, 0])
	Training Loss over Epochs	Line plot of training loss across epochs (axis [0, 1])
	Reward over Epochs	Line plot of rewards across epochs (axis [1, 0])
	Reliability (Success Ratio) over Epochs	Line plot of success ratio across epochs (axis [1, 1])

2) Implementation details: The Transformer-based RL scheduler is implemented in Python using the PyTorch framework and is integrated into the existing network infrastructure to optimise ping scheduling across multiple sites/IP addresses. The implementation involves:

- Deploying the scheduling algorithm on a Raspberry Pi device that serves as the network management node.
- Configuring scheduler parameters, including hyperparameters derived from model training, to meet network requirements.
- Conducting integration testing to ensure stability and performance within the live network environment.

3) Model Architecture: The network traffic scheduler utilises a Transformer-based neural network augmented with RL techniques. The current state is a numerical identifier corresponding to the selected site or IP address. This integer is processed through an embedding layer to generate a dense representation suitable for the Transformer encoder. The model comprises two Transformer encoder layers, each with four attention heads. These attention heads capture sequential dependencies within the state-action space, enhancing the model's ability to learn optimal scheduling policies. A linear decoder, followed by a softmax activation function, produces a probability distribution over potential actions (sites or IP addresses to ping), enabling stochastic policy exploration.

4) Training Process: The model is trained online using real-time network data, enabling it to adapt to dynamic and non-stationary network conditions, such as fluctuating traffic loads and varying resource availability, which are often poorly captured by offline training datasets. This adaptability enhances performance in real-world scenarios where network characteristics frequently change. Online training eliminates the need for potentially inaccurate network models or simulations, as the scheduler learns directly from actual network behaviour. This direct interaction with the environment enables the scheduler to capture complex and subtle network dynamics that might be missed by simplified models, resulting in more efficient and robust scheduling policies. Furthermore, continuous online learning allows the scheduler to continuously refine its performance and track changes in the network over time, ensuring sustained high performance even as the network evolves.

The learning rate is optimised using Optuna within the range of 1×10^{-6} to 1×10^{-3} . This method offers the advantage of fine-grained exploration for the most relevant learning rate magnitudes. This range typically encompasses values that support stable, efficient learning in many DL models. Searching within this space enables Optuna to identify a learning rate that balances rapid convergence with avoiding overshooting and divergence, resulting in faster training and improved generalisation performance compared to a broader, less focused search or manual tuning. Other hyperparameters include a hidden dimension of 128, a batch size determined by the number of steps per epoch (100 steps), and 250 epochs (each experiment used a different number of epochs).

At each training step, the agent observes the current state and samples an action from the model's state-conditioned action distribution, $\pi(a|s)$. This approach enables the exploration of different sites and IP addresses while gradually focusing on those that yield higher rewards. The `ping_site` function uses ICMP to send a ping request to the selected site and returns the RTT as a latency measure. In the event of a ping failure, a high RTT value of 1000 ms is assigned to represent poor performance. Rewards are calculated using

the formula $\text{reward} = 1000 - \text{RTT}$, where lower RTT values yield higher rewards. This design incentivises the model to minimise latency by favouring sites/IP addresses with lower RTTs. The loss is computed using the policy gradient method, expressed as $-\log(\text{probability of Selected Action}) \times \text{Reward}$. This formulation provides a way to quantify the effectiveness of a policy's actions and to adjust the policy's parameters to favour actions that lead to higher rewards. Thus, it encourages the RL scheduler to assign higher probabilities to scheduling decisions that result in higher rewards. The Adam optimiser updates the model weights based on the calculated loss, facilitating efficient convergence during training.

5) Handling of Ping Failures: The results of failed ping attempts are assigned a higher RTT value (1000 ms) to account for network unreliability. The reward function ($1000 - \text{RTT}$), with failed pings assigned an RTT of 1000 ms, effectively translates network reliability into an RL signal. The policy gradient is penalised for actions leading to unreliable communication by assigning a maximum RTT to failures. This approach encourages the RL scheduler to learn policies that prioritise network stability and minimise latency, as maximising this reward directly corresponds to minimising RTT and avoiding communication failures. The design incorporates a practical measure of network performance (RTT) from a live network environment into the learning process, guiding the model toward robust, reliable network actions.

6) Success Ratio Calculation: The success ratio, defined as the proportion of successful ping attempts to the total number of attempts, is a quantitative metric for evaluating network reliability and the efficacy of the RL scheduler in selecting responsive sites. Mathematically, the success ratio is calculated as the quotient of the number of successful pings and the total number of pings:

$$\text{Success Ratio} = \frac{\text{Number of Successful Pings}}{\text{Total Number of Pings}}$$

A higher success ratio indicates a more reliable network and an effective RL scheduler, suggesting greater ability to identify responsive sites or IP addresses. On the other hand, a lower success ratio may indicate network instability or deficiencies in the RL scheduler's sites/IP address selection process. The success ratio provides a valuable, concise measure of network performance and the Transformer-based RL scheduler's performance.

7) Training Dynamics: The model undergoes dynamic policy adjustment during training, leveraging real-time feedback from the network environment. Empirical observations reveal a decreasing trend in the loss function and a concomitant increasing trend in cumulative rewards over training epochs, indicative of successful learning and progressive policy improvement. This online training approach enables the model to adaptively respond to fluctuations in network latency and reliability, thereby demonstrating robustness and resilience in a dynamic, live operational environment. The observed convergence of the loss function and the sustained increase in reward suggest that the model effectively learns to optimise site/IP address selection based on real-time network conditions.

8) Visualisation and Results: A post-training analysis of key performance metrics evaluates the model's effectiveness. Latency, characterised by the RTT to each site/IP address, is visualised as Latency per Site to identify both latency characteristics and consistently high-performing sites/IP addresses. Model convergence and training stability are assessed by plotting Training Loss as a function of epoch. Similarly, Rewards per Epoch are plotted to quantify the model's success in latency optimisation. Finally, the reliability of the scheduler's actions is determined by visualising the Success Ratio over Epochs.

9) Hyperparameter Optimisation: The learning rate is optimised using Optuna, a hyperparameter optimisation framework. Multiple trials are conducted within a specified range of learning rates. The optimal learning rate is the value that maximises the success ratio, balancing convergence speed and training stability.

10) Real-Time Data Handling: The model's online training within an operational live network environment presented advantages and challenges. The training ensured high practical relevance and adaptability to real-world conditions by leveraging real-time network data. However, the dynamic nature of live networks introduced fluctuations that required the model to continually adapt, underscoring the critical role of robust real-time learning mechanisms in maintaining performance and reliability.

Figure 4.14 presents a high-level functional decomposition of the Transformer-based RL scheduler agent pipeline. The SITES array feeds into the central TransformerScheduler class, which comprises the core model blocks (embedding layer, transformer encoders, decoder layer, and softmax layer) and helper functions. These helpers implement runtime and learning operations: ping_site for probing targets, choose_action for decision selection, and train_model for training, which further branches into hyperparameter setup, model initialisation, the training loop, visualisation, and an Optuna study that drives automated tuning.

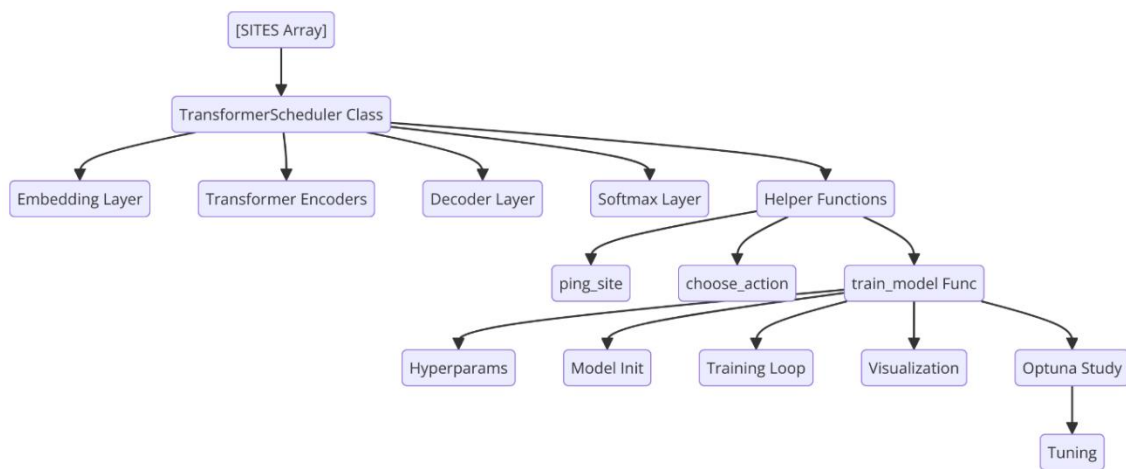


Figure 4.14: Flowchart of the proposed model.

Chapter 5: Results / Findings and Discussion

5.1 Introduction

This chapter presents the results of a series of experiments designed to address QoS challenges in 5G WSNs and beyond, with a focus on Ultra-Reliable and Low-Latency Communications (URLLC). The overarching goal of this research, as outlined in the project titled “Selective Prioritisation of Real-Time IP Packets to Improve QoS in 5G Wireless Sensor Networks,” is to develop and evaluate novel mechanisms for improving QoS, particularly end-to-end delay and reliability, for emerging applications demanding URLLC capabilities.

The core problem addressed in this research stems from the inherent characteristics of real-time traffic, including variable bit rates, sensitivity to network parameters (such as latency, jitter, and packet loss), and the need for strict, concurrent QoS guarantees. The anticipated explosion of connected devices exacerbated these challenges, leading to increased congestion, interference, and difficulty in maintaining QoS for safety-critical applications. Traditional scheduling and resource allocation methods often struggle with the scale and dynamism of these future networks. This research study explores the application of ML, DL, and DRL techniques to achieve more adaptive and efficient QoS management.

The experimental results and comparative analyses presented in this Chapter are based on the system architectures, models, and procedures developed in *Chapter 4*. Each experiment directly reflects the implementation strategies, dataset configurations, and algorithmic frameworks previously outlined. These results serve as the empirical foundation for the broader interpretations, implications, and contributions discussed in Chapter 6, where the findings are synthesised to address the research questions and inform future work on QoS optimisation in 5G WSNs.

5.1.1 Research Problems and Scientific Questions

The fundamental scientific research question that drives this work is: **Can advanced AI techniques, specifically ML, DL, and DRL, whether as a part of a broader system or a standalone tool, be effectively applied to dynamically prioritise and schedule real-time IP packets in 5G WSNs, thereby significantly improving QoS metrics (latency, reliability, jitter) compared to traditional methods, especially under conditions of high network load and diverse traffic demands?**

This study aims to address the established aims and objectives by formulating the following research problems and questions:

1) Experiments on Scheduling Algorithms and Data-Driven (Sections 5.2, 5.3, and 5.4):

a) Experiment 1 (Section 5.2 – Comparative Evaluation of Scheduling Algorithms Using Regression Models)

- **Research Problem:** How can traditional scheduling algorithms such as Proportional Fair (PF) and Round Robin (RR) be quantitatively compared using key performance indicators extracted from live network data?
- **Scientific Question:** Can regression techniques such as EGPR reveal underlying relationships between network performance metrics and help identify which scheduling strategy offers better QoS under dynamic real-time conditions?

b) Experiment 2 (Section 5.3 – Classification-Based Comparison Using Operational Network Traffic)

- **Research Problem:** How effectively can ML and DL classification models differentiate scheduling disciplines (Prio, HTB, and CBQ) using raw operational network traffic?
- **Scientific Question:** Is it possible to accurately classify and predict scheduling performance based solely on traffic patterns, and what insights do these predictions offer regarding the inherent characteristics of each scheduling discipline?

c) Experiment 3 (Section 5.4 – Comparison using DL Models)

- **Research Problem:** To what extent can DL models (CNN, CNN-LSTM) leverage image-based representations of network traffic data to predict scheduling outcomes and QoS performance?
- **Scientific Question:** Can advanced DL architectures capture complex spatiotemporal features in network data to improve scheduling prediction accuracy in live network scenarios?

d) Experiment 4 (Section 5.5 – Comparative Evaluation of Network-Delay Prediction Models (Hybrid (Linear+LSTM+Transformer), LSTM, and Random Forest))

This experiment aims to enhance QoS metric prediction in 5G networks by:

- Accurately forecast the next-step network delay in a wireless network by using historical sequences of QoS metrics, a task complicated by the network's dynamic temporal behaviour.
- Do DL models that explicitly model sequential data (such as LSTM or Transformers) predict network delay more accurately than a strong, non-sequential machine-learning baseline (such as Random Forest)?

e) Experiment 5 (Section 5.6 – Transformer-Based RL Agent for Network Traffic Scheduling)

This experiment addresses the network scheduling challenge by framing it as a reinforcement-learning problem. Here, the network state – encompassing traffic demands, topology, and flow characteristics – guides the scheduling decisions (actions) to maximise a reward function that reflects throughput, delay minimisation, and overall network performance. The problem is formulated mathematically as a Markov Decision Process (MDP): $MDP = (S, A, P, R, \gamma)$.

5.1.2 Chapter Roadmap

The chapter is structured to progressively build from a comparative analysis of traditional scheduling algorithms using ML techniques to the development and evaluation of novel, AI-driven QoS prediction and scheduling mechanisms. Each experiment investigates a specific aspect of the problem, culminating in a cross-experiment discussion and broader implications.

1) Section 5.2 – Experiment 1: Comparative Evaluation of Scheduling Algorithms with ML regression Models

- Brief overview of the implementation of PF and RR algorithms in a live network.
- Presentation of the dataset and performance metrics.
- Analysis using the EGPR model.

2) Section 5.3 – Experiment 2: Classification-Based Comparison of Scheduling Algorithms using Operational Network Traffic.

- A brief overview of the configuration and deployment of Prio, HTB, and CBQ scheduling disciplines.
- Description of the image-based dataset generation.
- Comparative analysis using ML models (Optimised Tree, SVM, and Fine Tree) and DL models (CNN and CNN-LSTM).

3) Section 5.4 – Experiment 3: Comparison of Scheduling Algorithms using DL Models

- Presentation of results from CNN-based, CNN-LSTM-based, and LSTM-based approaches.
- Discussion on performance metrics and interpretation of results for the DL model.

4) Section 5.5 – Experiment 4: Hybrid (Linear +LSTM+Transformer) DL Model for QoS Prediction

- Detailed presentation of dataset summaries, performance metrics, and cross-validation results.
- Comparative discussion with similar studies and impact on QoS in 5G networks.

5) Section 5.6 – Experiment 5: Transformer-Based RL Agent for Network Traffic Scheduling

- Presentation of dataset, performance metrics, and experimental results.
- Discussion of model components, comparison with benchmarks, and alignment with industrial applications.

6) Section 5.7 – Cross-Experiment (Cross-Dataset) Comparison and Discussion

- Synthesis of findings across all experiments
- Comparative insights and identification of trends across different methodologies.

5.2 Experiment 1: Comparative Evaluation of Scheduling Algorithms with ML Regression Models

This section presents the experiments in this research study on WLAN scheduling algorithms conducted in a live network environment. In this section, we compare the performance of Proportional Fair (PF) and Round-Robin (RR) scheduling by collecting real-time QoS metrics, including delay, jitter, throughput, and packet loss, from heterogeneous devices. The datasets from these experiments served as a basis for applying an EGPR model to evaluate and predict scheduler performance under realistic network conditions.

This study framed this problem as a data-driven comparison of WLAN scheduling algorithms. Specifically, the goal is to quantify how different scheduling approaches balance performance and fairness in a heterogeneous network environment. The study systematically collected and analysed vital QoS metrics for ML models to discern patterns, predict performance outcomes, and ultimately guide the design of more efficient and equitable scheduling algorithms.

5.2.1 Proportional Fair Dataset Results

5.2.1.1 Summary of PF Dataset

This study utilises a two-hour, packet-level trace from an IEEE 802.ac WLAN operating under the PF scheduler. The dataset contains 54,984 device-round observations across four stations (192.168.1.140-192.168.1.143). Each record includes temporal data (timestamps and interarrival times), traffic metrics (per-round packet counts, instantaneous and average throughput, loss rate), and scheduler information (binary flags, PF score), along with the target variable, *delay_ms*. Collected in a live environment, the trace data accurately reflects genuine contention, channel variability, and user-driven traffic dynamics. These characteristics create a statistically rich yet compact tabular dataset, well-suited for supervised regression modelling of delay. This enables reproducible exploration of how PF decisions and instantaneous load conditions affect end-to-end latency.

1) Dataset overview: Table 5.1 summarises the PF scheduling dataset collected from our IEEE 802.11ac WLAN testbed over a continuous two-hour capture window. It provides an overview of the dataset scale (54,984 device-round records across four stations) and the 15 extracted features used for subsequent analysis and modelling.

Table 5.1: Proportional Fair Scheduling Dataset Overview.

Characteristic	Value
Total observations	54,984 device-round records (~4 stations x 13,746 scheduling rounds)
Capture window	2 hours

Medium and algorithm	IEEE 802.11ac WLAN, Proportional-Fair (PF) scheduler
Stations (IPs)	IP addresses (192.168.1.140 – 192.168.1.143)
Features (15)	Timestamp, station identifiers, round ID, latency and jitter, inter-arrival time, per-round packet counters, loss rate, instantaneous and average throughput, PF metric, and binary selection flag.

2) Descriptive statistics: Table 5.2 reports descriptive statistics for the PF dataset's continuous features, summarising central tendency and dispersion across delay, jitter, inter-arrival time, packet counters, loss rate, throughput, and the PF metric. This provides an initial view of typical operating ranges as well as the presence of variability and occasional extremes (e.g., tail delay and inter-arrival outliers).

Table 5.2: Summary of Continuous Features for the PF Dataset.

Feature	Count (obs)	Mean	Std	Min	Q1	Median	Q3	Max
Delay (ms)	13,707	11.09	16.54	3.10	6.82	8.26	9.14	331.88
Jitter (ms)	13,707	6.79	13.85	0.00	1.13	1.74	4.38	116.02
Interarrival (s)	13,742	2.00	55.92	0.50	0.51	0.51	0.51	4,988.06
Packets sent	54,984	1,718.4	1,789.6	0	475.8	1,225	1,756	7,154
Packets received	54,984	1,711.9	1,789.7	0	470	1,219	1,750	7,148
Loss rate	13,746	0.008	0.044	0	0.001	0.001	0.006	1.000
Inst. Throughput (bps)	54,984	16,174.6	29,030	0	0	0	7,044.6	165,001.6
Avg. Throughput (bps)	54,984	51,147.3	11,165.3	0	47,017.8	49,127.9	55,995.6	138,378.4
PF metric	54,984	0.62	19.07	0	0	0	0.17	1,000

3) Per-station profile (round averaged): Table 5.3 summarises the round-averaged per-station performance under the PF scheduler, reporting mean delay, jitter, instantaneous and average throughput, the PF metric, and each station's share of selected rounds. Overall, 192.168.1.140 was prioritised most frequently (52.0%), while the remaining stations received a smaller and more evenly distributed selection share.

Table 5.3: Round-Averaged Performance by WLAN Station with PF Scheduler.

Station (IP)	Delay (ms)	Jitter (ms)	Int.Thrp (bps)	Avg.thrp (bps)	PF metric	Selected rounds (%)
192.168.1.140	10.46	4.81	31,932	55,608	0.665	52.0

192.168.1.141	10.97	7.08	8,796	49,523	0.208	12.3
192.168.1.142	12.87	13.76	10,023	47,004	0.141	13.4
192.168.1.143	11.56	7.08	13,948	52,454	1.463	22.3

4) Statistical insights

a) Correlation: Instantaneous and average throughput are strongly correlated ($\rho \approx 0.70$), whereas the PF metric reveals negligible correlation with either throughput ($\rho \approx 0.01$) or delay ($\rho \approx 0.02$), confirming its role as an independent scheduling weight.

b) Dispersion: The right skew in inter-arrival times ($\max \approx 5,000$ s) stems from idle periods in non-selected devices; a log-transform mitigates heavy tails for modelling.

c) Missing values: Delay, jitter and loss-rate observations are present in ≈ 25 % of rows, aligned with times when probes were received; zero-imputation would bias variance, thus our model pipeline employs an indicator feature (*is_selected*). This feature flags whether a device was the target of a probe in a given round, ensuring that the absence of a value is understood as a “not applicable” state rather than an error. This approach avoids biased techniques such as zero imputation and enables robust modelling.

5) Suitability for classical ML (delay-prediction regression): The dataset is well-suited for traditional tabular learners, such as a supervised regression problem with *delay_ms* as a continuous response. The feature space – comprising instantaneous and average throughput, jitter, inter-arrival time, loss rate, PF metric, binary selection flag, and one-hot-encoded station identifiers – totals fewer than 20 dimensions, allowing efficient model fitting without the curse of dimensionality. Missing values should be imputed using preprocessing techniques. After variance stabilising transforms on heavy-tailed predictors, ML models can be tuned via nested k-fold cross-validation; the sample size ($\approx 55k$) ensures robust error estimates and guards against overfitting. Feature importance analysis enables interpretability and quantitative attribution of delay to scheduler metrics and traffic conditions while supporting rigorous model diagnostics.

6) Suitability for DL: DL architectures can exploit the strong temporal ordering of rounds (≈ 14 Hz sampling) to capture cross-round dynamics. A sequence-to-label framing – using LSTM or Transformer encoders over a fixed-length window or per-device metric – could predict forthcoming PF selections, throughput bursts, or latency. Although the corpus’s temporal depth is ample ($\approx 13k$ steps), the width (four devices) remains small. If needed, one can use augmentation techniques, such as time-warping or GAN-based traffic synthesis, to mitigate overfitting. Learning can be further regularised by multitasking objectives, such as simultaneously predicting selection and delay. Furthermore, graph neural networks that treat stations as nodes and airtime as edges may illuminate fairness interactions, yet will require integrating exogenous PHY-layer context to generalise beyond this controlled WLAN scenario.

To summarise, the PF-WLAN dataset provides a statistically rich, moderate-sized corpus with a clean tabular structure and temporal coherence, making it equally amenable to rigorous classical modelling and more expressive deep-sequence architectures, provided that missing values are handled adequately and, if needed, temporal augmentation is conscientiously addressed.

5.2.1.2 Performance Metrics for PF Scheduler Experiment

This subsection discusses the performance metrics utilised to evaluate the PF scheduler experiment. We present and discuss statistical outcomes derived from experimental data, emphasising the performance characteristics exhibited by individual network devices. Additionally, we conduct a detailed analysis of an EGPR model, utilising the dataset collected during the PF scheduling experiment. This analysis offers insights into the model's efficiency, reliability, and predictive accuracy within the defined experimental framework.

1) Statistical Analysis of PF Scheduler: The experimental results under the Proportional Fair scheduler yielded the following statistics:

Table 5.4: Final Experiment Summary under Proportional Fair Scheduling.

Metric	Device 1 (192.168.1.140)	Device 2 (192.168.1.141)	Device 3 (192.168.1.142)	Device 4 (192.168.1.143)
Packets (Tx/Rx)	7,148 / 7,154 (99.9%)	1,677 / 1,687 (99.4%)	1,832 / 1,839 (99.6%)	3,050 / 3,066 (99.5%)
Average Throughput (bps)	51,066.45	47,017.76	40,672.92	68,612.19
Final PF Metric	0.0000	0.0000	0.0000	1.3431
Average delay (ms)	7.66	8.55	14.33	8.28
Minimum delay (ms)	6.41	6.44	5.78	5.56
Maximum delay (ms)	9.10	11.25	27.57	11.16

Table 5.4 summarises the results of a 7,200-second PF scheduling experiment conducted in a live WLAN environment with four devices. The results reveal near-perfect packet delivery rates (over 99%), with Device 4 achieving the highest average throughput (68.6 kbps) and a final value metric of 1.3431. The other devices ended with a final PF metric near 0.0, reflecting lower instantaneous-to-average throughput ratios at the end of the experiment. The delay statistics indicate overall low latency across devices, with Device 3 occasionally experiencing a higher maximum delay (27.57 ms).

a) Device 1 (192.168.140): Device 1 exhibited a high level of network activity, as evidenced by the substantial number of packets transmitted (7,154) and received (7,148) over the experimental duration. The observed average throughput was 51,066.45 bps, indicating a sustained, moderate data transfer rate. However, the terminal Proportional Fair (PF) metric of 0.0000 and the high packet count indicate a consistently high throughput over most of the experiment. This consistency likely led to the average throughput converging to the instantaneous throughput, thereby minimising the transmission ratio. The low average delay (7.66 ms) and relatively narrow delay range (6.41 ms to 9.10 ms) further support the inference of stable and favourable network conditions, potentially indicating a strong signal and minimal interference.

b) Device 2 (192.168.1.141): Device 2 exhibits a significantly lower packet exchange volume compared to Device 1, with 1,687 packets sent and 1,677 received. Despite this reduced activity, the average throughput remained relatively high at 47,017.76 bps. Similar to Device 1, the final PF metric was 0.0000. This, combined with lower packet volume, suggests two possibilities that are not mutually exclusive. Device 2 may have ceased data transmission earlier in the experiment, allowing the average throughput to dominate the PF calculation. Alternatively, similar to Device 1, it may have experienced stable network conditions, driving the average throughput close to its instantaneous value and minimising the PF metric near the end of the experiment. The low average delay (8.55 ms) reinforces the possibility of favourable network conditions, although the slightly wider delay range (6.44 ms to 11.25 ms) suggests a somewhat higher degree of variability than Device 1.

c) Device 3 (192.168.1.142): Device 3 exhibited the lowest average throughput (40,672.92 bps) among the four devices and a relatively low packet exchange count (1,839 sent and 1,832 received). In this context, the final PF metric of 0.0000 likely arises from consistently low but steady throughput. This device experienced the highest average delay (14.33 ms) and the widest delay range (5.78 ms to 27.57 ms). This substantial delay variation strongly indicates unfavourable and fluctuating network conditions. These conditions could stem from factors such as greater susceptibility to interference or intermittent periods of channel congestion. The combination of lower throughput and higher variable delay suggests that Device 3 likely faced significant network impairments.

d) Device 4 (192.168.1.143): Device 4 presented a distinct performance profile. It achieved the highest average throughput (68,612.19 bps) among all devices, with a moderate packet count (3,066 sent and 3,050 received). The defining characteristic of Device 4's performance was its non-zero final PF metric of 1.3431. This positive value indicates that, when the experiment was about to terminate, the instantaneous throughput was significantly higher than the recent average. This suggests that Device 4 experienced improved network conditions towards the end of the experiment, causing the Proportional Fair scheduler to prioritise its transmissions. The relatively low average delay (8.28 ms) and a delay range comparable to Device 2 (5.56 ms to 11.16 ms) suggest that, while Device 4 generally experienced good network conditions, it also experienced periods of

reduced performance, reflected in the lower average throughput leading up to the final measurement.

2) EGPR Model for PF Dataset: The evaluation of the network performance of four devices under Proportional Fair (PF) scheduling algorithms generated a dataset containing key metrics: delay, jitter, instantaneous throughput, average throughput, and packet loss. This dataset was used to train an EGPR model, chosen for its ability to model non-linear relationships and provide probabilistic predictions. The EGPR model facilitates a detailed analysis and visualisation of the relationships between delay and other performance metrics.

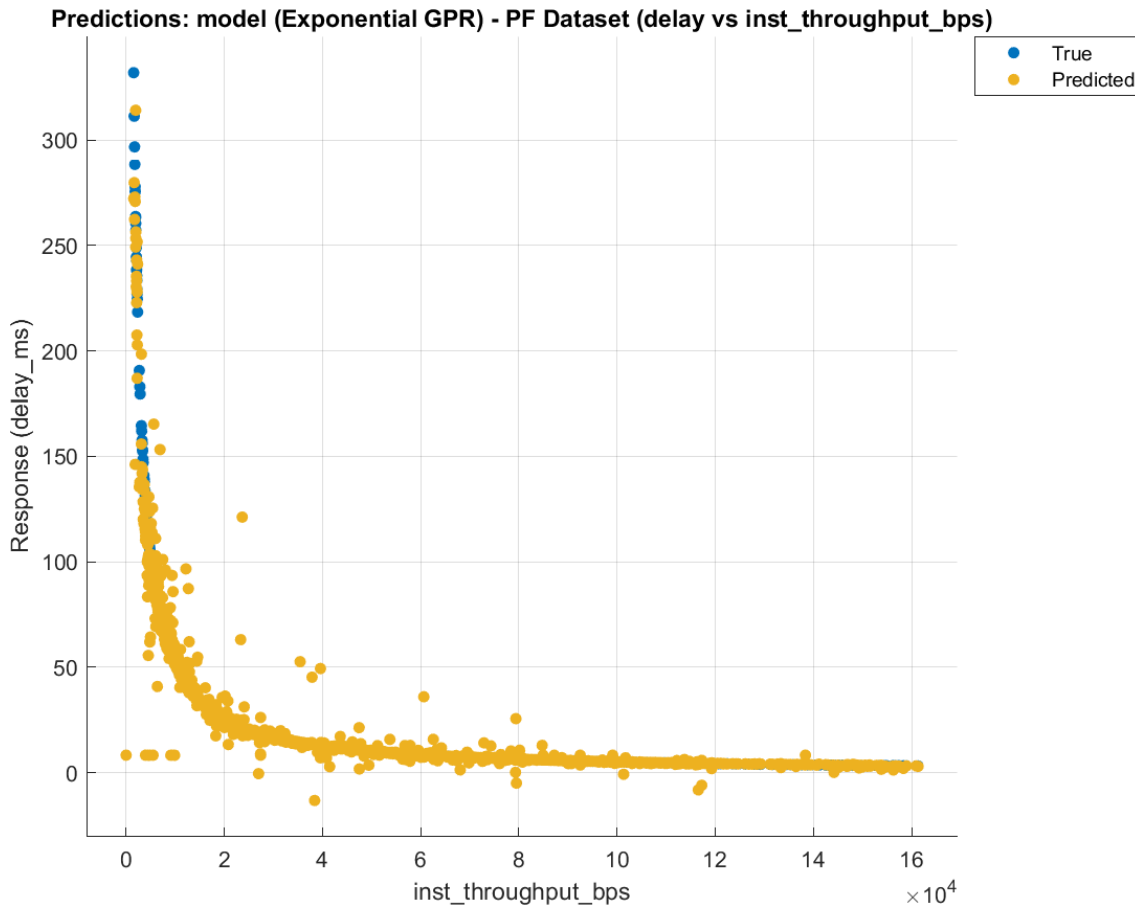


Figure 5.1: Delay prediction using the EGPR model under PF scheduling. The close overlap between predicted and true delays across throughput values indicates accurate delay-throughput modelling, with only a few low-throughput delays slightly underestimated.

Figure 5.1 illustrates the performance of the EGPR model for predicting delay based on instantaneous throughput in a proportional fair scheduler. The scatter plot shows the instantaneous throughput (x-axis, in bps $\times 10^4$) against delay (y-axis, in ms). The “True” values (blue dots) represent the actual measured delay for a given instantaneous throughput, while the “Predicted” values (yellow dots) show the delay predicted by the EGPR model for the same throughput. The plot reveals a clear inverse relationship: as

instantaneous throughput increases, delay generally decreases. The model's predictions (yellow) closely follow the trend of the true values (blue), particularly for higher throughputs. However, there is increased scatter and some underestimation of delay at very low throughput values, where the true values exceed 150 ms.

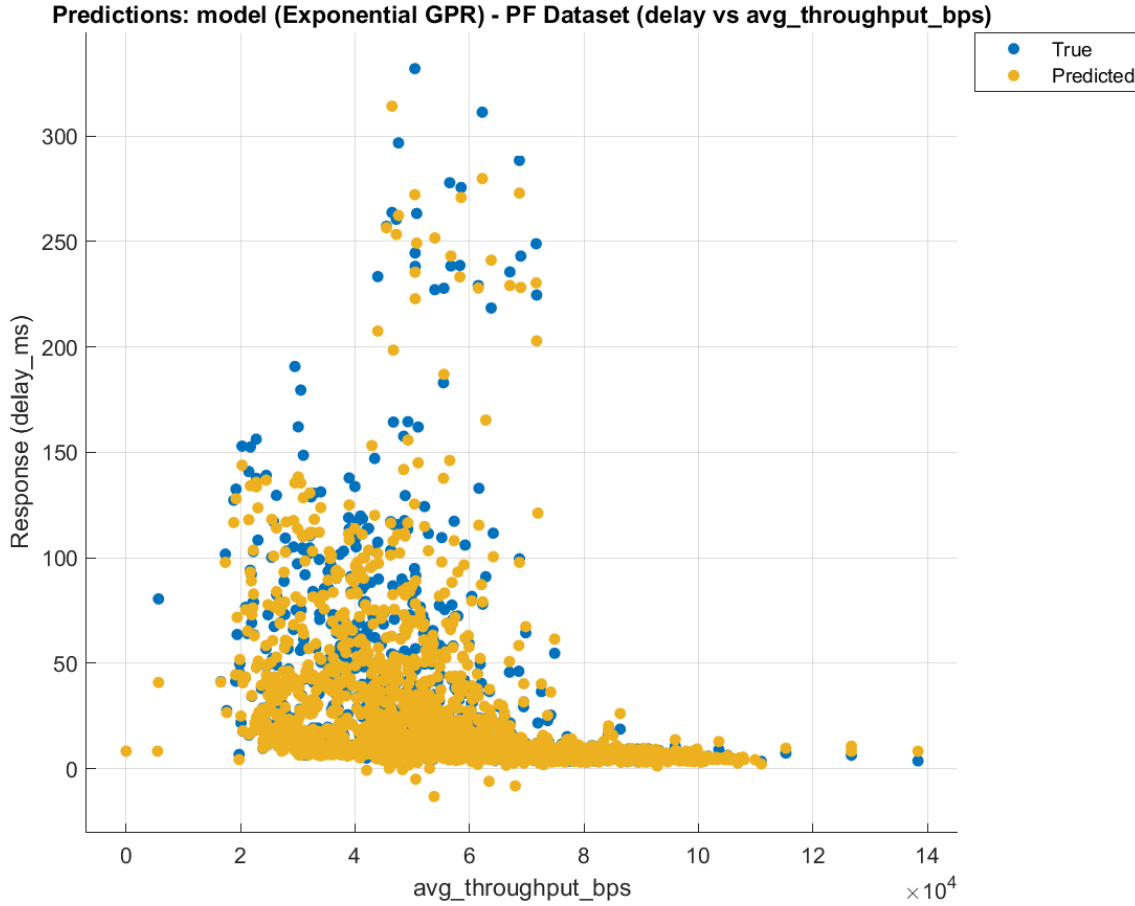


Figure 5.2: EGPR model performance for delay prediction versus average throughput under PF scheduling. The weak correlation and large scatter indicate that average throughput is a poor predictor of delay, and the model struggles to capture the high-delay variability, especially at lower average throughputs.

Using PF scheduler experiment data, Figure 5.2 shows a scatter plot of the relationship between average throughput (x-axis, in bps $\times 10^4$) and delay (y-axis, in ms). Similar to Figure 5.1, it compares “True” values (blue dots), representing the measured delay at a given average throughput, with “Predicted” values (yellow dots), the delay predictions from the EGPR model. This plot shows a less-defined trend than the one in Figure 5.1. While there is a general tendency for delay to be lower at very high average throughputs, there is significant scatter and overlap between true and predicted values across the entire range of average throughputs. The model struggles to capture delay variability, particularly for average throughputs below 8×10^4 bps. A considerable number of true values exceed 150 ms, which are not well predicted.



Figure 5.3: EGPR model performance for delay versus jitter under PF scheduling. Delay generally increases with jitter, confirming a positive association; however, the wide scatter – especially at low jitter – shows substantial variability that the model does not fully capture.

Figure 5.3 illustrates the relationship between jitter (x-axis, in milliseconds) and delay (y-axis, in milliseconds) using a scatter plot, derived from the Proportional Fair scheduler experiment data. As with the previous plots (Figures 5.1 and 5.2), the “True” values (blue dots) represent the measured delay for a given jitter, and the “Predicted” values (yellow dots) are the delay predictions from the EGPR model. The plot reveals a positive correlation, although with considerable scatter. As jitter increases, delay tends to grow as well. However, the relationship is not strictly linear; there is a large spread of delay values for any given jitter value, particularly at lower jitter values (below 40 ms). The model’s predictions (yellow) generally follow the trend of the true values, and the scatter suggests considerable variability that the model does not fully capture. There is a large concentration of data points in the lower jitter range.

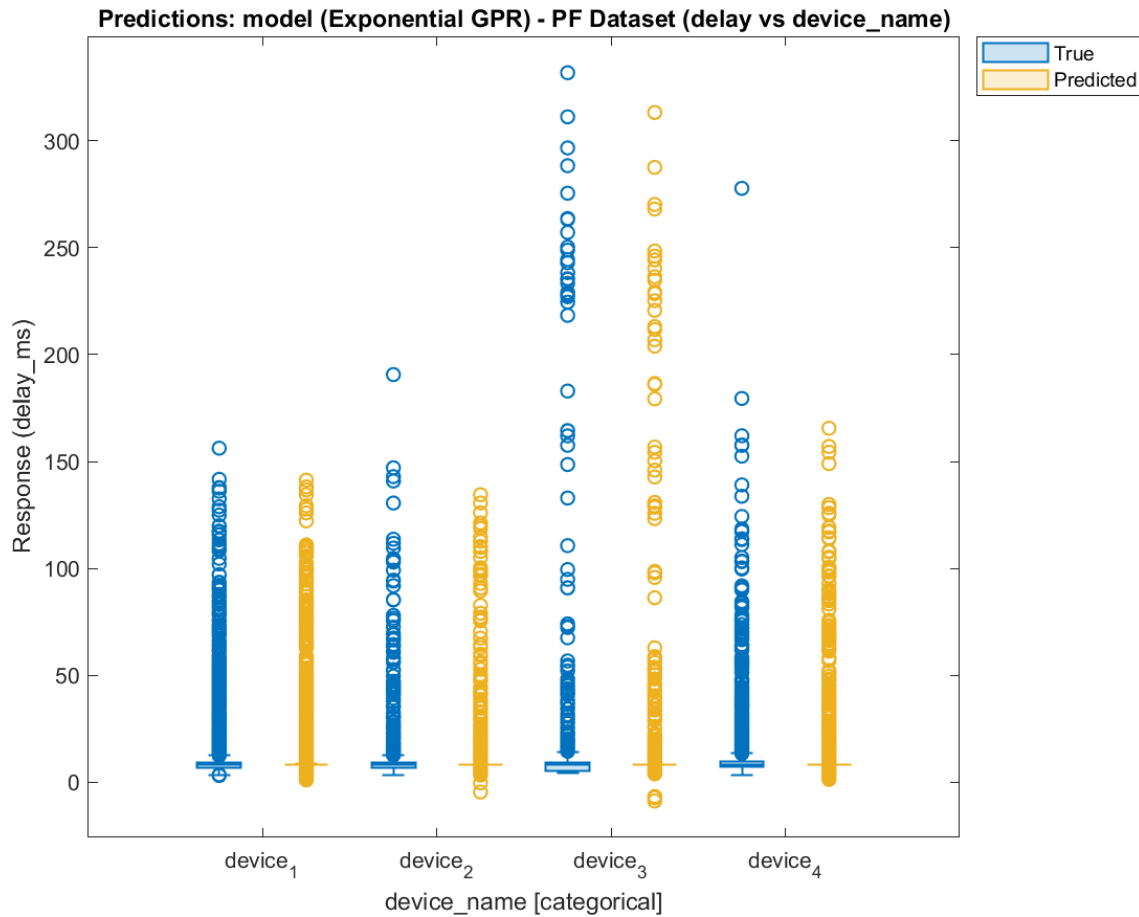


Figure 5.4: EGPR Model Performance – Delay vs Device in PF network. Predicted and true delays reveal that Device 3 has a markedly higher median delay and wider spread than Devices 1, 2, 3, and 4, consistent with the descriptive statistics in Table 5.4. The EGPR model correctly ranks devices by delay but slightly underestimates the extreme values for Device 3.

Figure 5.4 illustrates the EGPR model’s predicted and observed delay values categorised by each device. The scatter plot shows the variability in delay performance across devices, with Device 3 standing out, exhibiting a broader distribution of delay values that includes significantly higher peaks than the other devices. While the EGPR model effectively captures delay trends for Devices 1, 2, and 4 with closely aligned predicted and true values, it experiences slightly higher prediction variability for Device 3. This suggests that Device 3’s delay is more challenging to predict accurately, possibly due to factors such as increased network congestion or poorer link conditions, reinforcing earlier statistical findings. Overall, the model demonstrates a strong predictive capability, providing valuable insights into device-specific QoS characteristics.

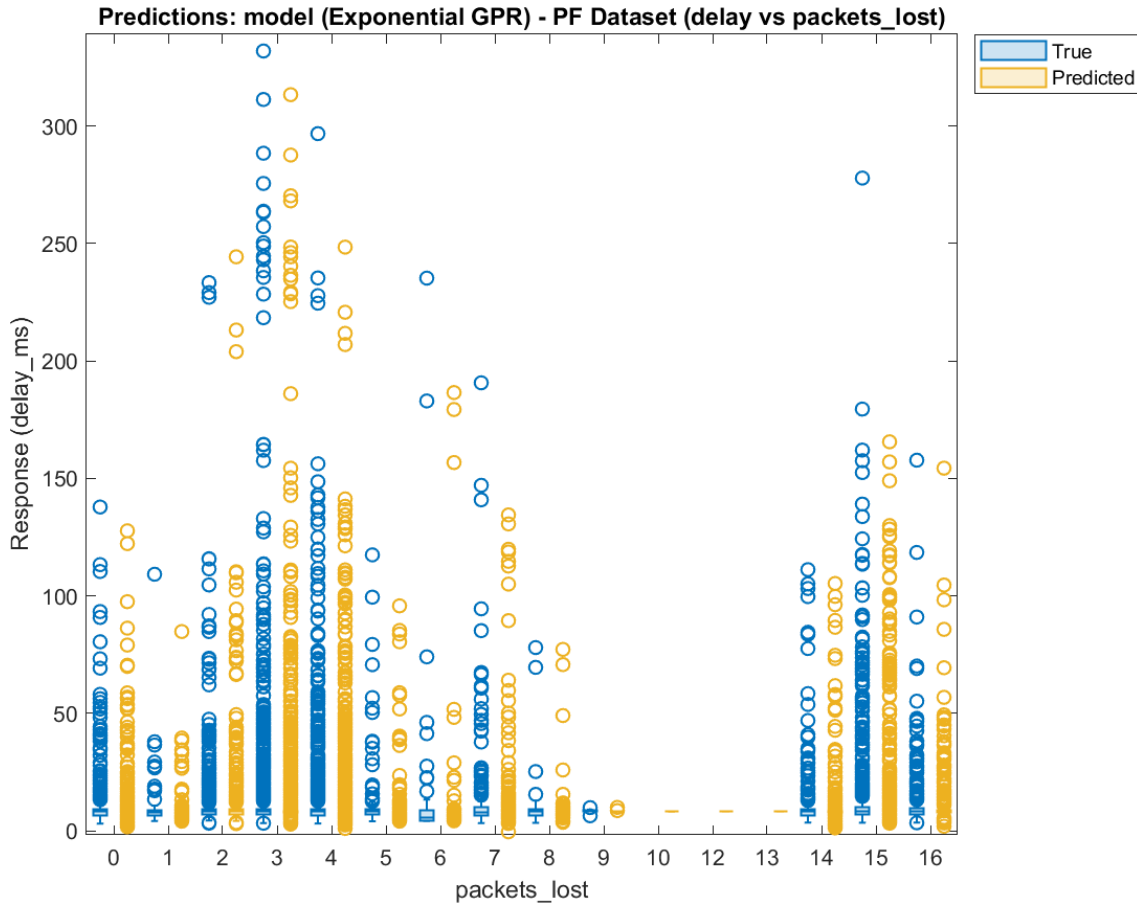


Figure 5.5: EGPR model performance – delay vs packet loss in a PF network. Delay shows a non-linear relationship with packet loss: some of the most significant delays occur when only 2-4 packets are lost, while very high loss (> 10 packets) often coincides with lower delays. The model broadly follows this pattern; however, the wide spread of points indicates that packet loss alone is an insufficient predictor of delay.

Figure 5.5 shows the relationship between packet loss and response delay in a PF network using an EGPR model applied to a PF scheduler dataset.

The x-axis represents the number of packets lost (packet-lost), while the y-axis shows the response delay in milliseconds. The data points are colour-coded: blue circles represent the true observed values, and yellow circles show the model's predictions.

Key observations from the plot:

- a)** The response delay varies significantly across different packet loss values, ranging from nearly 0 ms to around 330 ms.
- b)** There appears to be a non-linear relationship between packet loss and delay, with some of the highest delays occurring when packet loss is between 2 and 4 packets.

- c) When packet_lost is very high (>10), the delays tend to be lower, suggesting the model may be capturing some network behaviour where extreme packet loss triggers different handling mechanisms.
- d) The model's predictions (yellow) generally follow the distribution of the true values (blue), indicating reasonable model performance, though there are areas where the fit could be improved.
- e) The data shows considerable variance in delay for the same number of lost packets, suggesting other factors beyond packet loss also influence network delay. This visualisation plot helps evaluate how well the EGPR model captures the complex relationships between packet loss and delay in our experimental scenario.

5.2.1.3 Interpretation of Results for PF Scheduling Experiment

The experimental results obtained under the PF scheduling discipline consistently demonstrate a high packet delivery success ratio across all evaluated devices. Specifically, Device 1 (192.168.1.140) achieved the highest reliability, with a packet success rate of 99.9%, closely followed by Devices 3 (192.168.1.142) and 4 (192.168.1.143), which achieved rates of 99.6% and 99.5%, respectively. Device 2 (192.168.1.141) demonstrated slightly lower reliability at 99.4%, although it still represents robust transmission performance.

Regarding throughput, substantial variability was observed among the devices. Device 4 achieved the highest average throughput at approximately 68.6 kbps, significantly outperforming Devices 1, 2, and 3, whose throughput ranged from 40.6 kbps (Device 3) to 51.0 kbps (Device 1). These discrepancies highlight the PF scheduler's resource allocation strategy, which prioritises links based on link conditions and historical throughput performance.

The final PF metric further elucidates the scheduler's decision-making dynamics. Devices 1 through 3 concluded with a PF metric of 0, indicating minimal prioritisation at the end of the experiment. In contrast, Device 4 showed a notably elevated PF metric of 1.3431, indicating that the scheduler perceived it as having an optimal balance between channel conditions and past throughput, thereby allocating comparatively greater resources during scheduling intervals.

Latency analysis revealed a generally acceptable average delay across Devices 1, 2, and 4, ranging from 7.66 ms to 8.55 ms. In contrast, Device 3 experienced a notably higher average latency of 14.33 ms, coupled with a significant maximum delay spike of 27.57 ms. This disparity suggests potential issues related to suboptimal radio link quality, network congestion, or scheduling decisions that disproportionately affect this device.

In summary, the results underscore the efficacy of the PF scheduler in maintaining high packet reliability while dynamically optimising throughput in response to real-time conditions. Nevertheless, the observed latency anomalies for Device 3 necessitate further investigation into underlying factors, potentially including signal integrity, network interference, or scheduler parameter tuning. Future work should focus on adjusting scheduler configurations or network infrastructure to mitigate performance variations and ensure uniform QoS across all devices.

1) Interpretation of EGPR Model Results (PF Experiment): Figure 5.6 illustrates the MATLAB Regression Learner interface displaying the results obtained from the EGPR model trained on the collected network dataset. The dataset comprises QoS metrics derived from network traffic under a Proportional Fair (PF) scheduling scheme. As shown, the EGPR model achieved high predictive accuracy, with a validation R-squared of 0.96457 and an RMSE of 3.2002 ms.

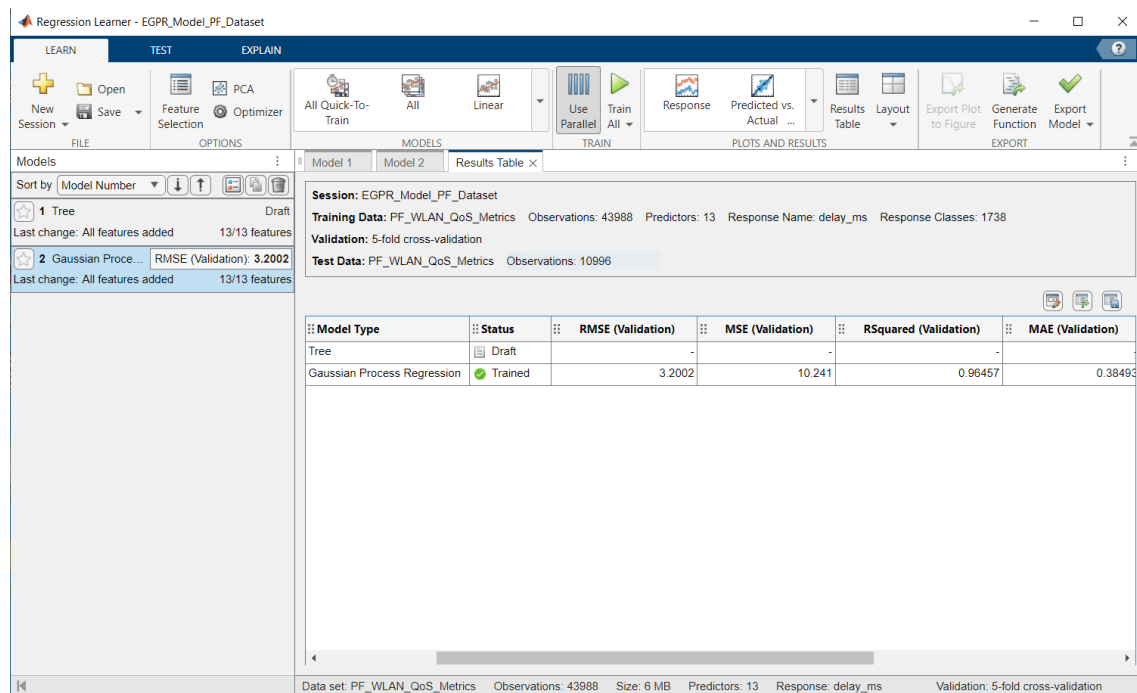


Figure 5.6: EGPR model performance on the PF dataset. The combination of low RMSE/MAE and high R^2 indicates that the model fits the PF delay data well, capturing the main delay-throughput relationships with relatively small prediction errors.

a) RMSE (Validation): 3.2002: The RMSE value of 3.2002 ms indicates that, on average, predictions made by the EGPR model deviate from actual delay measurements by approximately 3.2 milliseconds. This relatively low RMSE value suggests that the model provides precise predictions with minimal error, underscoring its reliability for delay estimation in QoS analysis.

b) MSE (Validation): 10.241: The MSE of 10.241 ms^2 represents the average squared difference between the predicted and observed delay values. Given its squared nature,

larger errors are disproportionately penalised, underscoring that the model exhibits relatively modest overall prediction variance. This metric complements the RMSE and further indicates stable predictive performance.

c) R-squared (Validation): 0.96457: An R-squared value of 0.96457 denotes that approximately 96.46% of the variance in delay observations can be explained by the EGPR model. This high-level value confirms that the model captures most of the variability in the network QoS data, indicating excellent goodness-of-fit and robust predictive capability.

d) MAE (Validation): 0.38493: The MAE of 0.38493 ms highlights the model's strong predictive accuracy by quantifying the average absolute difference between predicted and actual delay values. A notably low MAE reinforces the notion that individual prediction errors are typically minor, demonstrating that the model consistently produces precise predictions suitable for practical QoS metric forecasting tasks in networking contexts.

e) Connecting EGPR Model Results to Statistical Findings:

- **Overall Model Fit:** The EGPR model exhibits strong predictive performance, accurately modelling delay as evidenced by the close alignment between true and predicted responses across all plots. The high validation R-squared (0.96457) and low RMSE (3.2002 ms) confirm the excellent overall model fit.
- **13 Predictors:** The model utilises 13 predictors, indicating a comprehensive input space that effectively captures delay variations and yields accurate predictions.
- **Instantaneous vs. Average Throughput (Inferred):** The instantaneous throughput plot shows a more precise, more defined inverse relationship with delay compared to the average throughput, suggesting that instantaneous throughput is a stronger predictive factor for delay in this scenario. Average throughput shows a less distinct correlation, indicating a weaker predictive contribution than instantaneous throughput.
- **Jitter's Contribution (Inferred):** The scatter plot illustrating delay against jitter reveals a clear positive correlation, demonstrating jitter as a critical contributor to delay predictions. Higher jitter values are associated with increased delays, confirming their relevance as predictors in the EGPR model.
- **5-fold Cross-Validation:** The predictive accuracy indicated by the scatter plots in (Figures 5.1-5.3) was further validated through the robust 5-fold cross-validation approach employed. This cross-validation reinforces the model's generalisation ability, implying that its predictive quality is reliable and not limited to a specific subset of data.

The strong predictive performance demonstrated by the EGPR model, evidenced by validation metrics such as low RMSE (3.2002 ms) and R-squared value (0.96457), aligns closely with the empirical statistical results previously acquired under the Proportional Fair scheduler. Specifically, the model's ability to precisely estimate delay is particularly

valuable given the variations in delay observed among network devices, such as Device 3, which showed significantly higher delays (an average of 14.33 ms) compared to Devices 1, 2, and 4 (approximately 7.66 to 8.55 ms). The low prediction errors, as indicated by the model's MAE of 0.38493 ms, suggest that it accurately captures these variations, making it suitable for proactive delay prediction and QoS assurance. In essence, the EGPR model's effectiveness in predicting QoS metrics strongly complements the empirical findings, providing network operators with a practical tool to anticipate and manage the variability in network delay and throughput.

5.2.2 Round Robin Dataset Results

This section presents the results of the RR scheduler experiment conducted within the same experimental framework as the Proportional Fair algorithm. The primary focus is on evaluating the RR scheduler's performance and contrasting it with the PF algorithm. Data from this experiment, similar to the PF experiment, served as input for ML models, facilitating a comparative analysis of their predictive capabilities regarding network performance.

5.2.2.1 Summary of Round Robin Dataset

This sub-section analyses a two-hour, device round-trace of an IEEE 802.11ac WLAN operating under a strict Round-Robin (RR) scheduler. The trace comprises 55,216 per-station rounds across four client machines (192.168.1.140-143), each row encapsulating timestamp, instantaneous and average throughput, packet counters, loss metrics, inter-arrival time, and the target variable '*delay ms*', in conjunction with the binary '*is_selected*' flag that round robin sets deterministically every fourth slot. Because RR guarantees equal airtime, the dataset provides a fairness-controlled baseline against which the Proportional Fair (PF) results can be compared, while still offering sufficient temporal depth and traffic variability to support regression modelling of delay and its dependence on workload and channel conditions.

1) Dataset overview: Table 5.5 provides a high-level overview of the RR scheduling dataset, summarising the capture context (test duration, WLAN medium, stations) and the overall scale in terms of rounds and device-level observations. It also lists the main feature set and highlights the expected missingness patterns in delay and jitter, as probe responses are recorded only when an echo is received.

Table 5.5: Round Robin Scheduling Dataset Overview.

Characteristics	Value
Total observations	55 216 device-round records ($\approx 13,804$ rounds x 4 stations)
Capture window	2 hours
Medium and algorithm	IEEE 802.11ac WLAN, Round-Robin (RR) scheduler

Stations (IPs)	192.168.1.140-143
Features (14)	Timestamp station ID, round_ID, delay_ms, jitter_ms, inter_arrival_s, packets_sent/received/lost, loss_rate, instantaneous and average throughput, binary is_selected flag
Missing values	Delay-jitter present in 25% of rows (only when the probe echoed)

2) Descriptive statistics (packet-level): Table 5.6 summarises the packet-level descriptive statistics for the RR dataset, reporting central tendency and dispersion (mean, standard deviation, quartiles, and extrema) across the key continuous features. This provides a compact baseline view of delay, jitter, interarrival behaviour, loss, and throughput variability under RR scheduling.

Table 5.6: Summary of Continuous Features for the RR Dataset.

Feature	Count (obs)	Mean	Std	Min	Q1	Median	Q3	Max
Delay (ms)	13,782	11.90	24.63	3.140	6.61	7.68	9.15	367.63
Jitter (ms)	13,782	11.36	21.14	0.000	1.47	2.15	7.72	132.46
Interarrival time (s)	13,800	2.02	55.95	0.500	0.51	0.51	0.51	5,002.50
Packets sent	55,216	17,363	17,901.0	0.000	4,444.0	12,697	17,486	72,330
Packets received	55,216	17,319.3	17,900.7	0.000	4,420.0	12,666	17,457	72,304
Loss rate	13,804	0.009	0.045	0.000	0.001	0.001	0.006	1.000
Inst. Throughput (bps)	55,216	67,509	24,028	0.000	55,871	66,554	77,435	163,109
Avg. Throughput (bps)	55,216	67,489	11,735	0.000	60,194	65,572	72,811	111,832

3) Per-station profile (round averages): Table 5.7 summarises the round-averaged per-station performance under the RR scheduling, reporting delay, jitter, instantaneous and average throughput, selection share, and delivery ratio for each WLAN endpoint. This provides a compact comparison of latency, fairness, and reliability across the four stations.

Table 5.7: Round-Averaged Performance by WLAN Station under RR Scheduling.

Station (IP)	Delay (ms)	Jitter (ms)	Inst. Throughput (bps)	Avg. Throughput (bps)	Selected rounds (%)	Delivery ratio
192.168.1.140	9.56	4.77	65,014	65,012	25.0	0.9986
192.168.1.141	11.08	8.48	62,729	62,689	25.0	0.9942
192.168.1.142	16.87	26.43	79,368	79,358	25.0	0.9981
192.168.1.143	10.08	5.76	62,926	62,897	25.0	1.0000

4) Statistical insights

a) Fairness footprint: The RR scheduler guarantees uniform airtime (25% each), evident in identical ‘is_selected’ ratios.

b) Delay-throughput coupling: The packet-level correlations reveal a delay negatively associated with instantaneous throughput ($\rho \approx -0.50$), indicating congestion-induced queuing.

c) Heavy-tailed gaps: Inter-arrival shows strong right skew ($\max \approx 5,000$ s); log-transform is advisable for modelling.

d) Missing values: Values for delay, jitter, and loss only appear when probes return; models must treat missing values explicitly.

5) Suitability for ML (regression tasks): The RR dataset has a clean tabular structure with roughly a dozen engineered predictors once the four station identifiers are one-hot encoded, making it particularly amenable to traditional regression models. Before model induction, heavy-tailed variables – most notably ‘*inter-arrival_s*’ – should be established via logarithm transformation, while missing numeric fields such as ‘*delay*’ and ‘*jitter*’ (which are present only in probing rounds) require principled imputation. Standardising continuous inputs then allows regularised linear models, ensembles, decision trees and support vector machine models to learn effectively without undue sensitivity to scale. The sample size of approximately fifty-five thousand rounds supports nested cross-validation for hyperparameter tuning and affords statistically robust generalisation estimates. Explainability techniques can be readily applied to ML models to gain insights into the relative influence of throughput, selection state, and loss rate on end-to-end latency. This end-to-end latency is an essential requirement for WLAN QoS research.

6) Suitability for DL: Although moderate in volume, the RR dataset’s sequential ordering across 13,800 scheduler cycles provides fertile ground for temporal neural architectures. Framing the problem as sequence-to-one regression, gated recurrent units, LSTM networks, or lightweight Transformer encoders can ingest windows of consecutive rounds to capture longer-range dependencies between airtime allocation and subsequent delay. Because the dataset’s volume is not large by DL standards, careful regularisation – dropout, weight decay, and early stopping – is imperative, and performance can be further stabilised through modest data augmentation schemes such as time-warping or mix-up between adjacent windows. Multi-task learning, for example, by jointly predicting delay and the *is_selected* flag, introduces an auxiliary loss that can act as an additional form of regularisation. Finally, although the four-node topology limits the immediate benefit of graph neural networks, Transformer-style attention mechanisms can still discern station-specific temporal signatures, enabling the model to generalise delay dynamics beyond simple round counts.

5.2.2.2 Performance Metrics for RR Scheduler Experiment

This subsection outlines the evaluation metrics used to assess the RR scheduler experiment. We present statistical analyses derived from experimental observations, focusing on the operational performance of each participating device. We examine the metrics associated with the EGPR model utilising data collected throughout the RR

scheduling experiment. This comprehensive evaluation highlights the model's capability, consistency, and accuracy in predicting network behaviour under the RR scheduling conditions.

1) Statistical Analysis of RR Scheduler: The experimental results under the RR scheduler yielded the following statistics, which are presented in Table 5.8:

Table 5.8: Final Experiment Summary under Round Robin Scheduling algorithm.

Metric	Device 1 (192.168.1.140)	Device 2 (192.168.1.141)	Device 3 (192.168.1.142)	Device 4 (192.168.1.143)
Packets (Tx/Rx)	3451 / 3448 (99.9%)	3451 / 3438 (99.6%)	3451 / 3445 (99.8%)	3451 / 3451 (100%)
Avg Throughput (bps)	66,043.92	83,714.19	74,505.82	88,100.76
Delay (Avg ms)	8.55	9.72	65.40	7.05
Delay (min ms)	6.70	3.57	4.33	3.29
Delay (max ms)	11.41	32.27	297.94	10.77

The following analysis examines the performance of four devices under the RR scheduling algorithm. Key metrics were measured over two hours, including packet delivery rate, throughput, and delay. The results highlight significant variations in performance across the devices:

a) Device 1 (192.168.1.140): Device 1 demonstrated high packet delivery reliability, with 99.9% of sent packets successfully received. Its average throughput was the lowest among the four devices, at 66043.92 bps. The delay performance was good, with an average delay of 8.55 ms and a relatively tight range between the minimum (6.70 ms) and maximum (11.41 ms) delays. This indicates consistent, although slower, performance.

b) Device 2 (192.168.1.141): Device 2 also showed strong packet delivery reliability (99.6%). It achieved a higher average throughput than Device 1, at 83714.19 bps. The average delay was slightly higher than Device 1 at 9.72 ms, and notably, the maximum delay experienced was considerably higher at 32.27 ms, suggesting some occasional periods of increased latency despite a low minimum of 3.57 ms.

c) Device 3 (192.168.1.142): Device 3 had a reasonable packet delivery rate of 99.8%. Its throughput was intermediate, at 74505.82 bps. The most significant characteristic of Device 3 was its significantly higher average delay of 65.40 ms and a very high maximum delay of 297.94 ms. This suggests that while the device reliably delivered packets and achieved reasonable throughput, it experienced substantial latency, which may make it unsuitable for delay-sensitive applications.

d) Device 4 (192.168.1.143): Device 4 exhibited perfect packet delivery, with a 100% success rate. It also achieved the highest average throughput of all devices, at 88,100.76 bps. Furthermore, it maintained a low average delay of 7.05 ms, with a tight range from the minimum (3.29 ms) to the maximum (10.77 ms). Device 4 provided the best performance, combining high reliability, high throughput, and low, consistent latency.

2) EGPR Model for RR Dataset: The experiment evaluated network performance of four devices under the RR scheduling algorithm, focusing on key metrics such as packet loss, throughput, delay, and jitter, and collected network data over 2 hours. This data was used to train an EGPR machine-learning model. This enabled the visualisation and analysis of relationships between delay and various performance indicators, including instantaneous throughput, average throughput, jitter, and per-device delay characteristics.

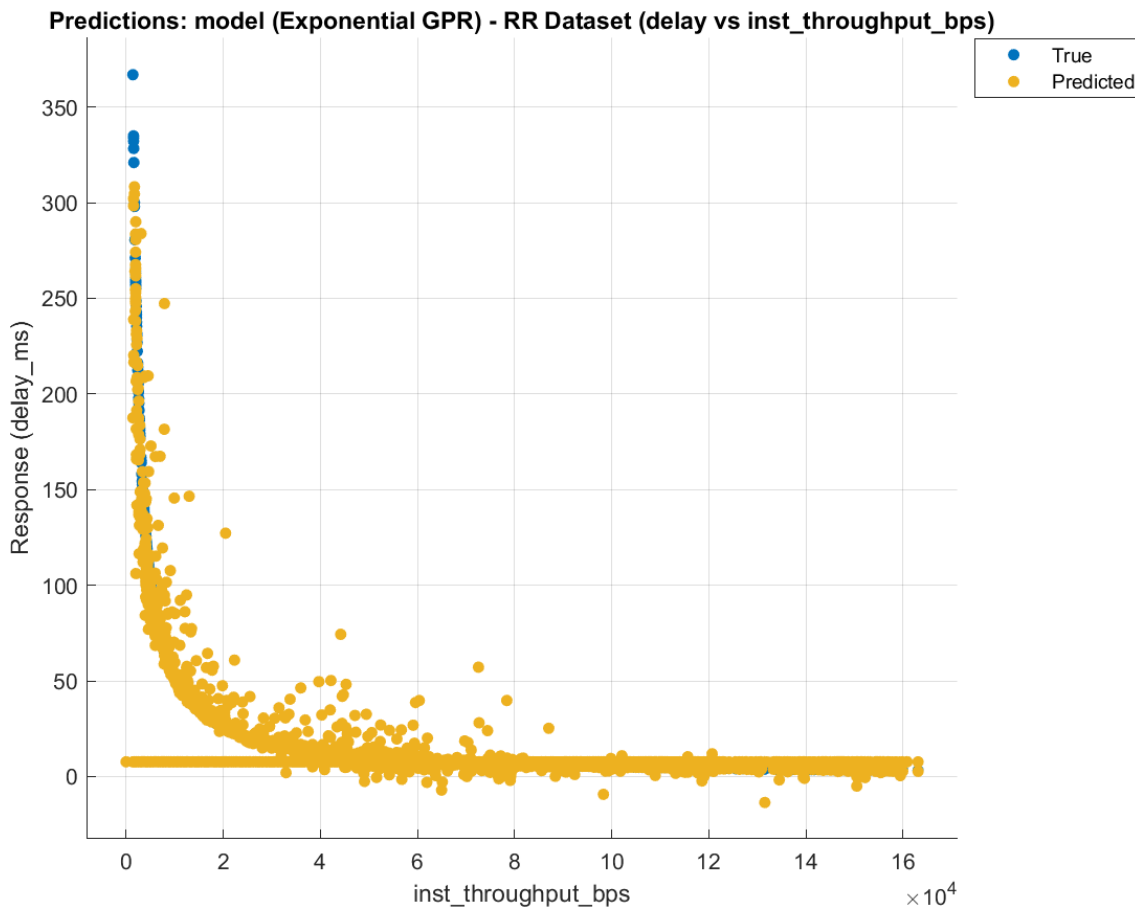


Figure 5.7: EGPR model performance – delay vs instantaneous throughput in a RR network. Delay decreases sharply as instantaneous throughput increases, and the EGPR predictions follow this inverse trend, confirming instantaneous throughput as a strong delay predictor under RR, with only some substantial delays at high throughputs slightly underestimated.

The scatter plot in Figure 5.7 visualises the relationship between instantaneous throughput (x-axis, in bps $\times 10^4$) and delay (y-axis, in ms) for the Round Robin (RR) scheduler experiment. As in previous plots, “True” values (blue dots) represent the actual measured

delay at a given instantaneous throughput, while “Predicted” values (yellow dots) show the delay predicted by the EGPR model. The plot reveals a clear inverse relationship: as instantaneous throughput increases, delay generally decreases. The EGPR model’s predictions (yellow) closely follow the trend of the true values (blue), particularly for higher throughputs. Nevertheless, there is increased scatter and some underestimation of delay at very high throughput values, similar to what was observed in the PF experiment (see Figure 5.1). Some true delay values exceed 250 ms.

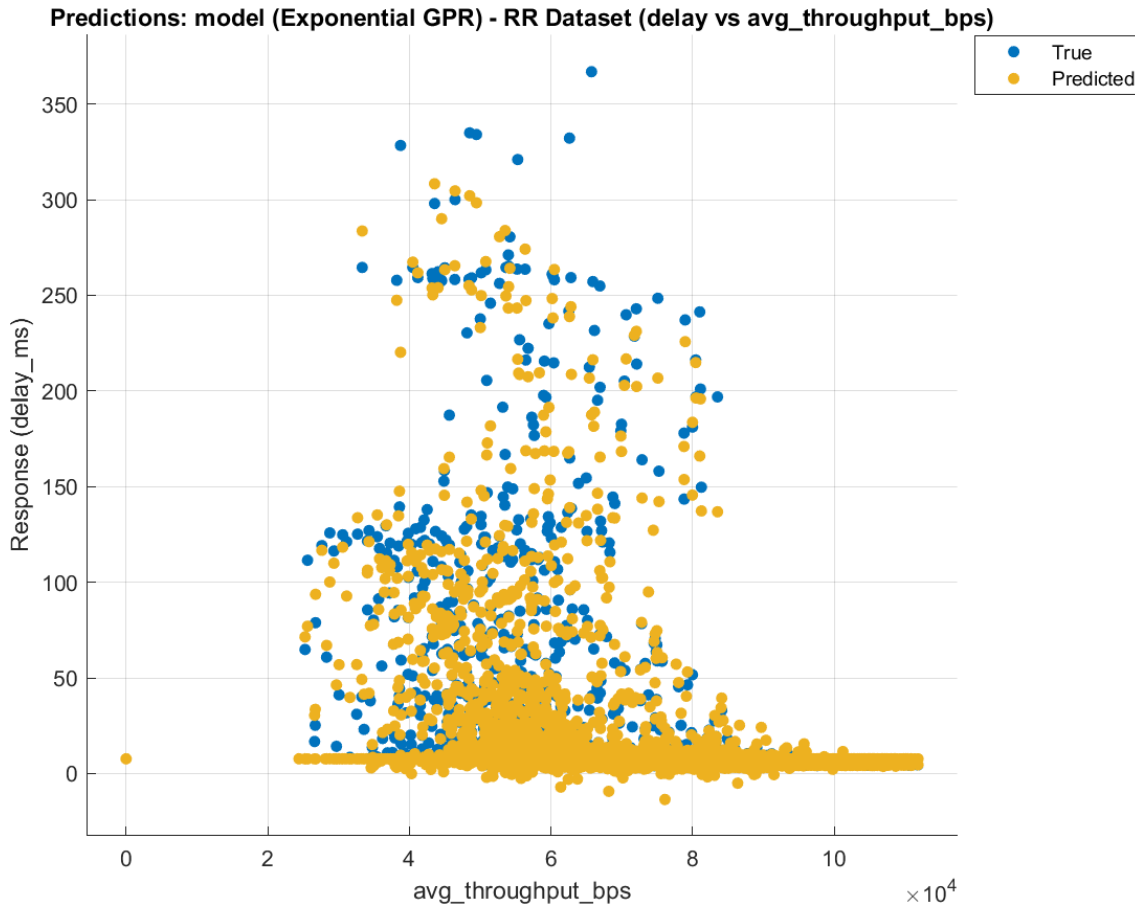


Figure 5.8: EGPR model performance – delay vs average throughput in a RR network. Average throughput shows only a very weak, highly scattered relationship with delay, and the EGPR model fails to capture many high-delay outliers – especially at lower average throughputs – confirming that average throughput is a poor predictor of delay in this scenario.

Figure 5.8 shows a scatter plot of the relationship between average throughput (x-axis, in bps $\times 10^4$) and delay (y-axis, in ms) for the RR scheduler experiment. The “True” values (blue dots) represent the measured delay at a given average throughput, while “Predicted” values (yellow dots) are the delay predictions from the EGPR model. Additionally, the plot shows a weak, poorly defined relationship between average throughput and delay. Furthermore, there is significant scatter and overlap between the true and predicted values across the entire range of average throughputs. The model struggles to capture any clear

trend, and the predictions are generally inaccurate. Many true values exceed 150 ms, and some even exceed 300 ms, particularly at lower average throughput.

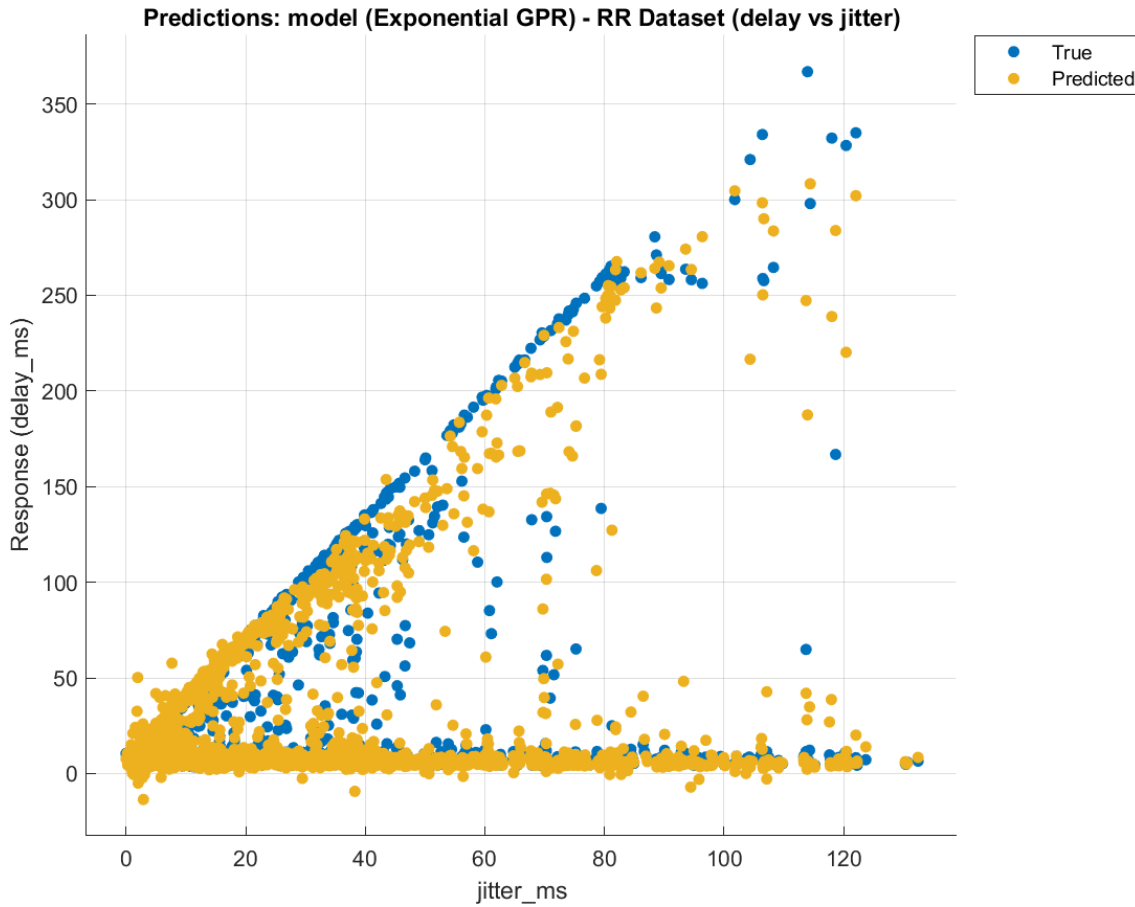


Figure 5.9: EGPR model performance – delay vs jitter in a RR network. Delay generally increases with jitter, confirming a positive association. However, the greater scatter (especially at low jitter) and systematic underestimation at high jitter indicate that the model captures the overall trend while failing to fully represent the variability of high-delay events.

The scatter plot in Figure 5.9 illustrates the relationship between jitter (x-axis, in ms) and delay (y-axis, in ms) for the data collected under the RR scheduler. The true values in blue dots represent the measured delay corresponding to a given jitter value, while the predicted values in yellow dots are the delay predictions made by the EGPR model. The plot reveals a positive correlation between jitter and delay: delay tends to increase as jitter increases. Nonetheless, the relationship is not strictly linear and shows considerable scatter, particularly at lower jitter values (below approximately 40 ms). The EGPR model's predictions generally follow the trend of the true values; however, there are noticeable discrepancies. The model tends to underestimate delay, particularly at higher jitter levels, and the scatter indicates significant variability that the model does not fully capture. There is a large concentration of data points in the lower jitter range.

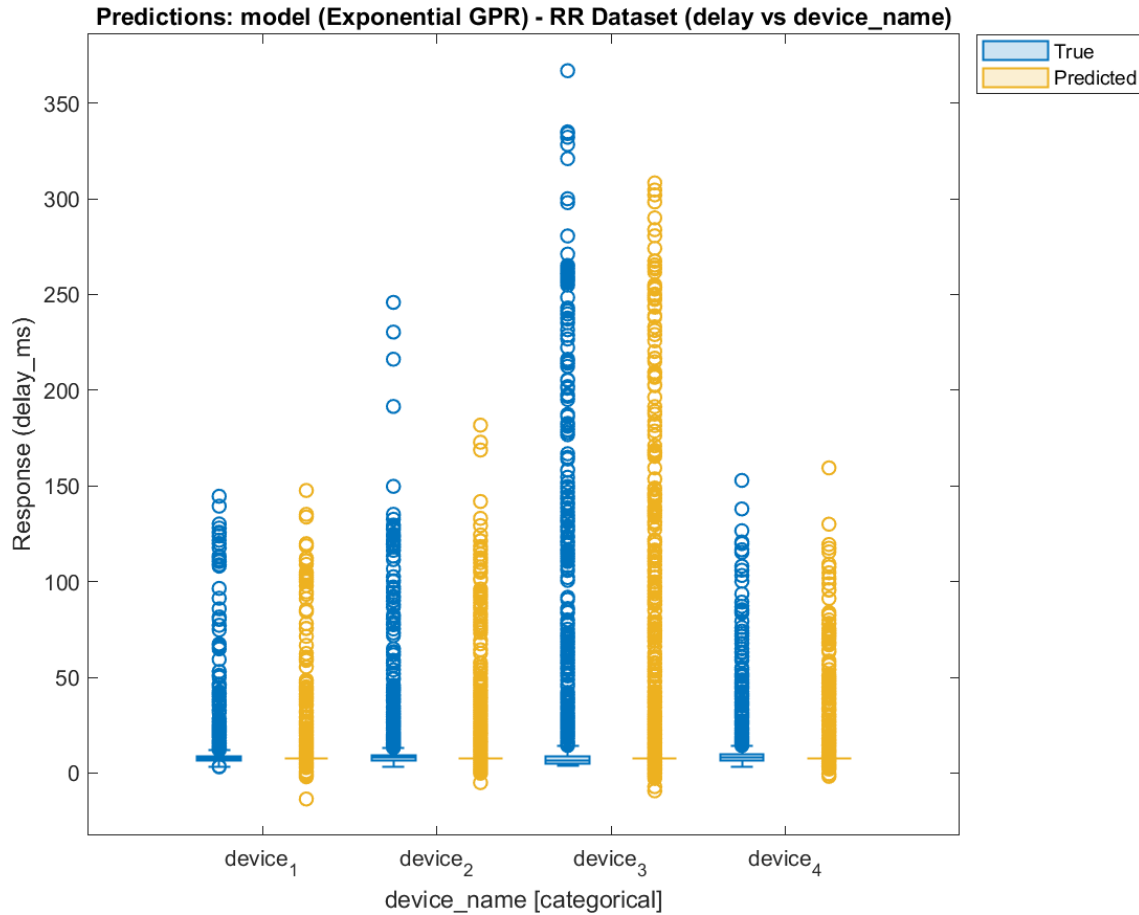


Figure 5.10: EGPR model performance – delay vs device in a RR network. The box plots show that Device 3 has a much higher median delay and wider spread than Devices 1, 2, and 4, and the model correctly identifies this worst-performing device, although it underestimates the full variability and extreme delay outliers.

Figure 5.10 depicts a box plot visualising the distribution of delay for each device. This plot further clarifies the model's performance and relationship to the observed device-specific behaviour. The plot shows the median, quartiles, and outliers for each device's delay, both for the true measured values (blue) and the EGPR model's predictions (yellow). Notably, device 3 exhibits a significantly higher median delay and much wider Interquartile Range (IQR) than the other devices, both in true and predicted values. This aligns perfectly with the statistical findings, which show that Device 3 had the highest average and maximum delay. The model successfully captures this device-specific difference in delay distribution. While the model generally predicts the median delay for Devices 1, 2, and 4 reasonably well, it underestimates the spread (IQR) and extreme values (outliers) for all devices, particularly Device 3. This suggests that while the model can identify which device is likely to experience higher delay, it struggles to fully capture the magnitude of the delay variations, especially under poor network conditions.

5.2.2.3 Interpretation of Round Robin Scheduler Experiment Results

By design, the RR scheduling algorithm provides equal time-slice allocation to each connected device, irrespective of their instantaneous or historical network conditions. The experimental results summarised in Table 5.8 reflect these fundamental characteristics. Notably, the packet counts across all four devices (Device 1 to Device 4) were remarkably similar (3451 packets sent), confirming the equitable distribution of transmission opportunities. However, significant variations in average throughput and delay were observed despite equal access time.

Device 3 exhibited the most distinct performance profile, characterised by a substantially higher average delay (65.40 ms) and a dramatically wider delay range (4.33 ms to 297.94 ms) compared to the other devices. This indicates that device 3 received its fair share of transmission time but consistently encountered unfavourable network conditions (or a hardware problem) during those intervals. Furthermore, these conditions could be attributed to susceptibility to interference or persistent channel congestion. On the other hand, Devices 1, 2, and 4 demonstrated lower and more consistent delays, suggesting more stable and favourable network environments. The variations in average throughput (ranging from 66,043.92 bps for Device 1 to 88,100.76 bps for Device 4) further underscore that RR scheduling guarantees equal access time; however, not equal performance, as the latter is inherently dependent on the fluctuating conditions experienced during each device's allocation time slice.

The relationship between network performance and delay was further investigated using an EGPR model. Three distinct relationships were examined: delay versus instantaneous throughput, delay versus average throughput, and delay versus jitter. These relationships are presented in Figures 5.7-5.9.

The delay plot versus instantaneous throughput (Figure 5.7) revealed a strong inverse relationship, consistent with fundamental networking principles. Higher instantaneous data rates correlate with lower latencies. The EGPR model accurately captured this trend, particularly at higher throughput values, demonstrating its capability to predict delay based on instantaneous throughput under RR scheduling. This finding aligns with results from the PF scheduler, suggesting that the instantaneous throughput-delay relationship is mainly independent of the scheduling algorithm.

In stark contrast, the delay plot versus average throughput (Figure 5.8) shows a weak and poorly defined relationship. The EGPR model struggles to fit the data, producing significant discrepancies between predicted and actual delay values. This reinforces the crucial point that average throughput, which can mask short-term fluctuations, is a poor predictor of delay, regardless of the scheduling discipline. This finding is consistent with the results from the PF experiment.

Furthermore, the plot of delay versus jitter (Figure 5.9) showed a positive correlation, as expected. Higher jitter (variability in packet arrival time) generally corresponds to higher overall delay. However, the relationship was not strictly linear, and the EGPR model revealed limitations in accurately predicting delay based solely on jitter, especially at higher jitter values. This suggests that while jitter contributes to delay, it is not the sole determining factor; other network conditions also play a significant role. This result is also consistent with the PF experiment.

1) Interpretation of EGPR Model Results (RR Experiment): The figure below (Figure 5.11) shows the results of the EGPR model on data from the RR scheduler experiment.

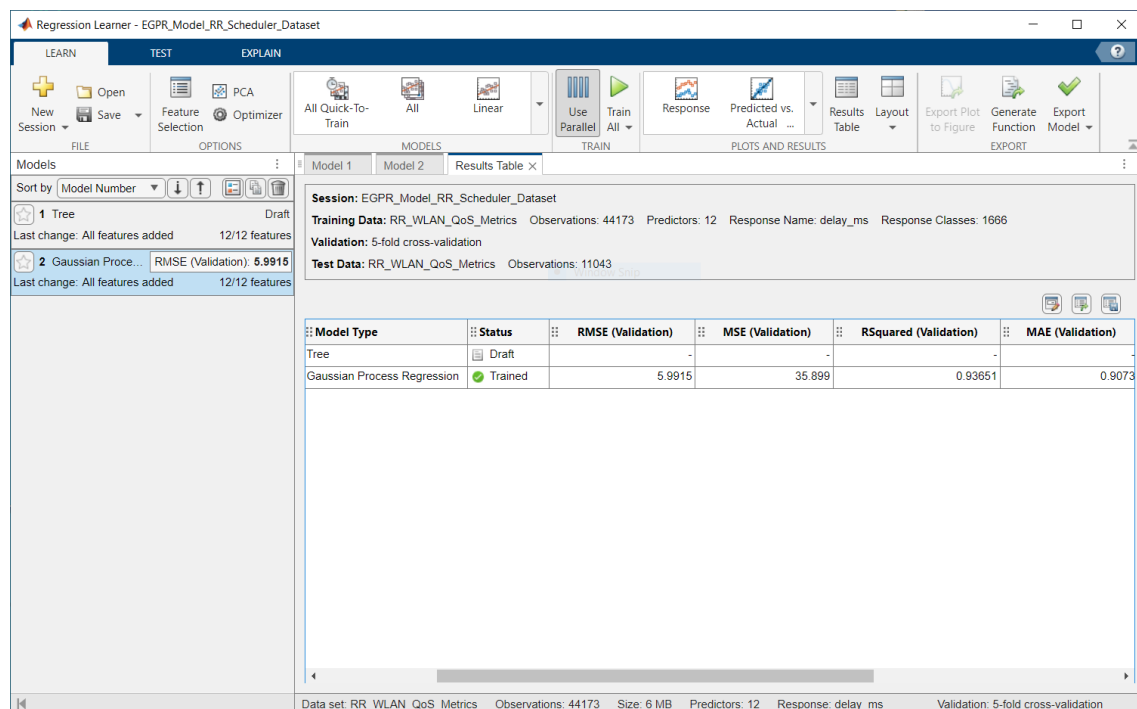


Figure 5.11: EGPR model performance on the RR dataset. The high validation R^2 (≈ 0.94) and low MAE (≈ 0.91 ms) show that the model captures most of the delay variability, while the RMSE (≈ 6 ms) indicates non-negligible errors on extremely high-delay events due to the wide delay range (up to ≈ 300 ms).

a) RMSE (Validation): 5.9915: The RMSE is a measure of the average magnitude of the errors between predicted and actual values. It is expressed in the same units as the response variable (delay, in milliseconds). An RMSE of 5.9915 ms means that, on average, the model's delay predictions are off by about 6 milliseconds. Whether this is a "good" or "bad" result depends on the context and the acceptable error tolerance. The wide range of delay values (up to nearly 300 ms) and RMSE of 6 ms suggest the model has predictive power; nonetheless, there is also significant room for error.

b) MSE (Validation): 35.899: MSE is the average of the squared errors. It is the square of RMSE. Thus, MSE gives more weight to larger errors. The MSE is less directly interpretable than the RMSE, but it helps compare different models (lower MSE is better).

c) R-Squared (Validation): 0.93651: R-squared (coefficient of determination) measures the proportion of the variance in the response variable (delay) that the model explains. It ranges from 0 to 1, with higher values indicating a better fit. An R-squared of 0.93651 is a good result, suggesting that the model explains about 93.7% of the variability in delay. This result seems to contradict the RMSE; however, it is essential to remember that the R-squared can be high even with a large RMSE if the response variable's range is also large.

d) MAE (Validation): 0.9073: MAE is the average of the absolute differences between predicted and actual values. It is also the same unit as the response variable (ms). An MAE of 0.9073 ms indicates that, on average, the model's predictions are off by less than 1 millisecond. This is a good result, and it is vital to consider it alongside the other results.

e) Connecting EGPR Model Results to Statistical Findings:

- **Overall Model Fit:** The high R-squared value indicates that the model captures a significant portion of the relationship between the input features (12 predictors) and the delay. Nonetheless, the RMSE of 5.9915 ms indicates that the average prediction error remains non-negligible. The low MAE might be somewhat misleading, given the RMSE.
- **12 Predictors:** The model used 12 predictors, which included instantaneous throughput, average throughput, jitter, packet loss, device name, and other vital network metrics. The high R-squared suggests that some combinations of these 12 predictors are useful for predicting delay.
- **Instantaneous vs Average Throughput (Inferred):** Based on the plots, instantaneous throughput is a strong predictor of delay, whereas average throughput is a weak predictor. Since the model has a reasonably good overall fit (high R-squared and moderate RMSE), the instantaneous throughput is likely a major contributor to the model's predictive power. The model is likely leveraging the strong inverse relationship between instantaneous throughput and delay.
- **Jitter's Contribution (Inferred):** Jitter also likely contributes to the model's performance to a lesser extent than instantaneous throughput. The plot of delay vs. jitter revealed a positive correlation, albeit with significant scatter. The model likely incorporates jitter; however, it is not as powerful a predictor as instantaneous throughput.
- **5-fold Cross-Validation:** 5-fold cross-validation involves training the model on 80% of the data and testing it on the remaining 20%. This process is repeated five times, using different subsets of the data each time. This process ensured that the model's performance was not overly optimistic due to overfitting to a specific training set. The reported metrics are the average across these five folds.

In summary, the EGPR model, using 12 predictor variables, achieves a reasonably good fit to the RR experiment data (high R-squared and moderate RMSE), primarily by leveraging the strong inverse relationship between instantaneous throughput and delay

and, to a lesser extent, the positive correlation between jitter and delay. The model also successfully captures the relative differences in delay distribution between devices, particularly the higher median delay and wider spread for Device 3, as visualised in the box plot (Figure 5.10). Nevertheless, the model tends to underestimate the magnitude of delay variations, especially for devices experiencing poor network conditions, as evidenced by smaller IQRs and fewer predicted outliers than the true values. The model's performance is consistent with the plots' statistical findings and visual analyses, highlighting its strengths and limitations in predicting delay in a dynamic wireless environment under RR scheduling. The model implicitly captures overall trends and device-specific differences, even though it does not explicitly model the RR scheduler's internal workings.

5.2.3 Comparison of PF and RR Results

This section provides a comparative analysis of the PF and RR scheduling schemes (presented in Table 5.9), drawing on statistical metrics derived from experimental network measurements and on predictive modelling using the EGPR model. The study emphasises key QoS parameters, such as delay, throughput, jitter, and packet transmission reliability, highlighting differences in scheduler performance. The EGPR model results further enrich the comparison by illustrating predictive accuracy and offering deeper insights into how each scheduler influences network behaviour and QoS outcomes.

Table 5.9: Comparative analysis of the PF and RR scheduling disciplines.

Metric	Device	PF Scheduler	RR Scheduler
Average Throughput (bps)	1	51,066.45	66,043.92
	2	47,017.76	83,714.19
	3	40,672.92	74,505.82
	4	68,612.19	88,100.76
Average Delay (ms)	1	7.66	8.55
	2	8.55	9.72
	3	14.33	65.40
	4	8.28	7.05
Minimum Delay (ms)	1	6.41	6.70
	2	6.44	3.57
	3	5.78	4.33
	4	5.56	3.29
Maximum Delay (ms)	1	9.10	11.41
	2	11.25	32.27
	3	27.57	297.94
	4	11.16	10.77
Packets Received	1	7,148	3,448
	2	1,677	3,438
	3	1,832	3,445

	4	3,050	3,451
Packets Sent	1	7,154	3,451
	2	1,687	3,451
	3	1,839	3,451
	4	3,066	3,451
Final PF Metric	1	0.0000	N/A
	2	0.0000	N/A
	3	0.0000	N/A
	4	1.3431	N/A

Analysis of Relative Performance:

1) Throughput Performance: The RR scheduler consistently delivers higher average throughput than the PF scheduler across all devices. For instance, Device 2's throughput increases from approximately 47,018 bps under PF to about 83,714 bps under RR. This result suggests that if maximising raw throughput is the primary objective, RR has a clear advantage by effectively maximising the amount of data transmitted across the network.

2) Delay Metrics: The performance differences in delay between the two schedulers are significant. Devices 1 and 2 show slightly lower average delays under PF, and Device 4 experiences a marginal benefit under RR. However, the most substantial difference is observed with Device 3. Under PF, Device 3 has an average delay of 14.33 ms, compared to 65.40 ms under RR. The maximum delay for Device 3 is also considerably lower with PF (27.57 ms) than with RR (297.94 ms). These results indicate that PF mitigates device delays more effectively under poor channel conditions by dynamically allocating resources based on channel quality.

3) Packet Distribution: The packet distribution also highlights the fundamental differences in scheduler design. RR ensures nearly equal packet counts for sent and received packets across all devices, with each device receiving roughly 3,450 packets. This uniform distribution underscores RR's commitment to strict fairness in time allocation. In contrast, PF demonstrates significant variability in packet counts; for example, Device 1 received 7,148 packets, while Device 2 received only 1,677, indicating that PF prioritises devices with better channel conditions at the expense of absolute fairness in packet distribution.

4) Final PF Metric: The final PF metric is reported exclusively for the PF scheduler, with Device 1-3 recording a value of 0 and Device 4 a value of 1.3431.

a) Device 4 (metric = 1.3431): Selected for transmission in the final round. Its metric was the highest, indicating it offered the best ratio of current channel quality to its historical average throughput at that moment.

b) Devices 1-3 (metric=0): A PF metric of zero occurs when a device's instantaneous throughput is zero. This means these devices were not candidates for scheduling because they either had no data to send/receive, or were inactive during this final round.

5.2.4 Statistical Significance of Validation Metrics (PF vs RR, EGPR (Experiment 1))

This subsection interprets the reported values of the EGPR validation metrics in light of the figures and tables in Section 5.2, focusing on what the numbers imply statistically for latency analysis and scheduler comparison.

1) EGPR validation performance, anchored to Figures 5.6 and 5.11: On the PF dataset, the EGPR achieves $R^2 = 0.96457$, $RMSE = 3.2002$ ms, and $MAE = 0.38493$ ms (see Figure 5.6), indicating that $\approx 96\%$ of latency variance is explained and that typical absolute error is sub-millisecond – statistically tight for delay estimation in this context. The RR dataset yields $R^2 = 0.93651$, $RMSE = 5.9915$ ms, and $MAE = 0.9073$ ms (see Figure 5.11). The larger RMSE-MAE gap under RR signals heavier-tailed residuals (i.e., more/larger outliers) than under PF, which is material for interpreting tail-latency risk. Both sets of metrics are reported under 5-fold cross-validation, so these values reflect averages across splits rather than a single favourable partition – strengthening the evidential weight of the comparison.

2) What the relationship plots imply about the effect sizes (PF: Figures 5.1-5.5; RR: Figures 5.7-5.10): The instantaneous throughput vs delay plots show a clear inverse coupling for both schedulers (PF Figure 5.1, RR Figure 5.7), with the predicted points closely tracking the empirical trend – evidence that a strong, scheduler-agnostic signal drives much of the explained variance reported above. On the contrary, the average throughput vs delay plots (PF Figure 5.2, RR Figure 5.8) show weak structure and high scatter, consistent with the models' residual variance and underscoring why average rate is a poor standalone predictor of latency. The jitter vs delay plots (PF Figure 5.3, RR Figure 5.9) reveal a positive association, but with broad dispersion; statistically, this aligns with a moderate contribution to explained variance and a residual spread that lifts RMSE relative to MAE – especially in RR.

3) Device-wise distributions reveal the source of heavier tails (PF: Figure 5.4; RR: Figure 5.9): The device-stratified delay views (PF Figure 5.4, RR Figure 5.10) make the tail behaviour concrete: Device 3 presents wider spread and extreme values, and this effect is amplified under RR. This visual evidence coheres with the validation metrics: PF's smaller RMSE and MAE reflect tighter distributions; RR's larger RMSE (vs a sub-ms MAE) reflects rare but much larger deviations.

4) Consistency with the comparative table (Table 5.9) and headline numbers: Table 5.9 shows Device 3 average delay 14.33 ms (PF) vs 65.40 ms (RR), and maximum delay 27.57 ms (PF) vs 297.94 ms (RR) – a direct, descriptive confirmation of RR's heavier tails. These observed distributions are precisely what elevated RR's RMSE while leaving MAE relatively low, and they align with the model-diagnostic narrative above. The

differences in R^2 , RMSE, and MAE between PF and RR are not merely numerical – they are statistically grounded in the data’s tail behaviour evidenced in the figures.

5) Statistical bottom line for this experiment

a) PF: High R^2 with low RMSE and sub-ms MAE (Figure 5.6) and tighter device distribution (Figure 5.4) indicate statistically precise latency estimation and low tails risk.

b) RR: Still-high R^2 but larger RMSE-MAE gap (Figure 5.11) and markedly heavier tails (Figure 5.10; Table 5.9) imply greater susceptibility to rare, high-delay events, which is pivotal for scheduling claims that depend on percentile latency.

Taken together, the cross-validation (averaged validation metrics) and the figure-level evidence point in the same direction: PF affords statistically tighter control of latency, while RR exhibits a heavier-tailed delay that inflates RMSE despite a low MAE – an effect that is visually and numerically anchored in Device 3’s distributions and the maxima reported in Table 5.9.

6) Overall Trade-offs and Conclusions: The choice between RR and PF schedulers depends heavily on the network’s performance goals and application requirements. RR excels in maximising throughput and maintaining equal packet distribution, which is ideal when the primary focus is on transmitting large amounts of data. Nevertheless, this comes at the cost of significantly higher delays for some devices, as seen with Device 3. In contrast, PF may be preferable when minimising delay is critical, particularly for devices operating under challenging conditions. However, it achieves lower overall throughput and displays uneven packet distribution. The optimal scheduler choice also depends on the acceptable trade-offs between throughput, delay, and fairness; none of these schedulers exhibits universal superiority.

5.3 Experiment 2: Classification-based Comparison of Scheduling Algorithms Using Operational Network Traffic

This section presents the experimental setup for evaluating and comparing the performance of various scheduling algorithms in a live network environment. Specifically, we configured scheduling disciplines such as Priority Queuing (PRIO), HTB, and CBQ utilising the Linux traffic control (tc) utility. The primary goal was to collect network traffic data across diverse QoS conditions, creating comprehensive datasets suitable for classification. ML methods, including Decision Trees and SVMs, as well as DL architectures such as CNNs and LSTMs, were applied to these datasets. This approach enabled us to uncover and analyse complex relationships within the data, facilitating a detailed comparative assessment of the effectiveness of scheduling algorithms and AI techniques.

5.3.1 Performance Evaluation of ML Models on the PRIO Dataset

5.3.1.1 Summary of Prio Dataset

The PRIO dataset comprises a network traffic capture, consisting of 134,309 packet-level records under a PRIO queuing discipline. Each packet record includes seven primary attributes: Frame Number, Timestamp, Source Address, Destination Address, Protocol, Packet Length (in bytes), and miscellaneous Packet information. All attributes are fully populated across the dataset.

Table 5.10: Descriptive Statistics for Numeric Attributes in the PRIO Dataset.

Feature	Count	Mean	Std	Min	25%	50%	75%	Max
Time (s)	134,309	394.94	231.75	0.0000	204.79	368.73	593.91	847.70
Interarrival time (s) ¹	134,308	0.00631	0.04358	- 0.0103 ²	0.00011	0.00031	0.00079	1.8983
Length (bytes)	134,309	1,036.34	1,975.25	42	60	86	1,417	26,334

¹ Δt : Time between consecutive packets.

² Minor negative values due to timestamp rounding.

Table 5.10 shows that packet length exhibits considerable variability, with a mean of 1,036.34 bytes and a standard deviation of 1,975.25 bytes. The lengths range from 42 to 26,334 bytes, with a median of 86. The interquartile range indicates that 50% of packets have lengths between 60 and 1,417 bytes.

Table 5.11 reports the protocol composition of the PRIO dataset, showing a strong dominance of TCP-based traffic and TLS (v1.2 and v1.3), which account for most encrypted sessions. The remaining protocols occur at very low frequency, indicating an imbalanced distribution that warrants note for downstream modelling and evaluation.

Table 5.11: Protocol Distribution for the PRIO Dataset.

Protocol	Count	Percentage (%)
TCP	81,954	61.02
TLSv1.2	41,836	31.15
TLSv1.3	8,094	6.03
UDP	884	0.66
DNS	708	0.53
ICMP	674	0.50
ARP	84	0.06
SSDP	56	0.04
QUIC	8	0.01
MDNS	6	<0.01
NTP	4	<0.01
TLSv1	1	<0.01

The traffic flow analysis of the dataset indicates a high degree of concentration. The top source address accounted for approximately 37.1% of the total packets (49,817), and the top five source addresses collectively accounted for over 91.7% of the traffic. Similarly, the primary destination address received approximately 42.3% of the packets (56,762), with the top five destination addresses accounting for over 93% of the traffic.

1) Dataset suitability for classical machine-learning models: From the standpoint of classical ML, the dataset’s tabular structure – with well-defined numeric (Time, Length, inter-arrival) and categorical (Protocol, and flag indicators extracted from Info) features – is immediately amenable to algorithms such as RF, Gradient Boosting, or SVMs. After one-hot encoding of protocol types and binary extraction of TCP flag presence (SYN, ACK, PSH, FIN, RST), a typical preprocessing pipeline would normalise continuous features (packet lengths, inter-arrival times) and address protocol-class imbalances via resampling or class weighting. The sample size ($\sim 1.3 \times 10^5$ packets) is sufficient to split into training, validation and test sets, and the modest dimensionality (~ 20 engineered features) ensures that classical learners can achieve low variance and high interpretability with minimal risk of overfitting, especially when combined with k-fold cross-validation.

2) Dataset suitability for DL models: For DL applications, the dataset supports sequence-modelling approaches (e.g. LSTM and Transformer encoders) by treating each flow or fixed-length packet window as a time series of feature vectors. Raw information strings can be embedded via learnable token embedding (for port flags), while numeric attributes are fed directly into dense layers or convolutional kernels. Nonetheless, the dataset’s size may constrain DL networks from overfitting unless augmented. CNN operating on multi-channel ‘packet byte’ representations or graph neural networks modelling host-interaction graphs are also feasible, but will require substantial compute and careful regularisation. In short, deep architectures can capture complex temporal and spatial correlations induced by PRIO queuing, yet demand meticulous hyperparameter tuning, and regularisation (dropout, weight decay).

5.3.1.2 Performance Metrics for PRIO Dataset

This subsection analyses the performance of ML models (Optimizable Tree and SVM) using the PRIO dataset. We present confusion matrices detailing the number of observations, Positive Predictive Value (PPV), False Discovery Rate (FDR), True Positive Rate (TPR), False Negative Rate (FNR), and Receiver Operating Characteristics (ROC) curves for the Optimizable Tree and SVM algorithms.

1) Confusion Matrix for Prio Dataset

Optimizable Tree Validation Confusion Matrix (Number of Observations) - Prio Dataset

True Class \ Predicted Class	ARP	DNS	HTTP	ICMP	MDNS	NTP	OCSP	QUIC	SSDP	TCP	TLSv1	TLSv1.2	TLSv1.3	UDP
ARP	128													
DNS		693		3				1						
HTTP											1	2		
ICMP				903								3		
MDNS					2							3		
NTP												3		
OCSP												1		
QUIC								45	1		1	7		
SSDP								83						
TCP										117472		750	5004	3
TLSv1												1	2	
TLSv1.2								1	3431		7059	63		
TLSv1.3					4		3		4483	2	94	56257		
UDP														633

Figure 5.12: Optimisable Tree validation confusion matrix (number of observations) for the Prio dataset. Most high-volume protocols (especially TCP and TLS variants) are correctly classified, with only occasional confusion between TLSv1.2, TLSv1.3 and a few HTTP/DNS flows, indicating generally strong but not perfect protocol discrimination.

Figure 5.12 shows the validation confusion matrix for the optimizable tree classifier on the Prio dataset. Each row represents the true network protocol class, while each column denotes the class predicted by the model. The diagonal entries reflect correct classifications, and off-diagonal entries indicate misclassifications. This figure also shows that most protocols are classified correctly, with only occasional confusion, primarily among some Transport Layer Security (TLS) variants (e.g., TLSv1.2 and TLSv1.3), and a small number of misclassifications involving HyperText Transfer Protocol (HTTP) or DNS.

TCP traffic constitutes the bulk of the dataset, with 117,472 observations. TLS protocols also have a strong presence (notably TLSv1.2 and TLSv1.3), while Address Resolution Protocol (ARP), Multicast Domain Name System (MDNS), and User Datagram Protocol (UDP) appear relatively infrequently, each with fewer than 1,000 samples overall.

Furthermore, these results indicate that the classifier effectively distinguishes the majority of traffic types, a point discussed in detail in the interpretation subsection.

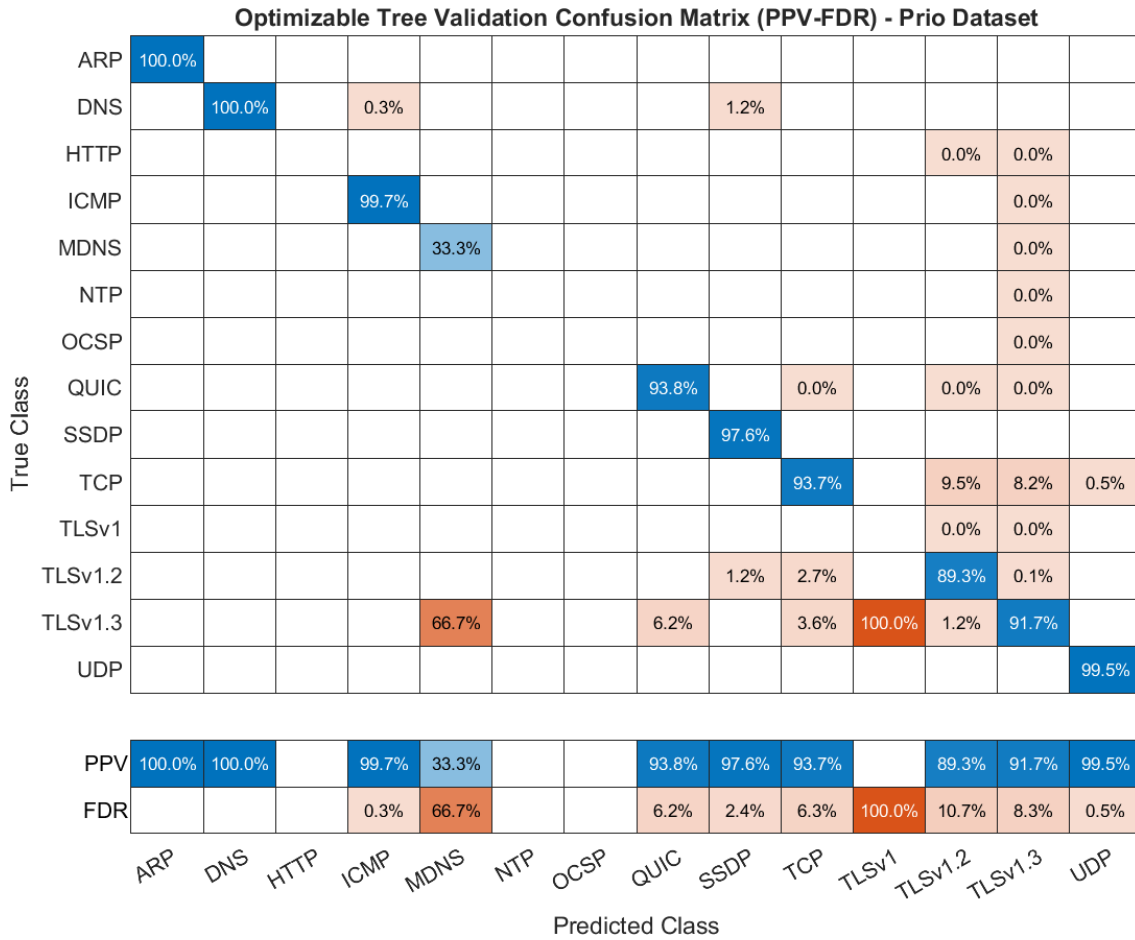


Figure 5.13: Optimisable Tre PPV-FDR validation confusion matrix (Prio dataset). Most protocols (e.g., ARP, DNS, and Quick UDP Internet Connections (QUIC)) exhibit very high PPV and low FDR, indicating few false positives, whereas TLSv1 and especially TLSv1.3 show much higher FDR, revealing substantial confusion within the TLS family.

The validation confusion matrix illustrated in Figure 5.13 summarises how well the classifier performs in terms of Positive Predictive Value (PPV) and the corresponding False Discovery Rate (FDR) across the Prio Dataset. Protocols such as ARP, DNS, and QUIC exhibit consistently high PPV (e.g., 100%, 100%, and 93%, respectively), indicating minimal false positives. In contrast, TLSv1.3 shows a more pronounced FDR, reflecting some confusion among the TLS-family protocols.

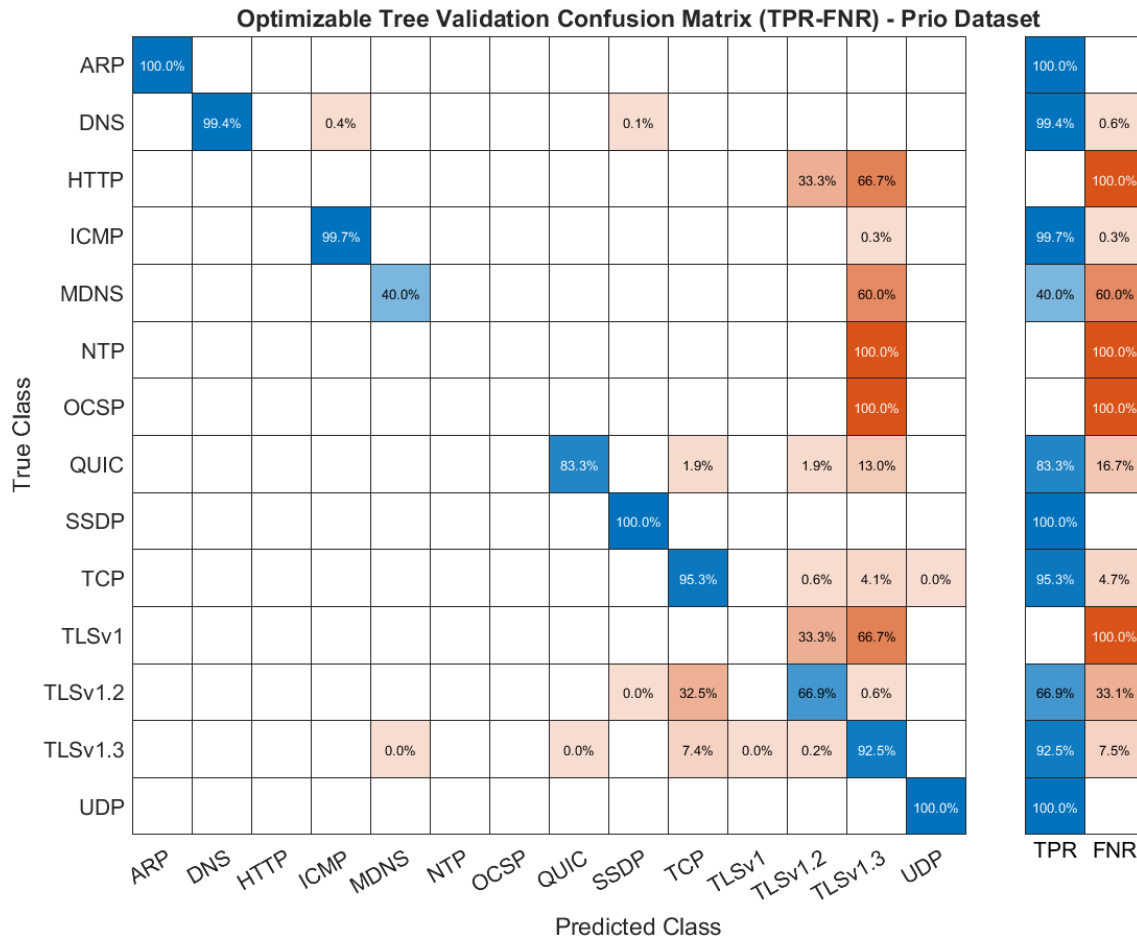


Figure 5.14: Optimisable Tree TPR-FNR validation confusion matrix (Prio dataset). ARP, NTP, and SSDP achieve near-perfect TPR with very low FNR, whereas MDNS and TLSv1 exhibit much higher FNR, revealing systematic blind spots in these protocols that could affect QoS decisions.

The confusion matrix in Figure 5.14 shows the True Positive Rate (TPR) and corresponding False Negative Rate (FNR) for each protocol in the Prio dataset. Protocols such as ARP, NTP, and SSDP achieved TPRs of 100% or near 100%, resulting in very low FNR. In contrast, other protocols, such as MDNS (40% TPR, 60% FNR) and TLSv1 (33.3% TPR, 66.7% FNR), are more prone to misclassification. Furthermore, TLSv1.2 and TLSv1.3 perform reasonably well, though they exhibit moderate FNRs that reflect occasional misidentification across TLS versions.

2) SVM Validation Confusion Matrix (TPR-FNR) for Prio Dataset: The confusion matrix presented below (Figure 5.15) illustrates the validation performance of the Fine Gaussian SVM model, specifically highlighting the TPR and FNR for each protocol class within the PRIO dataset. These metrics demonstrate the model's sensitivity in correctly identifying protocol classes and pinpoint where it is more susceptible to errors. This detailed breakdown facilitates targeted improvements to enhance network protocol classification accuracy.

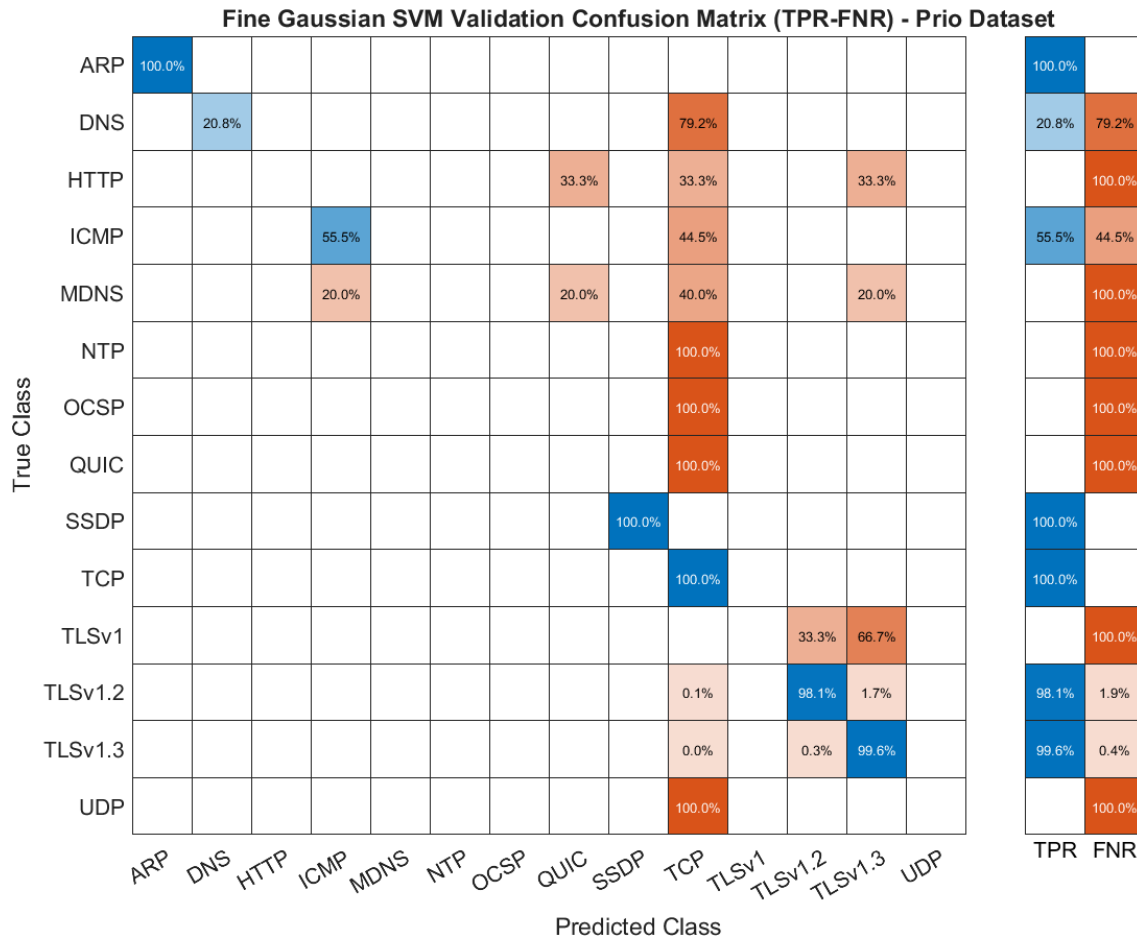


Figure 5.15: Fine Gaussian SVM TPR-FNR validation confusion matrix (Prio dataset).

The SVM detects TCP and TLSv1.2/TLSv1.3 with very high TPR and low FNR, but performs poorly on DNS, MDNS, QUIC, UDP, NTP, OCSP, and TLSv1, indicating that several non-encrypted or low-volume protocols remain difficult to recognise reliably.

Figure 5.15 presents the Fine Gaussian SVM validation confusion matrix for the Prio dataset, illustrating substantial variation in classification performance across protocols. Notably, ARP, SSDP, TCP, TLSv1.2, and TLSv1.3 achieve near-perfect TPRs and minimal FNRs, indicating that the model reliably discriminates among these protocols. In contrast, DNS, Internet Control Message Protocol (ICMP), MDNS, QUIC, UDP, Network Time Protocol (NTP), Online Certificate Status Protocol (OCSP), and TLSv1 demonstrate pronounced misclassification, reflected by reduced TPRs and elevated FNRs. These discrepancies suggest that while the classifier is well calibrated for specific protocols, further optimisation or additional features may be needed to improve the recognition of underperforming classes.

3) ROC Curve for Prio Dataset

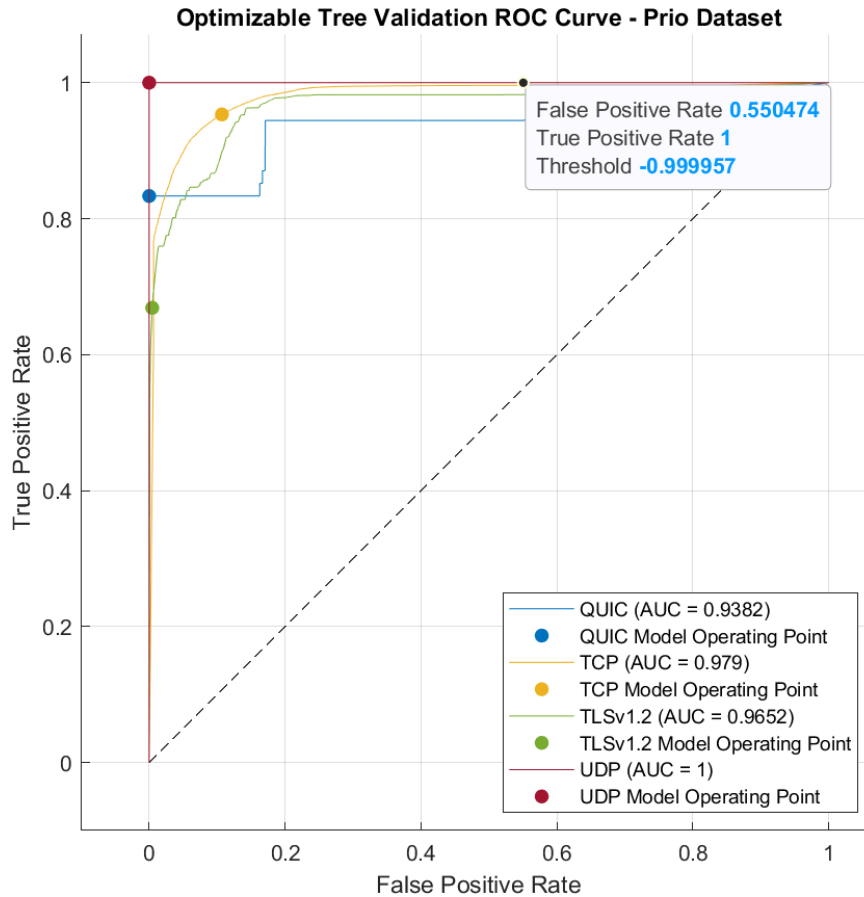


Figure 5.16: Optimisable Tree Validation ROC curve (Prio dataset). Very high AUC values for UDP, TCP, TLSv1.2, and QUIC indicate that these protocols are, in principle, highly separable, although the QUIC operating point trades a high TPR for a relatively high FPR, highlighting a QoS trade-off between sensitivity and false alarms.

Figure 5.16 shows the ROC-curve performance for each protocol’s classifier on the Prio dataset. The vertical axis indicates the True Positive Rate (TPR), while the horizontal axis indicates the False Positive Rate (FPR). Each curve corresponds to a classifier trained in a different protocol, with the respective area under the curve (AUC) values provided in parentheses: UDP achieves an AUC of 1.0, TCP achieves an AUC of 0.979, TLSv1.2 achieves an AUC of 0.9652, and QUIC achieves an AUC of 0.9382. The coloured markers along each curve indicate the “operating point” chosen for that specific model.

4) Minimum Classification Error Plot for Prio Dataset

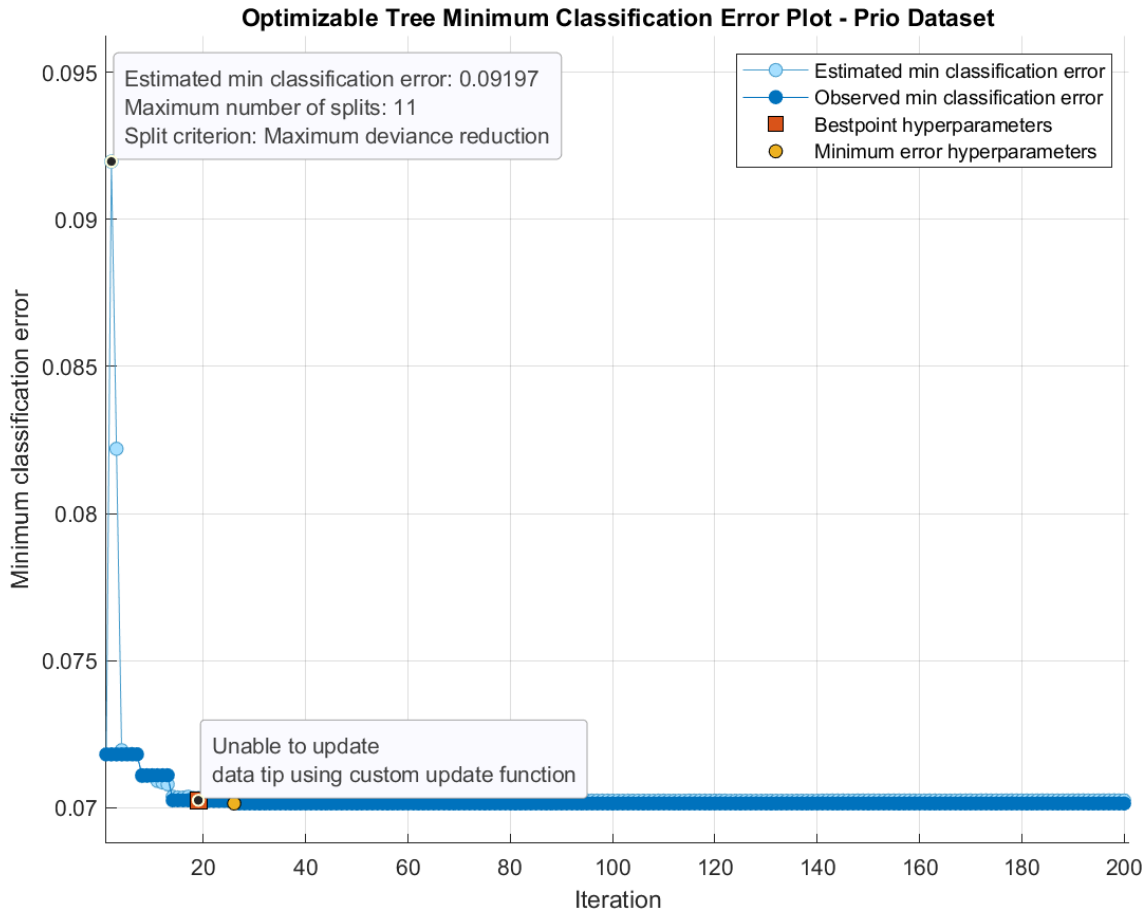


Figure 5.17: Minimum classification error for the Optimisable Decision Tree (Prio dataset). Hyperparameter optimisation reduces the minimum classification error from ~0.092 to ~0.07, demonstrating that tuning tree depth and split criteria yields a meaningful improvement in protocol classification performance.

Figure 5.17 illustrates how the optimisable decision tree's minimum classification error evolves over successive Prio dataset iterations. Initially, the estimated minimum classification error is reported at around 0.09197, reflecting the baseline parameters (with up to 11 splits and maximum deviance reduction as the split criterion). As hyperparameter optimisation proceeds, the observed classification error decreases rapidly and converges to around 0.07. The orange square and yellow circle indicate the best-performing hyperparameters and configurations, respectively, and the minimum-observed-error hyperparameters, illustrating how iterative search identifies an increasingly refined set of tree parameters that yield improved classification performance.

5.3.1.3 Interpretation of Results for Prio Dataset

This subsection discusses the significance of the performance results presented in subsection 5.3.1.2 for the Prio dataset. Reviewing each plot – the confusion matrix, ROC curves, classification error trends, and partial dependence outputs – can help better

understand the classifier's predictive behaviour, interpret key strengths or limitations, and draw broader conclusions about the model's performance.

1) Optimizable Tree Validation Confusion Matrix (Number of Observations) (Prio Dataset): The confusion matrix depicted in Figure 5.12 shows how well the Optimizable Tree model distinguishes between protocols, which is crucial for prioritising network traffic by type. A significant number of misclassifications between specific protocols could lead to QoS policies being misapplied, potentially degrading the performance of critical applications. Thus, below are some key insights and their implications on QoS:

- **TCP Dominance:** The model correctly classifies the majority of TCP packets (117,472). This fact suggests that the model accurately identifies TCP traffic, which is vital for managing most Internet traffic.
- **TLSv1.3 Performance:** The Optimisable Tree correctly classifies many TLSv1.3 packets (56257). This classification result is vital for QoS because TLSv1.3 is a modern encryption protocol for secure communication.
- **Misclassifications of TLSv1.2 Packets:** Many TLSv1.2 packets are misclassified as TCP (port 3431) and TLSv1.3 (port 63). This misclassification could be problematic if QoS policies are designed to treat these protocols differently.
- **Misclassification of TCP as TLSv1.2 and TLSv1.3:** Many TCP packets are misclassified as TLSv1.2 (750) and TLSv1.3 (5004). This misclassification of TCP packets could be problematic if QoS policies are designed to treat these protocols differently.
- **DNS Performance:** The model accurately classifies a significant number of DNS packets (693). However, some DNS packets are misclassified as ICMP (3) and SSDP (1).
- **QUIC Performance:** The model correctly classifies several QUIC packets (45). However, there are some misclassifications of QUIC packets as TCP (1), TLSv1.2 (1), and TLSv1.3 (7).

The confusion matrix in Figure 5.12 highlights the model's strengths in identifying TCP and TLSv1.3 traffic; however, it also reveals potential weaknesses in distinguishing between TLSv1.2 and TLSv1.3, as well as between TCP and TLSv1.2/TLSv1.3. These misclassifications could impact QoS policies if they rely on accurate protocol identification. Further investigation and model refinement may be necessary to improve the accuracy of protocol classification for effective QoS management.

a) QoS Implications for the Results in Figure 12: Misclassifying TCP as TLSv1.3 can distort QoS decisions, causing TCP flows to be unnecessarily prioritised or unfairly deprioritised in accordance with policies intended specifically for encrypted traffic. Such errors can trigger security policy mismatches, where controls designed for TLSv1.3 are applied to plain TCP traffic (or vice versa). This may introduce vulnerabilities, apply inappropriate inspection/handling rules, or degrade performance due to incorrect processing.

The most critical issue is the misclassification of TLSv1.3. If the QoS strategy is designed to prioritise and protect encrypted traffic, then failing to reliably identify TLSv1.3 undermines those protections and weakens the intended QoS guarantees for secure flows.

b) Alignment of the Experiment with Theoretical Foundations: Tree-based ML algorithms, such as the Optimizable Tree, aim to learn decision boundaries from the data. The misclassifications may suggest that the features used by the model are insufficient to distinguish between specific protocols, particularly TLSv1.3 and MDNS, reliably. This could be due to overlapping feature spaces or noisy, irrelevant features in the dataset. Prio scheduling relies on accurate traffic classification to assign priorities. If the model misclassifies traffic, the Prio scheduler will make incorrect decisions, potentially negating the scheduler's benefits. Misclassification of TLSv1.3 is particularly problematic, as it could lead to encrypted traffic being treated as low-priority, therefore impacting security and performance.

c) Recommendations for Improving Protocol Classification and QoS Reliability: Future work should prioritise improving feature engineering to better separate similar protocols (especially TLSv1.3 and MDNS) by auditing the current feature set, removing noisy or redundant variables, and introducing more discriminative flow and time-based features. In parallel, the Optimisable Tree should be systematically tuned (e.g., depth, splits, leaf size, and class-weighting) using cross-validation to reduce confusion between encrypted and non-encrypted traffic, while benchmarking alternative ML models that may offer stronger decision boundaries for this task.

In summary, while the model demonstrates promise in classifying some protocols, the significant misclassifications, especially those involving TLSv1.3, pose serious challenges to the effectiveness of QoS policies and the Prio scheduling algorithm. Addressing these issues through feature engineering, model tuning, or alternative algorithms is vital for ensuring reliable and secure network performance.

2) Optimisable Tree PPV-FDR Validation Confusion Matrix (Prio Dataset): Figure 5.13 displays the confusion matrix for the Positive Predictive Value (PPV) and False Discovery Rate (FDR) for the Optimisable Tree model for classifying network protocols in the Prio dataset. High PPV values along the diagonal indicate that the model generally predicts correctly when it identifies a specific protocol. On the other hand, low FDR values suggest that a small percentage of the model's predictions are incorrect. Below are some key insights and implications for QoS:

a) The model achieves high PPV for ARP, DNS, ICMP, QUIC, SSDP, TCP, and UDP, indicating reliable protocol identification. Furthermore, this is critical for QoS, as accurate protocol identification is the foundation for applying appropriate prioritisation policies.

b) While TCP classification is good (93.7% PPV), misclassifications as TLSv1.2 (9.5%) and TLSv1.3 (8.2%) are problematic. This could cause TCP traffic to be misclassified as encrypted traffic, potentially affecting performance.

c) The most significant issue is the misclassification of TLSv1.3 traffic; high FDR for TLSv1.3 as MDNS (66.7% FDR) and QUIC (6.2% FDR) is problematic because this could result in encrypted traffic not receiving the appropriate QoS treatment, potentially leading to performance bottlenecks or security vulnerabilities.

d) A 100% FDR for TLSv1 indicates that all TLSv1 predictions are false positives.

e) Network Management Implications of FDR in Protocol Classification (Figure 5.13): Protocol misclassification can lead to incorrect traffic prioritisation, directly degrading user experience and wasting network resources. For example, if HTTP is incorrectly treated as low priority, users may see slow page-load times; similarly, if video streaming traffic is deprioritised, it can cause buffering and reduced quality, even when sufficient capacity is available. Misclassification is especially risky for encrypted traffic (e.g., TLSv1.3) because it can affect both QoS enforcement and security posture. If encrypted flows are not correctly identified, they may miss intended prioritisation and protection and may bypass the correct security policies, increasing the likelihood that malicious or non-compliant traffic evades controls.

f) Theoretical Interpretation of Optimisable Tree Misclassification Behaviour: The Optimisable Tree results suggest that the model may struggle to learn clear decision boundaries between certain protocols. From a theoretical perspective, this indicates that the input features may not provide sufficient separability for the classifier to consistently distinguish protocol-specific patterns.

One likely cause is an overlapping feature space, where multiple protocols share similar statistical or flow characteristics, reducing class separability and increasing confusion. In addition, noise or errors in the training data (e.g., inconsistent labels, capture artefacts, or variable traffic conditions) can blur genuine protocol signatures and degrade the model's ability to accurately identify protocols.

g) QoS Policy Risk from High-FDR Predictions and Recommended Mitigations (Figure 5.13): The Prio discipline depends on accurate protocol labels to assign priority levels, so the false discovery patterns in Figure 5.13 translate directly into QoS policy risk. The most severe failure occurs when the model predicts TLSv1 (100% FDR) and MDNS (66.7% FDR; 33.3% PPV), meaning these predictions are frequently wrong and can trigger inappropriate handling. In practice, this can lead to misapplied prioritisation and policy enforcement, such as treating non-encrypted flows as encrypted (or the reverse), which can degrade performance and weaken security controls.

To reduce these risks, improvements should target class separability and decision reliability: refine or add discriminative features (especially those that help distinguish encrypted traffic and service-discovery patterns), apply systematic hyperparameter tuning and class weighting for the Optimisable Tree, and verify labels and clean noisy samples that can inflate false positives. Because Figure 5.13 reflects precision-style behaviour (PPV-FDR) rather than missed detections, it is worth pairing it with recall or FNR-style results to ensure that QoS-critical protocols are not only predicted accurately when detected, but also not missed entirely.

In summary, while the Optimisable Tree shows strong reliability for several protocols (e.g., ARP, DNS, ICMP, and UDP), Figure 5.13 reveals critical precision failures that can undermine Prio-based QoS enforcement. The most severe issues are the extremely high false discovery rates for TLSv1 (100% FDR) and MDNS (66.7% FDR), meaning these predictions are frequently incorrect and may trigger misapplied prioritisation or inappropriate security handling. Although TLSv1.2, TLSv1.3, QUIC, and TCP achieve relatively high PPV, their non-trivial FDR values still pose policy risk when classification outputs are used to drive traffic management decisions; therefore, improving feature separability, model tuning, and data quality is essential for reliable and secure QoS operation.

3) Optimisable Tree TPR-FNR Validation Confusion Matrix (Prio Dataset, Figure 5.14): Figure 5.14 displays the True Positive Rate (TPR) and False Negative Rate (FNR) for the Optimisable Tree model used to classify network protocols in the Prio dataset. TPR represents the percentage of actual instances of a protocol that the model correctly identifies, and the percentage of actual instances of a protocol that the model fails to identify (i.e., misclassifies as another protocol). Figure 5.14 shows that:

a) High TPR Protocols: The model shows high TPR for ARP (100%), DNS (99.4%), ICMP (99.7%), SSDP (100%), TCP (95.3%), and UDP (100%). This indicates that the model is highly effective at identifying these protocols, which is crucial for QoS management. Correctly identifying these protocols allows for the application of appropriate QoS policies, such as prioritising critical traffic or limiting bandwidth for less important traffic.

b) Lower TPR Protocols: QUIC (83.3%) and TLSv1.3 (92.5%) exhibit lower TPR values than most other protocols, indicating that a noticeable proportion of their true instances are misclassified. This is a QoS concern because misidentified QUIC and TLSv1.3 flows may not receive the intended prioritisation or policy enforcement, potentially degrading performance and weakening traffic management objectives.

c) Significant Misclassifications (High FNR): Figure 5.14 indicates substantial missed detections for several protocols, with HTTP showing 66.7% FNR and MDNS 60% FNR, meaning a large share of their true instances are misclassified. The most severe failures are NTP and OCSP, each with 100% FNR, indicating that the model fails to correctly

identify these protocols. TLSv1.2 exhibits a notable 31% FNR, and QUIC shows 16.7% FNR, both of which can undermine QoS policy enforcement by causing these flows to be treated under incorrect priority rules.

4) Optimisable Tree Validation ROC Curve (Prio Dataset, Figure 5.16): Figure 5.16 presents the ROC curve plot for the Optimisable Tree classifier. Overall, the ROC results are consistent with the confusion-matrix findings, but they provide a threshold-independent view of separability by showing the trade-off between TPR and FPR for each traffic type.

a) Per-Traffic-Type Results

UDP (red curve): UDP achieves an AUC of 1.0, indicating perfect discrimination between UDP and non-UDP traffic across all thresholds. This is reinforced by the operating point at the top-left of the ROC space (TPR=1.0, FPR=0), indicating that the classifier makes no errors on UDP in this validation set.

TCP (yellow curve): TCP attains an AUC of 0.979, reflecting excellent separability from other classes. The operating point lies near the top-left region, indicating a high TPR with a low – but non-zero FDR, meaning a small fraction of non-TCP samples are still incorrectly predicted as TCP.

TLSv1.2 (green curve): TLSv1.2 achieves an AUC of 0.9652, indicating strong discriminative capabilities and near TCP-level separability from other traffic types. However, its operating point lies slightly further from the top-left corner than UDP and TCP, implying a higher FPR at the chosen threshold. This is consistent with the confusion matrix findings, which show that TLSv1.2 exhibits some degree of misclassification.

QUIC (blue curve): QUIC attains an AUC of 0.9382, which remains a strong result but is lower than UDP, TCP, and TLSv1.2, indicating comparatively weaker separability. Its operating point is farther from the top-left corner, indicating a higher FPR at the selected threshold, consistent with the confusion matrix evidence of QUIC misclassifications. This pattern may reflect feature-level similarity between QUIC and other traffic types (and/or limited QUIC samples), and it suggests that selecting an alternative threshold could be beneficial depending on whether the QoS objectives prioritise reducing FPR over maximising detection.

b) ROC-specific QoS Implications for 5G Networks and Beyond: High AUC values, particularly for UDP, TCP, and TLSv1.2, indicate that the classifier can reliably distinguish these traffic types. In operational networks, this supports granular QoS enforcement and enables operators to prioritise or throttle specific services with reduced risk of unintentionally affecting other traffic classes.

UDP's near-perfect separability is especially valuable because many UDP applications are delay-sensitive (e.g., real-time voice and streaming). With reliability identification, QoS policies such as higher priority scheduling or tighter latency targets can be applied more confidently, reducing the likelihood of misapplied treatment for non-UDP traffic.

For QUIC, the ROC curve suggests a more pronounced trade-off between detection and false alarms at the chosen operating point. If the threshold is configured to minimise false negatives (to capture most QUIC flows), a higher FPR may cause non-QUIC traffic to be incorrectly prioritised, potentially wasting resources and distorting scheduling decisions. In this case, threshold recalibration (or cost-sensitive tuning) is essential to ensure that QoS measures are applied only when genuinely warranted.

c) Theoretical Foundations and Interpretation of ROC-AUC Behaviour (Figure 5.16)

Area Under the Curve (AUC): AUC measures the probability that a classifier ranks a randomly chosen positive instance higher than a randomly chosen negative instance. An AUC close to 1.0 indicates near-perfect separability. The fact that these protocols achieve AUC values above 0.9 (with UDP at 1.0) shows the classifier performance far better than random guessing (AUC=0.5).

ROC vs. Precision–Recall: Although not shown here, ROC curves are typically used to assess overall discriminative performance and can be informative even when class distributions vary. However, if the dataset is highly imbalanced (e.g., one protocol is much rarer than others are), Precision-Recall curves may provide clearer insights into performance on the minority class. Nevertheless, the consistently high AUC values in this experiment indicate a strong discrimination capability.

Threshold and Trade-Offs: The ROC curve illustrates how varying the decision threshold shifts the balance between TPR and FPR. For QUIC in particular, part of the curve approaches $\text{TPR} \approx 1.0$ at the cost of a higher FPR, highlighting the theoretical trade-off: near-maximal detection can be achieved only by accepting more false positives.

d) Practical Considerations for ROC Interpretation and Deployment Robustness

Dataset Composition: The perfect classification of UDP suggests either (a) UDP flows in the dataset have highly distinctive features or (b) the UDP class is comparatively homogeneous (and/or limited in variety), making it less prone to confusion. Future work could include a broader range of UDP applications (e.g., additional streaming and real-time variants) to confirm whether the classifier maintains perfect discrimination.

Operating Points and Real-world Policy: In operational deployments, it is often preferable to choose an operating point that controls FPR more strictly, especially when QoS actions are costly or affect other services. For example, for QUIC traffic, it may be

acceptable to reduce TPR slightly to avoid excessive misclassification of non-QUIC flows.

Model Complexity and Overfitting: A near-perfect performance can sometimes indicate overfitting. However, because these ROC curves are reported on the validation set, the results suggest reasonable generalisation within the current dataset. Testing on more diverse, truly unseen traffic remains essential to confirm robustness.

e) Feature Engineering and Cross-Protocol Overlaps: Specific protocols may share packet-size distributions or timing patterns (e.g., QUIC vs. TCP and TLS). Examining the underlying features driving the splits in the Optimisable Tree would be valuable, as it can indicate whether more granular features (e.g., handshake metadata or early flow statistics) are required

Another critical aspect is that encrypted traffic obscures payload details, forcing the classifier to rely on metadata (e.g., packet size, timing, or handshake signatures). Because the TLS family (including TLSv1.2 and TLSv1.3, as well as QUIC's TLS-based handshake) shares many handshake and encryption characteristics, decision boundaries between these protocols can blur. Consequently, engineering features that capture subtle differences, such as session establishment behaviour, cypher negotiation fields, and inter-packet timing, become essential; without such targeted features, overlapping encryption patterns can increase false positives and reduce classification confidence across TLS-based protocols.

5) Cross-Figure QoS Implications for Prio-Based Scheduling (Figures 5.13, 5.14 and 5.16): While the ROC curves (Figure 5.16) indicate strong, overall separability for several classes, the PPV/FDR and TPR/FNR results (Figures 5-13-5.14) reveal the operational risk that matters for Prio enforcement: false positives can trigger incorrect prioritisation, and false negatives can cause QoS-critical flows to miss their intended treatment. In particular, TLSv1 predictions are unreliable (100% FDR) and MDNS predictions are frequently wrong (66.7% FDR), creating a high likelihood of misapplied policy actions; meanwhile, high FNR for HTTP, NTP, and OCSP indicates these flows may be consistently under-identified and therefore not prioritised when required. For Prio-based scheduling, these error modes can translate into inconsistent policy enforcement (especially for encrypted and service-discovery traffic), suggesting the need for more conservative handling of low-confidence classes and targeted improvements to reduce both false discoveries and missed detections.

6) Cross-Figure Theoretical Interpretation of Protocol Separability (Figure 5.13-5.16): The results across Figures 5.13-5.16 align with the theoretical expectation that protocols with clear, more separable signatures (e.g., ARP and ICMP) are easier to classify, as reflected in consistently strong PPV/TPR and near-perfect ROC behaviour. In contrast, QUIC and TLS-family traffic exhibit greater overlap in observable metadata due to encryption and similar handshake characteristics, reducing feature separability and

increasing the likelihood of confusion or threshold trade-offs despite strong overall ROC-AUC. Notably, the unreliable behaviour for specific labels (e.g., TLSv1 and MDNS in PPV/FDR, and high-FNR protocols in TPR/FNR) reinforces the principle that when classes share structural traits or when distinguishing cues are weak at the feature level, classification errors increase, even if aggregate discrimination metrics remain high.

7) Summary: Across Figures 5.13-5.16, the Optimisable Tree shows strong overall discrimination for key classes in the Prio dataset, with ROC-AUC results supporting good separability (notably for UDP, TCP, and TLSv1.2), enabling more confident protocol-driven QoS enforcement under Prio. However, the confusion matrix metrics reveal operational risks that ROC alone can hide: unreliable labels, such as TLSv1 (100% FDR) and MDNS (66.7% FDR), can lead to incorrect prioritisation, while high FNR for protocols such as HTTP, NTP, and OCSP suggests missed identification or inconsistent policy application. For deployment, threshold selection should reflect the cost of false positives versus false negatives (particularly for QUIC), and validation on more diverse traffic is essential to confirm robustness.

5.3.1.4 Comparative Performance Analysis (PRIO Dataset): Optimizable Tree vs. Fine Gaussian SVM

Table 5.12 presents the per-class True Positive Rates (TPR) for the Optimizable Tree and Fine Gaussian SVM models on the Prio dataset. By listing each protocol and the corresponding TPR side by side, this table offers a clear view of which model excels for specific classes and where improvements may be needed.

Table 5.12: Comparative analysis between Optimisable Tree and Fine Gaussian SVM on the PRIO Dataset.

Class	Optimizable Tree (TPR)	Fine Gaussian SVM (TPR)	Best Model
ARP	100%	100%	Tie
DNS	99.4%	20.8%	Tree
HTTP	0.0%	0.0%	Tie
ICMP	99.7%	55.5%	Tree
MDNS	40%	0.0%	Tree
NTP	0.0%	0.0%	Tie
OCSP	0.0%	0.0%	Tie
QUIC	83.3%	0.0%	Tree
SSDP	100%	100%	Tie
TCP	95.3%	100%	SVM
TLSv1	0.0%	0.0%	Tie
TLSv1.2	66.9%	98.1%	SVM
TLSv1.3	92.5%	99.6%	SVM
UDP	100%	0.0%	Tree

The findings reveal notable performance differences between the Optimizable Tree and the Fine Gaussian SVM classifiers when applied to network traffic data collected under the PRIO scheduling scheme. The results clearly illustrate varying levels of classification accuracy depending on the type of network traffic.

Both classifiers demonstrated exceptional accuracy in identifying ARP and SSDP traffic, each achieving a TPR of 100%, indicating that the features extracted from these classes are sufficiently distinct and easily separable by both approaches. In contrast, neither classifier correctly identified HTTP, NTP, OCSP, or TLSv1 traffic, with TPRs remaining at 0% for these classes. This poor classification performance suggests the possible presence of indistinguishable or overlapping traffic characteristics in these cases, or insufficient or poorly representative data in the training dataset.

The Optimizable Decision Tree classifier significantly outperformed the Fine Gaussian SVM across multiple classes, notably DNS (99.4% vs 20.8%), ICMP (99.7% vs 55.5%), QUIC (83.3% vs 0.0%), MDNS (40% vs 0.0%), and UDP (100% vs 0.0%). In addition, these results highlight the Tree model's robustness and superior ability to distinguish diverse network traffic classes, especially in contexts where feature characteristics may be complex or less distinctly separated. The strong performance of the Decision Tree model in correctly identifying UDP and ICMP traffic underscores its suitability for real-time and diverse network environments.

In contrast, the Fine Gaussian SVM demonstrated significant performance advantages, particularly in classes involving standard TCP and encrypted traffic, such as TLSv1.2 (98.1% vs. 66.9%) and TLSv1.3 (99.6% vs. 92.5%). The near-perfect accuracy of the SVM classifier for these encrypted and complex-protocol traffic classes suggests that the SVM model is particularly effective at distinguishing subtle variations in traffic characteristics, presumably due to its intrinsic capacity to capture intricate patterns inherent to encrypted data streams.

In summary, these results indicate that neither the Optimizable Decision Tree nor the Fine Gaussian SVM demonstrated universal superiority across all network traffic classes under the PRIO scheduling scheme. Instead, each model excelled in distinctly different scenarios, with the Decision Tree notably outperforming the SVM in classifying diverse, non-encrypted traffic, such as DNS, ICMP, UDP, and QUIC. In contrast, the Fine Gaussian SVM demonstrated exceptional accuracy for TCP and encrypted protocols, including TLSv1.2 and TLSv1.3. This finding underscores the importance of selecting machine-learning algorithms tailored to specific protocol characteristics and suggests that leveraging the complementary strengths of multiple models via ensemble techniques could improve overall network classification performance.

5.3.2 Performance Evaluation of ML Models on the Hierarchical Token Bucket (HTB) Dataset

5.3.2.1 Summary of HTB Dataset

The HTB dataset comprises a network traffic capture containing 488,673 packet-level records extracted from a Wireshark CSV file. Each packet record is characterised by seven attributes: Frame Number, Timestamp, Source Address, Destination Address, Protocol, Packet Length (in bytes), and Packet Information (Info). The dataset is complete, with no missing values across these attributes.

Table 5.13 summarises the key numeric attributes:

Table 5.13: Descriptive Statistics for Numeric Attributes in the HTB Dataset.

Feature	Count	Mean	Std	Min	25%	50%	75%	Max
Time (s)	488,673	860.16	463.14	0.0000	502.08	876.59	1,257.56	1,630.35
Inter arrival time (s)¹	488,672	0.00334	0.02724	0.0000	0.00018	0.00046	0.00093	1.96448
Length (bytes)	488,673	1,323.88	2,194.35	42	60	207	1,437	26,334

¹ Δt : Time between consecutive packets.

As shown in Table 5.13, packet lengths in the HTB dataset exhibit considerable variability. The mean packet length is 1,323.88 bytes, with a standard deviation of 2,194.35 bytes. The median packet length is 207 bytes, and the interquartile range indicates that 50% of packets have lengths between 60 and 1,437 bytes.

Table 5.14 details the protocol breakdown. The protocol distribution is predominantly TCP-based, with a significant presence of its secure variants.

Table 5.14: Protocol Distribution for the HTB Dataset.

Protocol	Count	Percentage (%)
TCP	364,289	74.55
TLSv1.3	80,430	16.46
TLSv1.2	40,015	8.19
ICMP	1,947	0.40
DNS	851	0.17
UDP	774	0.16
ARP	194	0.04
SSDP	104	0.02
QUIC	41	0.01
MDNS	10	<0.01
NTP	4	<0.01

1) Dataset suitability for classical ML models: The tabular nature of this dataset, with continuous features (such as packet length and inter-arrival times) and categorical fields (such as protocol types and binary TCP-flag indicators), is well-suited to classical ML pipelines. After applying one-hot encoding to extract flag presence (SYN, ACK, PSH, FIN, RST), standardising continuous variables, and mitigating class imbalance (via resampling or class weights), a feature set of modest dimensionality (≈ 20) was obtained. With ($\approx 4.9 \times 10^5$) samples, one can reliably partition into training, validation and test subsets and apply k-fold cross-validation to optimise algorithms such as Random Forests, Decision Trees, EGPR, and SVM models. The ample sample size and limited feature space promise low variance and high interpretability, facilitating explainable intrusion, or anomaly-detection models with minimal risk of overfitting.

2) Dataset suitability for DL models: For DL models, the HTB dataset enables sequence-modelling approaches (LSTM and Transformers) by treating flows or fixed-length packet windows as a temporal series of feature vectors. These models can be used for tasks such as fine-grained encrypted traffic analysis, sophisticated intrusion detection, and advanced forecasting of QoS metrics. CNN models can be used with raw packet-byte ‘images’ and graph neural networks modelling host-interaction graphs; however, these models might demand substantial computational resources and extensive hyperparameter search to generalise effectively under the HTB queuing scheme.

5.3.2.2 Performance Metrics for HTB Dataset

In this subsection, we present and critically analyse performance metrics resulting from the evaluation of ML models applied to the HTB dataset. The analysis comprises three distinct parts:

- a) Firstly, we discuss the validation metrics of the Optimizable Tree model by evaluating matrices depicting PPV against FDR, and TPR against FNR. This provides comprehensive insights into both the predictive reliability and error characteristics inherent to the model.
- b) Next, we provide a detailed assessment of the SVM model through its validation confusion matrix, emphasising TPR and FNR. This examination demonstrates the model’s efficacy in accurately distinguishing between classes in the HTB dataset.
- c) Lastly, we present and discuss the ROC curve for the Optimisable Tree model validation. ROC analysis is an essential evaluation tool for quantifying a model’s discriminative capability and explicitly illustrating its performance in distinguishing between classification outcomes.

1) Optimisable Tree Validation Matrices for HTB Dataset

Optimizable Tree Validation Confusion Matrix (PPV-FDR) - HTB Dataset												
True Class	ARP	DNS	ICMP	MDNS	NTP	QUIC	SSDP	TCP	TLSv1	TLSv1.2	TLSv1.3	UDP
	100.0%											
		100.0%						0.0%				
			84.8%									
				100.0%								
					100.0%						0.0%	
								0.0%			0.0%	
							100.0%					
			0.3%					100.0%	40.0%	13.6%	21.0%	
									20.0%	0.0%	0.0%	
								0.0%	26.7%	86.3%	0.1%	
									13.3%	0.2%	78.9%	
			15.0%									100.0%
PPV	100.0%	100.0%	84.8%	100.0%	100.0%		100.0%	100.0%	20.0%	86.3%	78.9%	100.0%
FDR			15.2%					0.0%	80.0%	13.7%	21.1%	
Predicted Class												

Figure 5.18: Optimisable Tree PPV-FDR validation confusion matrix (HTB dataset). Most protocols (ARP, DNS, MDNS, NTP, SSDP, and UDP) achieve very high PPV and low FDR, indicating few false positives, whereas ICMP, TCP and especially the TLS variants show higher FDR, revealing notable cross-misclassification and reduced precision for these classes.

Figure 5.18 shows how well the Optimisable Tree classifier distinguishes among 12 network-protocol classes (row=true label, columns=predicted label) in the HTB dataset. Each cell reports the percentage of instances from a given true class (row) predicted as a particular class (column). The main diagonal (shaded blue) shows correct classification, while the off-diagonal cells indicate misclassifications.

At a glance, ARP, DNS, MDNS, NTP, SSDP, and UDP are classified with perfect accuracy, whereas ICMP and TCP show some confusion with other protocols. The three TLS variants (TLSv1, TLSv1.2, and TLSv1.3) also exhibit cross-misclassification among themselves and with TCP. The bottom row displays the Positive Predictive Value (PPV) and False Discovery Rate (FDR) for each predicted class, providing an additional view of precision and error rates across protocols.

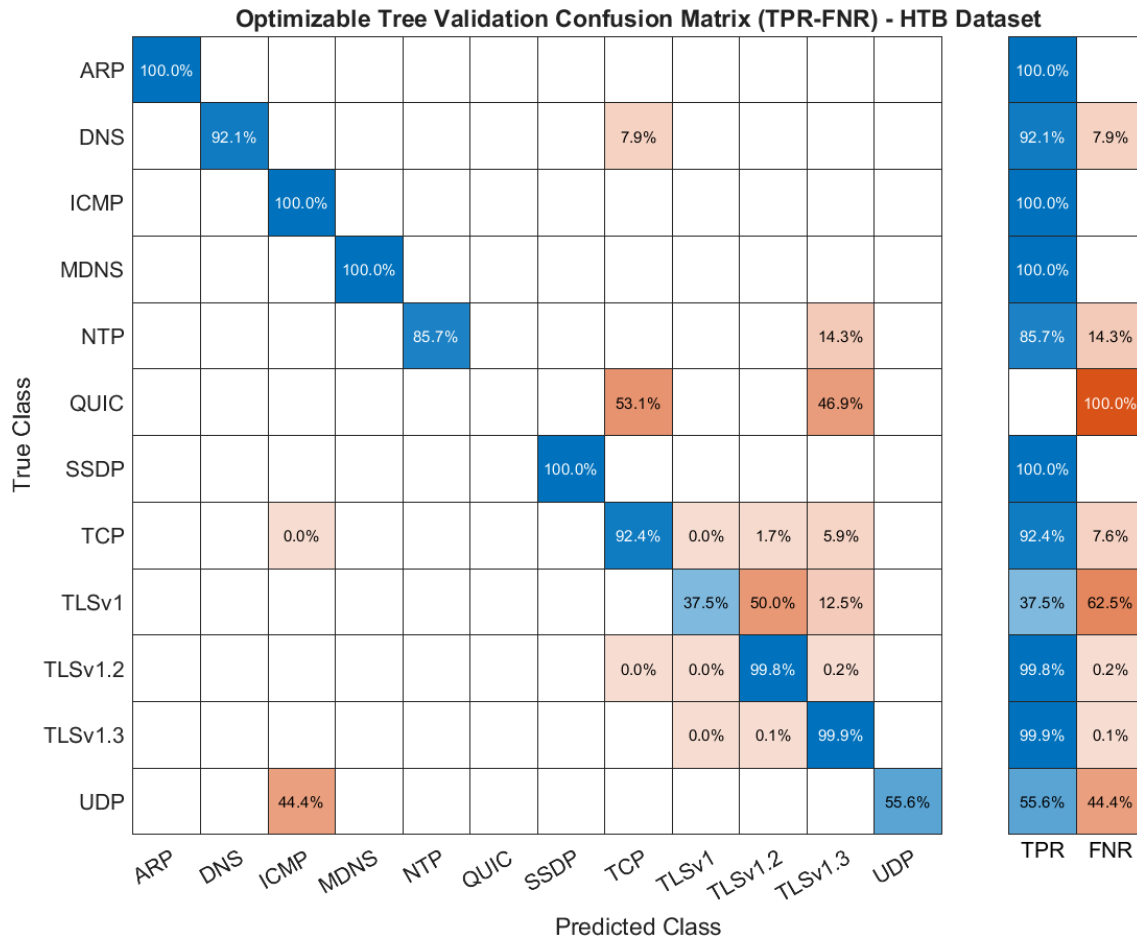


Figure 5.19: Optimisable Tree TPR-FNR validation confusion matrix (HTB dataset). ARP, ICMP, MDNS, and SSDP achieve perfect detection (TPR=100%), while DNS, NTP, and TCP retain high TPRs ($\geq 85\%$) with limited misses. In contrast, QUIC and UDP show much weaker detection (0-56% TPR), and TLSv1 remains hard to recognise, highlighting clear protocol-specific strengths and weaknesses of the classifier.

Figure 5.19 shows the validation confusion matrix for the HTB dataset, displaying the True Positive Rate (TPR) and corresponding False Negative Rate (FNR) for each of the 12 protocol classes (rows represent the actual class, and columns represent the predicted class). The blue diagonal cells indicate the percentage of correctly identified instances (the TPR), and the off-diagonal cells show the proportion of the true class incorrectly predicted as another protocol (contributing to the FNR).

A glance at the rows reveals that:

- ARP, ICMP, MDNS, and SSDP are each identified correctly 100% of the time.
- DNS has 92.1% TPR and 7.9% misclassified as TCP.
- NTP reaches 85.7% TPR, with 14.3% misclassified as TLSv1.3.
- QUIC traffic is misclassified, with 53.1% classified as TCP and 46.9% classified as TLSv1.3.
- UDP traffic has 55.6% TPR, and 44.4% is misclassified as ICMP.

- TCP achieves 92.4% TPR, with 7.6% of its instances misclassified as TLSv1.2 and TLSv1.3
- TLSv1 traffic is 62.5% misclassified as TLSv1.2 and TLSv1.3, with only 37.5% TPR.
- TLSv1.2 achieves a 99.8% TPR, with the remaining 0.2% of TLSv1.2 flows being misclassified as TLSv1.3.
- TLSv1.3 achieves a 99.9% TPR, with the remaining 0.1% of TLSv1.3 flows being misclassified as TLSv1.2.

2) SVM Validation Confusion Matrix (TPR-FNR) for HTB Dataset

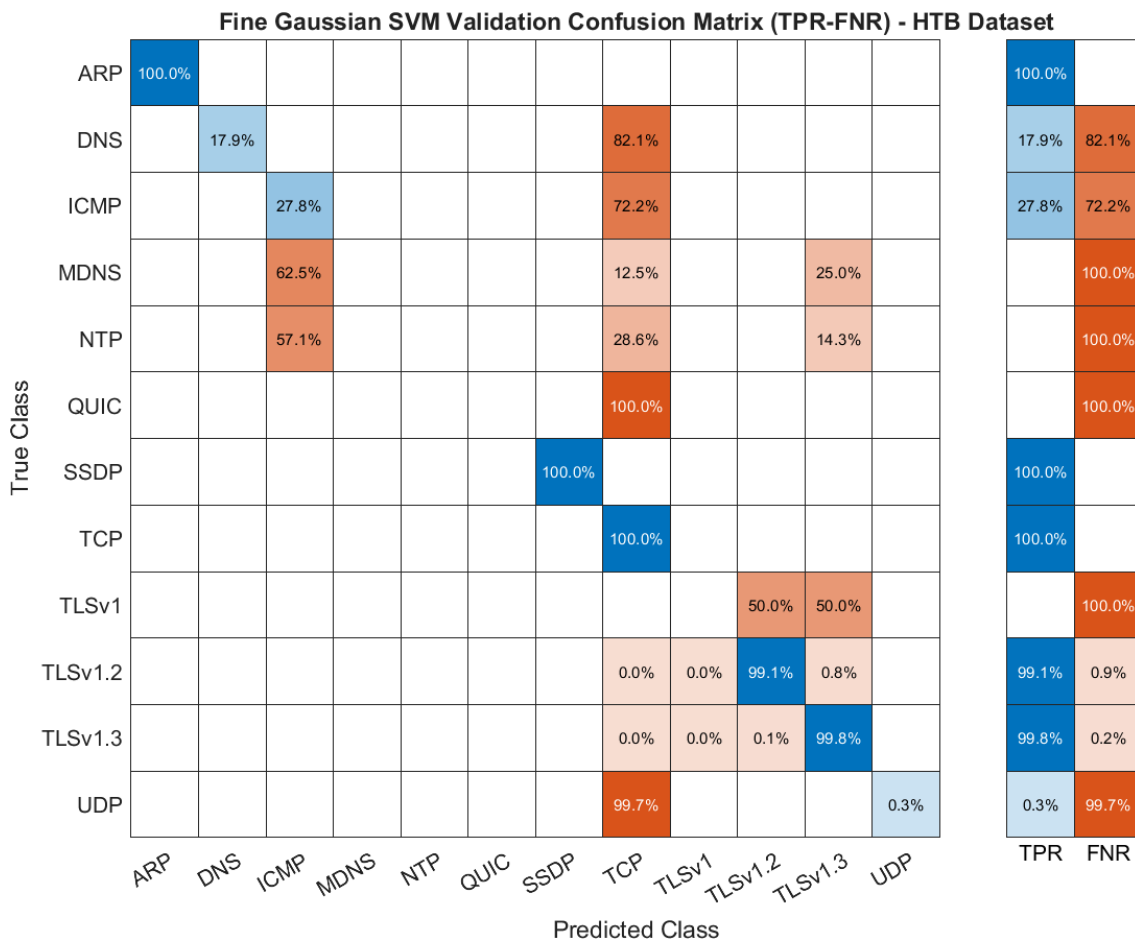


Figure 5.20: Fine Gaussian SVM validation confusion matrix (TPR-FNR) for the HTB dataset. ARP, SSDP, TCP, TLSv1.2 and TLSv1.3 achieve near-perfect TPRs (~99-100%), while MDNS, NTP and QUIC are never detected (TPR=0%, FNR=100%), and UDP, ICMP and DNS show very high FNRs, revealing strong performance for major TCP/TLS traffic but serious weaknesses for several low-volume or non-TCP protocols.

The Gaussian SVM validation confusion matrix analysis for the HTB dataset (Figure 5.20) shows heterogeneous classification performance across different network protocols. Protocols such as ARP, SSDP, and TCP are identified with exceptional accuracy, each achieving a TPR of 100%. Similarly, the secure protocols TLSv1.2 and TLSv1.3 exhibit

near-perfect detection rates, with TPRs of 99.1% and 99.8%, respectively. These outcomes indicate that the classifier is highly effective at recognising and correctly categorising these well-defined patterns, likely due to distinct, discriminative feature representations associated with these protocols.

In contrast, the confusion matrix highlights significant deficiencies in classifying other protocols. Protocols like MDNS, NTP, and QUIC suffer from a complete failure in detection, as evidenced by a 100% FNR, underscoring the model's inability to capture their unique characteristics. The situation is exacerbated by the misclassification of UDP traffic, with 99.7% erroneously labelled as TCP, suggesting considerable overlap or ambiguity in the feature space between the two protocols. Protocols such as ICMP and DNS present high misclassification rates, with FNRs of 82.1% and 72.2%, respectively, further indicating substantial challenges in their proper identification. These findings collectively emphasise the need for targeted model refinements and potential enhancements in feature engineering to bolster the classifier's overall robustness and accuracy across the full spectrum of protocol types.

3) Optimizable Tree Validation ROC Curve for the HTB Dataset

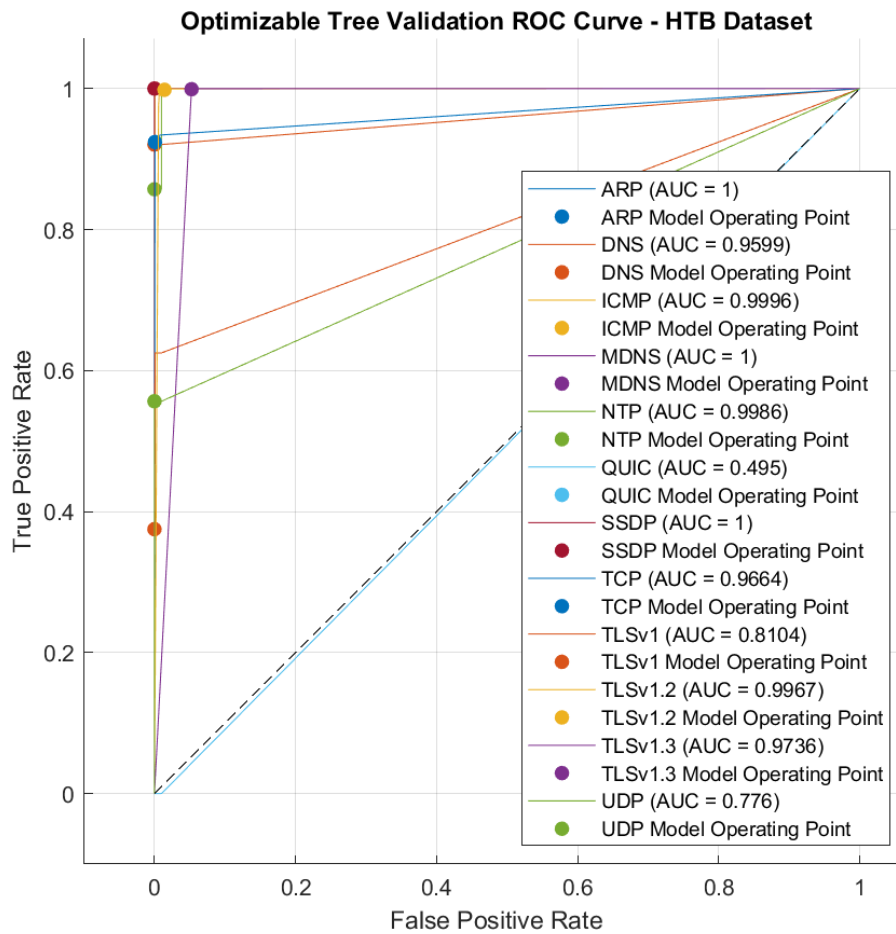


Figure 5.21: Optimisable Tree validation ROC curve (HTB dataset). Most protocols (ARP, MDNS, ICMP, NTP, TLSv1.2, and TLSv1.3) achieve very high AUCs,

indicating excellent separability, whereas QUIC (AUC ≈ 0.50) and, to a lesser extent, UDP (AUC ≈ 0.78) remain hard to distinguish, consistent with their poorer performance in the confusion matrices.

Figure 5.21 shows the ROC curves for each protocol, indicating how well the Optimizable Tree model distinguishes between them. The AUC result (in parentheses) next to the protocol label measures overall performance, ranging from nearly perfect (AUC = 1.0 for ARP, MDNS, SSDP) to nearly random (AUC = 0.495 for QUIC). Dots on each curve mark the chosen operating point (the model's final threshold). High-AUC classes (e.g., ICMP, NTP, TCP, TLSv1.2, and TLSv1.3) show strong discrimination against other protocols, while lower-AUC classes (notably QUIC and UDP) reveal more difficulty in differentiation.

5.3.2.3 Interpretation of Results for HTB Dataset

A careful analysis of both validation confusion matrices (PPV-FDR and TPR-FNR) reveals that ARP, ICMP, MDNS, and SSDP perform strongly. These protocols show nearly perfect TPRs and high Positive Predictive Values (PPVs), indicating they are rarely misidentified or over-predicted. Protocols such as DNS, NTP, and TCP also generally achieve high TPR. However, they are occasionally confused with other classes. This is reflected in a moderate False Negative Rate (FNR) and a slight reduction in PPV. The UDP and QUIC traffic show more pronounced confusion, as indicated by lower TPRs. Their False Discovery Rates (FDRs) are also higher in the PPV-FDR matrix. This suggests that these protocols not only miss actual instances but also misclassify other protocols as themselves. The TLS family exhibits varying levels of confusion among its versions. For instance, TLSv1 has a lower TPR and a higher FNR. In contrast, TLSv1.2 and TLSv1.3 exhibit extremely high TPR but moderate FDR in the PPV-FDR matrix. This indicates that these two protocols are usually recognised when present, but also tend to pull in some non-TLS traffic.

The ROC curve indicates the classifier's ability to isolate each protocol from the rest. Classes with very high AUC (e.g., ARP, MDNS, SSDP, ICMP, NTP, TLSv1.2, TLSv1.3, and TCP) generally show low misclassification in both TPR-FNR and PPV-FDR matrices, consistent with their curves clustering near the upper-left corner. Conversely, QUIC, with an AUC of 0.495, aligns with its frequent misclassifications in the confusion matrices, as it shows lower TPR and higher FDR. The UDP traffic exhibits moderate AUC (0.776) and moderate confusion in the matrices.

Regarding the TLS family, the ROC curves show high recognition for TLSv1.2 and TLSv1.3 (AUC = 0.9967 and 0.9736, respectively), which aligns with their nearly perfect TPR and minimal off-class predictions in the TPR-FNR matrix. Nevertheless, these protocols sometimes pull non-TLS traffic in the PPV-FDR matrix, mirroring the confusion matrices, where TLSv1 sees more cross-misclassification than the other protocols. Overall, the ROC curve corroborates the patterns observed in both confusion

matrices: most protocols are cleanly separable, while QUIC, UDP, and TLSv1 are more challenging to classify than TCP and TLSv1.3.

Furthermore, all protocols except QUIC are accurately predicted to achieve strong classification metrics; QUIC, with an AUC near 0.5, is practically random, suggesting it is notably more challenging for the model to distinguish. The TLSv1 protocol suffers more confusion than TLSv1.2 and TLSv1.3, which are almost perfectly identified; however, the algorithm sometimes captures non-TLS traffic.

5.3.2.4 Comparative Performance analysis (HTB Dataset): Optimizable Tree vs. Fine Gaussian SVM vs. Fine Tree

Table 5.15: Comparative Analysis between the Optimisable Tree vs Fine Gaussian SVM and Fine Tree on the HTB Dataset.

Protocol	Optimizable Tree (TPR%)	Fine Gaussian SVM (TPR%)	Fine Tree (AUC)
ARP	100%	100%	1.0
DNS	92.1%	17.9%	0.9599
ICMP	100%	27.8%	0.9996
MDNS	100%	0.0%	1.0
NTP	85.7%	0.0%	0.9986
QUIC	0.0%	0.0%	0.495
SSDP	100%	100%	1.0
TCP	92.4%	100%	0.9664
TLSv1	37.5%	0.0%	0.8104
TLSv1.2	99.8%	99.1%	0.9967
TLSv1.3	99.9%	99.8%	0.9736
UDP	55.6%	0.3%	0.776

The comparative performance analysis shown in Table 5.15 reveals distinct strengths and limitations among the evaluated models. For several protocols, such as ARP and SSDP, the Optimizable Tree and Fine Gaussian SVM both achieved perfect TPRs (100%), with the Fine Tree classifier also yielding an AUC of 1.0. This result indicates that, for these protocols, the classification boundaries are well delineated, and each method can achieve near-ideal performance.

In contrast, there is a pronounced discrepancy for protocols such as DNS, ICMP, MDNS, NTP, TLSv1, and UDP. For DNS and ICMP, the Optimizable Tree achieved TPRs of 92.1% and 100%, respectively, while the Fine Gaussian SVM lagged significantly at 17.9% and 27.8%, respectively. Similar trends are observed for MDNS and NTP: the Optimizable Tree maintained perfect or near-perfect detection rates, whereas the SVM registered 0% TPR for both. UDP demonstrates a stark contrast: the Optimizable Tree achieves 55.6% TPR, whereas the SVM achieves a negligible 0.3%. The AUC values for the Fine Tree remain robust (ranging from 0.776 to 1.0) for most protocols, except for QUIC, where an AUC of 0.495 suggests a performance near chance level.

Furthermore, for protocols such as TCP, TLSv1.2, and TLSv1.3, both approaches perform comparably. TCP detection rates are high for both methods (92.4% for Optimizable Tree and 100% for SVM), while TLSv1.2 and TLSv1.3 exhibit TPRs close to 100% with minor differences between the classifiers. These outcomes suggest that the dataset characteristics for these protocols align well with the underlying assumptions of both the Tree-based and SVM models.

In summary, the analysis underscores that while the Optimizable Tree consistently achieves superior TPRs across multiple protocols, the Fine Gaussian SVM struggles with a subset of protocols, likely due to its sensitivity to the feature space or to parameter-tuning challenges inherent to the dataset. The consistently high AUC values obtained with the Fine Tree reinforce the notion that a Tree-based approach may be more suitable for this dataset, providing a reliable decision boundary across most protocols.

5.3.3 Performance Evaluation of ML Models on the Class-Based Queuing (CBQ) Dataset

5.3.3.1 Summary of CBQ Dataset

The CBQ dataset comprises 301,629 network packets, derived from a network traffic capture conducted under the CBQ queuing discipline. Similar to the PRIO and HTB datasets, each packet entry includes seven standard attributes: Frame Number, Timestamp, Source and Destination Addresses, Protocol, Packet Length (in bytes), and Packet Information (Info). All attributes are fully populated across the dataset, with no missing values.

Table 5.16 provides descriptive statistics for the captured CBQ traffic features, including time, interarrival, and packet length. The table summarises central tendency and dispersion (mean, standard deviation, and quartiles) across 301,629 observations. It highlights high packet-size variability and generally short interarrival times, with occasional significant gaps reflected in the maximum values.

Table 5.16: Descriptive Statistics for Numeric Attributes in the CBQ Dataset.

Feature	Count	Mean	Std	Min	25%	50%	75%	Max
Time (s)	301,629	1,238.88	696.88	0.000	617.88	1,297.85	1,837.89	2,418.05
Interarrival (s) ¹	301,628	0.000802	0.05380	0.000	0.00010	0.00036	0.00088	2.19640
Length (bytes)	301,629	1,158.22	2,205.89	42	60	89	1,417	26,334

¹ Δt : Time between consecutive packets.

Table 5.17 summarises the protocol composition of the CBQ dataset, reporting per-protocol counts and relative percentage across the captured traffic. The dataset is dominated by TCP (65.78%) and TLS-encrypted traffic (TLSv1.3 and TLSv1.2), while

control and discovery protocols (e.g., ICMP, DNS, ARP) account for only a marginal share.

Table 5.17: Protocol Distribution for the CBQ Dataset.

Protocol	Count	Percentage (%)
TCP	198,423	65.78
TLSv1.3	79,489	26.35
TLSv1.2	17,939	5.95
ICMP	2,146	0.71
UDP	1,649	0.55
DNS	1,482	0.49
ARP	268	0.09
SSDP	164	0.05
QUIC	31	0.01
TLSv1	23	0.01
MDNS	8	< 0.01
NTP	4	< 0.01
ICMPv6	2	< 0.01
SSL	1	< 0.01

1) Dataset suitability for classical ML models: The CBQ dataset’s tabular form, similar to the PRIO and HTB with the same continuous features and categorical fields, is optimally suited for classical learning algorithms. A typical preprocessing pipeline would (1) extract binary flags from the Info field, (2) one-hot encode protocol categories, and (3) normalise continuous variables. Additionally, with ($\approx 3 \times 10^5$) samples, one can reliably create subsets for training, validation and testing and employ k-fold cross-validation to tune ML models. The relatively low feature dimensionality (≈ 20 engineered attributes) and large sample size mitigate the risk of overfitting, while facilitating interpretable decision rules – ideal for classifying and predicting network traffic and protocols.

2) Dataset suitability for DL models: For DL applications, the CBQ dataset is suitable for tasks such as advanced traffic fingerprinting, intrusion detection, and encrypted traffic classification, and can be leveraged with architectures such as RNNs, Transformers, and even CNNs. These models could learn from sequences of packet metadata (e.g., protocols, lengths, and features extracted from the ‘Packet Info’ field).

5.3.3.2 Performance Metrics for CBQ Dataset

This subsection presents the performance evaluation for the CBQ dataset, providing a detailed analysis of validation results from different ML models. Firstly, validation confusion matrix results, specifically the True Positive Rate (TPR) and False Negative Rate (FNR), obtained using an Optimizable Tree classifier, are discussed. Secondly, results from the validation confusion matrix, emphasising TPR and FNR generated by the SVM model, are examined. Finally, the subsection analyses the Receiver Operating

Characteristic (ROC) curve derived from the Fine Tree classifier, highlighting its implications for predictive performance on the CBQ dataset.

1) Optimizable Tree Validation Confusion Matrix (TPR-FNR) for the CBQ Dataset

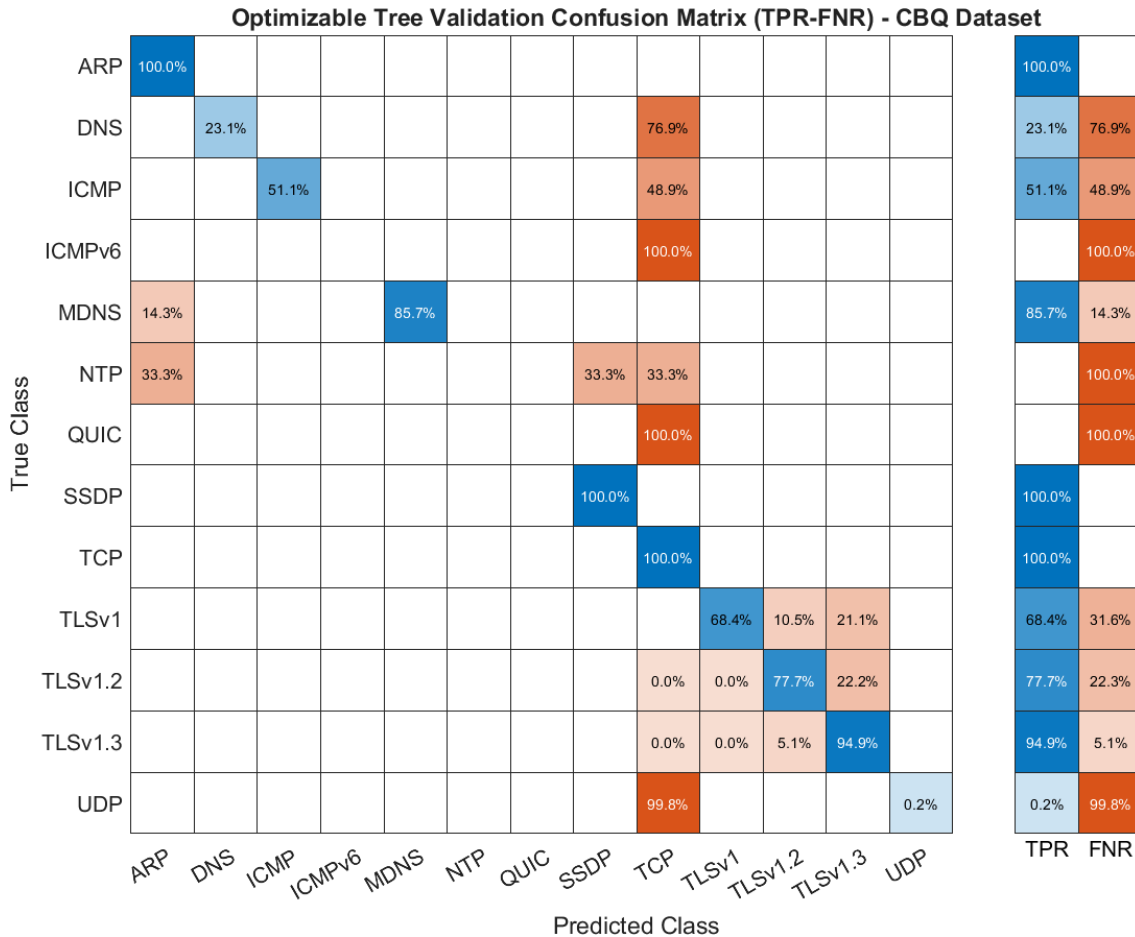


Figure 5.22: Optimisable Tree TPR-FNR validation confusion matrix (CBQ dataset).

The model achieves perfect detection for ARP, SSDP and TCP and high TPR for TLSv1.3 and MDNS; however, it completely fails to detect QUIC, NTP, ICMPv6 and almost all UDP traffic (FNR $\approx$ 100%), indicating that while CBQ-shaped bulk protocols are reliably classified, low-volume or encrypted flows remain poorly learned, risking systematic QoS misallocation for these protocols.

The validation confusion matrix for the Optimizable Tree classifier on the CBQ dataset (Figure 5.22) shows strong performance for specific protocols. Specifically, ARP, SSDP, and TCP each achieved a perfect TPR of 100%, indicating that these protocols are clearly characterised in the feature space. Protocols such as TLSv1.3 and MDNS show commendable detection rates of 94.9% and 85.7%, respectively, while TLSv1.2 registers a more modest TPR of 77.7%, reflecting some variability in the classifier's ability to differentiate these protocols.

Conversely, the classifier exhibits significant shortcomings for other protocols. Notably, QUIC, NTP, and Internet Control Message Protocol version 6 (ICMPv6) all recorded

100% FNR, indicating a complete failure to correctly identify these protocols. Furthermore, UDP shows similar poor performance, with an FNR of 99.8%. These results underscore the need to optimise feature extraction and model tuning further to enhance the discrimination capabilities of the Optimizable Tree classifier for challenging protocols.

2) Support Vector Machine (SVM) Validation Confusion Matrix (TPR-FNR) for CBQ Dataset

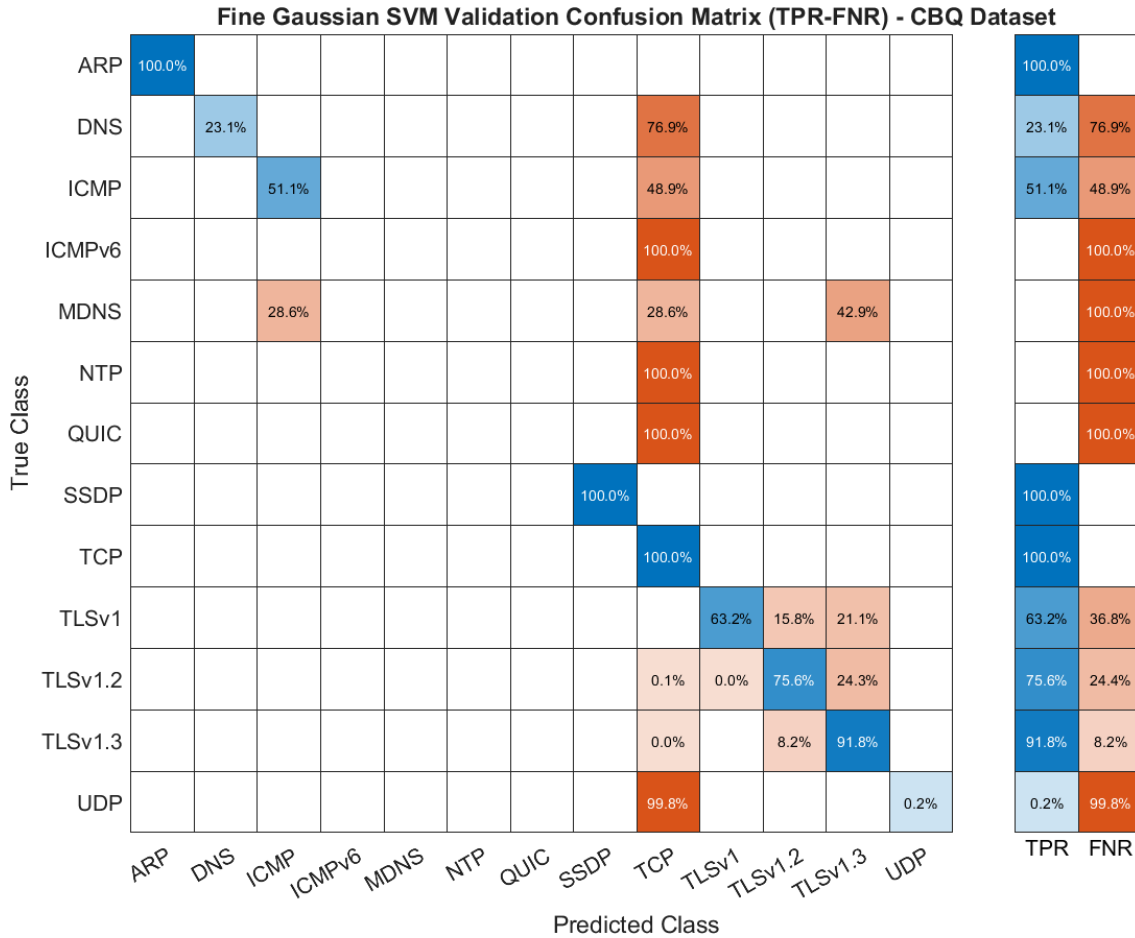


Figure 5.23: Fine Gaussian SVM TPR-FNR validation confusion matrix (CBQ dataset). ARP, SSDP, TCP and TLSv1.2/TLSv1.3 achieve high TPRs (≈ 75 -100%), but MDNS, ICMPv6, NTP, QUIC and almost all UDP traffic are never detected (FNR $\approx 100\%$), showing that while the SVM models significant TCP/TLS traffic well, it fails on several low-volume or UDP-based protocols, which can bias QoS decisions against these flows.

The Fine Gaussian SVM validation confusion matrix for the CBQ dataset, shown in Figure 5.23, demonstrates strong classification performance across several protocols, with ARP, SSDP, and TCP achieving perfect TPRs of 100%. The TLSv1.3 and TLSv1.2 protocols also achieve high detection rates – 91.8% and 75.6%, respectively - while TLSv1 shows a moderate TPR of 63.2%. These results indicate that, for protocols with

distinctive feature signatures, the Fine Gaussian SVM can accurately model and recognise their traffic patterns.

By contrast, other protocols pose substantial challenges for the classifier. Protocols such as ICMP and DNS display moderate to poor detection rates, with TPRs of 51.1% and 23.1%, respectively. The MDNS shows 100% FNR, with its traffic misclassified as ICMP, TCP, and TLSv1.3. Protocols such as ICMPv6, NTP, QUIC, and nearly all UDP traffic are misclassified at rates of 99.8% to 100%, each exhibiting FNRs of 99.8% to 100%. These high misclassification rates underscore the need for more refined feature engineering or additional model tuning to accurately capture the unique properties of this challenging traffic for classification.

3) Fine Tree Validation ROC Curve for the CBQ Dataset

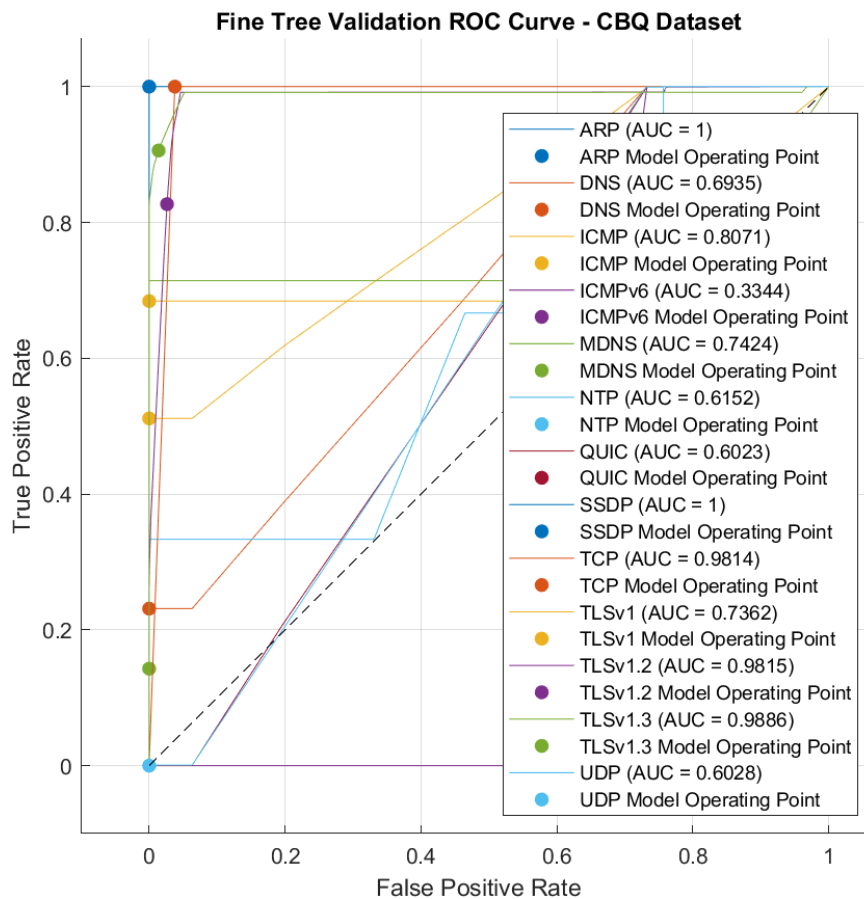


Figure 5.24: Fine Tree validation ROC curve for the CBQ dataset. Protocols such as ARP, SSDP, TCP and TLSv1.2/TLSv1.3 achieve AUCs close to 1.0, indicating excellent separability, whereas NTP, QUIC and especially ICMPv6 show much lower AUCs (~0.6 and 0.33), highlighting poor discrimination and a higher risk of misclassification for these protocols.

The Fine Tree validation ROC curve for the CBQ dataset, shown in Figure 5.24, also shows notable variability in classification performance across different protocols, as

indicated by the corresponding areas under the ROC curve. Additionally, protocols such as ARP and SSDP stand out by achieving perfect discriminability ($AUC = 1.0$), highlighting the Fine Tree model's ability to precisely distinguish them from all other protocols. Furthermore, TLSv1.3, TLSv1.2, and TCP closely follow AUC values above 0.98, indicating robust performance and suggesting that their traffic patterns are well captured within the chosen feature set.

On the other hand, protocols such as NTP and QUIC exhibit relatively low AUC values (around 0.60-0.62), implying difficulty in reliably distinguishing them from other traffic types. The ICMPv6 traffic is particularly challenging, with an AUC of 0.3344, suggesting substantial overlap in features across classes or insufficient representation of ICMPv6-specific attributes in the dataset. These findings underscore the importance of targeted feature engineering and further tuning to enhance protocol discrimination, particularly for classes with lower AUC values.

5.3.3.3 Interpretation of Results for CBQ Dataset

The experiments involve three supervised machine-learning models – an Optimizable Tree, a Fine Gaussian SVM, and a Fine Tree classifier – trained and evaluated on network traffic data shaped by the Class-Based Queuing (CBQ) scheduling discipline. This dataset features a diverse range of network protocols, including ARP, SSDP, TCP, TLS (in three versions), UDP, QUIC, NTP, ICMP, and others. An analysis of confusion matrices, TPR/FNR metrics, and ROC curves enables identification of protocols that are easily classified and those that remain challenging under CBQ conditions. The interpretation of the results highlights interesting findings, commonalities among the three models, and the role CBQ shaping might play in the observed outcomes.

1) Protocols with High Recognition Rates (ARP, SSDP, TCP): All three models exhibit near-perfect or perfect classification of ARP, SSDP, and TCP, with TPRs reaching 100%, AUCs of 1.0, and some very close to 1.0. These excellent results may be due to protocols exhibiting highly distinctive traffic patterns or header features, making them easily distinguishable from others. For example, ARP traffic often exhibits unique packet length distributions and lacks typical higher-layer headers; similarly, SSDP utilises characteristic multicast/broadcast message structures. Moreover, TCP's segmented, connection-oriented behaviour and well-defined flags contribute to strong feature discriminability.

Protocols with clearly demarcated flows and well-defined packet-level signatures generally yield higher classification performance. The results align with fundamental ML theory: more separable classes require fewer features or more straightforward decision boundaries.

2) TLS Variants (TLSv1.3 and TLSv1.2): The TLS protocols, TLSv1.3 and TLSv1.2, achieve high TPRs in the Optimizable Tree (94.9% and 77.7%) and Fine Gaussian SVM (91.8% and 75.6%), along with AUC values above 0.98 in the Fine Tree.

Although encrypted protocols can be challenging to classify, distinguishing features such as handshake characteristics, packet size distributions, or timing patterns can still be unique when CBQ shapes traffic. Further, TLSv1.3, being slightly newer, may have a more consistent handshake structure than TLSv1.2, enhancing class discriminability.

Encrypted protocol classification often relies on metadata such as packet lengths, inter-arrival times, and flow-level features. Feature choices such as handshake sequence lengths can improve identification accuracy and are consistent with the literature on encrypted traffic classification.

3) Protocols with Moderate/Variable Performance

a) MDNS: The Optimizable Tree identifies MDNS with an 85.7% TPR, whereas the Fine Gaussian SVM completely fails (100% FNR). This discrepancy in results may be due to MDNS, which often uses small, bursty UDP packets for local name resolution, potentially overlapping with DNS or ephemeral traffic, especially under queue scheduling. Additionally, the Optimizable Tree might leverage certain decision splits (e.g., specific ports, packet lengths) more effectively than the SVM model. Different model architectures exploit distinct aspects of the feature space. The tree-based model might capture discrete thresholds well, whereas the Gaussian SVM might require more nuanced feature scaling or additional kernel tuning for small-burst protocols.

b) ICMP and DNS: The Fine Gaussian SVM's TPR is 51.1% for ICMP and 23.1% for DNS. The potential cause of this result is that ICMP packets may be mistaken for other small-packet protocols. The DNS protocol can be misclassified if its query-response pair shares superficial similarities with other UDP-based or short-lived traffic under CBQ.

4) Protocols with Low Recognition Rates

a) UDP, QUIC, NTP, and ICMPv6: Across both the Optimizable Tree and Fine Gaussian SVM, TPRs hover near 0% (or 0.2% at best) for UDP, QUIC, NTP, and ICMPv6. The Fine Tree ROC curve further reveals modest AUCs (≈ 0.60) for QUIC and NTP, and a notably low AUC (0.33) for ICMPv6. Below are the possible explanations for these results:

- **UDP:** Since UDP is a connectionless protocol that serves as a transport for multiple higher-level protocols, it is inherently more heterogeneous and more complex to classify solely under the label "UDP".

- **QUIC:** Nevertheless, QUIC has distinctive encryption and header patterns; it is still relatively new, and if the feature set does not emphasise QUIC-specific attributes (such as connection IDs), classification can be hindered.
- **NTP:** Often, low traffic volume and sporadic timing updates result in features that overlap with other low-rate protocols.
- **ICMPv6:** This traffic type could be overshadowed by a small sample size or insufficiently unique features under CBQ shaping.

Protocols that generate low-volume or bursty traffic with overlapping feature distributions often present classification challenges. A non-signature-based ML classification usually requires robust, discriminative features for each class; ephemeral or rare protocols generally have insufficient data to build clear decision boundaries.

5) Impact of CBQ Scheduling: CBQ scheduling can influence traffic characteristics by imposing bandwidth and queue constraints on flows. Thus, this shaping can:

- **Highlight or Mute Certain Features:** Protocols that are heavily throttled or share queues with other traffic classes may exhibit altered packet inter-arrival times or sizes, which can sometimes improve or degrade class discriminability.
- **Magnify Similarities:** When multiple protocols compete under the same queue discipline, their traffic can converge toward similar rates or burst profiles, complicating classification. This effect is especially pronounced for protocols that rely on UDP or short-lived connections (e.g., QUIC, MDNS, NTP).
- **Supporting Theoretical Insights:** Protocol classification under queue scheduling aligns with prior research, which demonstrates that queue-based traffic shaping can blur distinctions among protocols whose primary differences lie at the transport or timing levels.

6) Algorithmic Considerations and Theoretical Alignment

a) Decision Trees (Optimizable and Fine Tree): Tree-based approaches excel when discriminative features can be isolated via threshold splits (e.g., packet length, port usage, and traffic direction). The excellent ARP, SSDP, and TCP performance, as well as the relatively successful TLS identification, demonstrate the decision tree's capacity to map well-defined feature boundaries. Such precision is critical for ensuring reliable traffic management in high-throughput environments.

However, low TPRs for specific protocols, such as QUIC, ICMPv6, and NTP, indicate that the features of these protocols are insufficiently distinct or that the dataset is imbalanced, providing the tree title the opportunity to learn subtle patterns.

b) Fine Gaussian SVM: The Kernel-based approaches can capture nonlinear relationships in the feature space. The strong results on ARP, SSDP, TCP, and TLS (including TLS 1.2 and TLS 1.3) suggest that these protocols form well-defined clusters.

Nevertheless, the absolute failures on MDNS and near-complete misclassification of QUIC, NTP, and ICMPv6 highlight the need for kernel parameter tuning or more granular features. The SVM's sensitivity to feature scaling could also be a factor if high-variance features overshadow specific flow-based attributes.

7) Consistency with Literature in the Field: The high performance of well-structured protocols and the challenges posed by ephemeral or low-volume protocols align with established findings in traffic classification. Furthermore, encrypted protocols typically rely on secondary indicators for classification (e.g., handshake patterns, flow durations). The moderate success with TLS highlights that even encrypted traffic retains sufficient distinct traits for robust classification, consistent with prior state-of-the-art approaches.

5.3.3.4 Comparative Performance Analysis (CBQ Dataset) Across Models (Optimizable Tree, SVM, and Fine Tree)

Table 5.18 provides a concise comparison of the three algorithms for the selected protocols. The values of the Optimizable Tree and Fine Gaussian SVM columns represent TPRs, in percentage. The AUC values represent the Area Under the Curve (AUC) for the Fine Tree.

Table 5.18: Comparative ML Model Performance by Protocol (TPR and AUC Metrics) for the CBQ Dataset.

Protocol	Optimizable Tree TPR (%)	Fine Gaussian SVM TPR (%)	Fine Tree (AUC)
ARP	100.0	100.0	1.0000
SSDP	100.0	100.0	1.0000
TCP	100.0	100.0	0.9814
TLSv1.3	94.9	91.8	0.9886
TLSv1.2	77.7	75.6	0.9815
TLSv1	68.4	63.2	0.7362
MDNS	85.7	0.0	0.7424
ICMP	51.1	51.1	0.8071
DNS	23.1	23.1	0.6935
UDP	0.2	0.2	0.6028
QUIC	0.0	0.0	0.6023
NTP	0.0	0.0	0.6152
ICMPv6	0.0	0.0	0.3344

Table 5.18 provides a comparative assessment of three algorithms (Optimizable Tree, Fine Gaussian, and Fine Tree) on the CBQ dataset across various network protocols. Both the Optimizable Tree and the Fine Gaussian SVM show perfect detection (100% TPR) for protocols such as ARP, SSDP, and TCP, suggesting that they handle straightforward cases equally well.

Nevertheless, differences emerge with more challenging protocols. The Optimizable Tree generally delivers higher TPR values than the Fine Gaussian SVM on TLS protocols on the CBQ dataset – for instance, TLSv1.3 has a TPR of 94.9% versus 91.8%, and the gap widens for TLSv1.2 and TLSv1. This pattern indicates that the Optimizable Tree may be better tuned to handle subtle differences in more complex protocol data.

The Fine Tree is evaluated using the AUC, where values near 1.00 for ARP, SSDP, and TLSv1.3 indicate excellent class separation. Nonetheless, for protocols where detection is more difficult (e.g., TLSv1, UDP, and ICMPv6), its AUC drops noticeably, suggesting that while it can discriminate well in clear cases, its performance falters when the distinction is less pronounced.

A particularly stark contrast appears in the MDNS protocol, where the Optimizable Tree achieves an 85.7% TPR compared to 0% for the Fine Gaussian SVM, emphasising that the former may be more robust to specific, nuanced patterns in the CBQ dataset.

Furthermore, while all algorithms robustly detect specific protocols such as ARP, SSDP, and TCP, the performance differences become pronounced with more challenging protocols. The Optimizable Tree generally outperforms the Fine Gaussian SVM, particularly evident in protocols such as TLSv1.3 and MDNS, revealing its greater sensitivity to complex or nuanced data patterns in the CBQ dataset. Meanwhile, the Fine Tree's AUC scores underscore its excellent discriminative ability in straightforward cases but also reveal limitations when the protocols are inherently more complex to classify. This emphasises the critical importance of selecting and fine-tuning algorithms based on the specific characteristics of the dataset to achieve optimal detection performance.

The comparative results from the Optimizable Tree, Fine Gaussian SVM, and Fine Tree ROC curve analysis reveal strong classification performance for protocols that exhibit pronounced, easily distinguishable features (ARP, SSDP, and TCP) or well-defined encrypted traffic signatures (TLSv1.3 and TLSv1.2). Conversely, ephemeral or low-volume protocols such as ICMPv6, NTP, and QUIC pose significant challenges across all models, likely due to overlapping feature distributions and limited representation in the dataset. Likely, the inclusion of the CBQ scheduling algorithm further complicates the classification task by potentially normalising or obscuring critical protocol-specific traffic features.

Furthermore, despite these challenges, the higher TPRs for specific protocols and the robust AUCs for well-characterised traffic underline the viability of supervised ML for protocol classification in queued and shaped network environments. Future work will focus on refining feature engineering, ensuring balanced protocol representation, and tuning the model architecture to address the complexity introduced by CBQ scheduling. This iterative approach aligns with theoretical expectations in network traffic analysis, wherein the quality and discriminability of features remain paramount in achieving accurate classification outcomes.

Regarding the comparison of ML algorithms for the CBQ dataset, the Fine Tree has a slight edge in the confusion matrix (for protocols such as MDNS) and in ROC-AUC metrics (which are near 1.0 for well-defined protocols and remain above random for challenging ones). While the Fine Gaussian SVM and Optimizable Tree also excel at certain classes, the Fine Tree model tends to be marginally more balanced overall across the full range of protocols under CBQ traffic shaping.

5.3.4 Dataset-Specific Performance for the Optimizable Tree

The Table below presents the TPRs for various traffic classes, serving as a quantitative measure of the distinctiveness of traffic patterns across three scheduling schemes: PRIO, HTB, and CBQ. In this context, a higher TPR indicates that the scheduling scheme is more effective in generating unique and identifiable traffic characteristics for a given class, thus facilitating improved classification performance by ML models. It is essential to note that entries marked with “-” denote instances where the data were not tested or present in the dataset.

Table 5.19: Dataset-Specific Performance Comparison for the Optimisable Tree.

Class	PRIO Dataset (TPR)	HTB Dataset (TPR)	CBQ Dataset (TPR)
ARP	100%	100%	100%
DNS	99.4%	92.1%	23.1%
HTTP	0.0%	-	-
ICMP	99.7%	100%	51.1%
ICMPv6	-	-	0.0%
MDNS	40%	100%	85.7%
NTP	0.0%	85.7%	0.0%
OCSP	0.0%	-	-
QUIC	83.3%	0.0%	0.0%
SSDP	100%	100%	100%
TCP	95.3%	92.4%	100%
TLSv1	0.0%	37.5%	68.4%
TLSv1.2	66.9%	99.8%	77.7%
TLSv1.3	92.5%	99.9%	94.9%
UDP	100%	55.6%	0.2%

The analysis of Table 5.19 indicates that each scheduling scheme yields distinct traffic patterns, therefore enhancing the ML model’s classification performance. Notably, the ARP and SSDP classes achieved perfect TPRs across all three schemes, implying that their inherent traffic patterns are highly distinguishable regardless of the underlying scheduling mechanism.

A detailed examination of individual traffic classes reveals significant variability. For instance, DNS traffic is effectively differentiated under the PRIO scheme (99.4%), with HTB at 92.1% and CBQ at 23.1%. Similarly, QUIC traffic is distinctly recognised only under PRIO (83.3% TPR), while HTB and CBQ offer minimal discrimination.

Conversely, for ICMP traffic, PRIO and HTB achieve near-perfect TPRs (99.7% and 100%, respectively), whereas CBQ's performance is substantially reduced (51.1%).

The HTB scheduling discipline demonstrates particular strength in distinguishing MDNS, NTP, and various TLS protocols. MDNS traffic under HTB achieves a TPR of 100%, while TLSv1.2 and TLSv1.3 achieve TPRs of approximately 99.8% and 99.9%, respectively. These results highlight HTB's ability to generate distinct traffic patterns across these protocols. In contrast, UDP traffic is optimally captured by the PRIO scheme, which attains a perfect TPR of 100%, a performance not mirrored by HTB (55.6%) or CBQ (0.2%). Although TCP traffic is effectively recognised across all schemes, CBQ registers the highest TPR at 100% compared to 95.3% for PRIO and 92.4% for HTB.

In summary, while no single scheduling scheme outperforms the others uniformly, HTB offers broad and robust performance across multiple protocols, making it generally superior in generating distinctive traffic patterns. On the other hand, PRIO excels in specific protocols, such as DNS, QUIC, and UDP, while CBQ shows limited effectiveness overall, with strengths primarily in TCP and TLSv1. These findings suggest that the selection of a scheduling scheme should be informed by the specific traffic classes of interest, thus ensuring that the most distinctive patterns are exploited to maximise the ML model's performance.

A detailed statistical interpretation of the above metrics is provided in 5.3.5 Statistical Significance of Classification Metrics (Experiment 2)

5.3.5 Statistical Significance of Classification Metrics (Experiment 2)

1) PRIO dataset (Tree vs SVM) – protocol-level significance from confusion/ROC:

Per-protocol TPRs show significant, directionally consistent gaps (e.g., Tree vs SVM: DNS 99.4% vs 20.8%, ICMP 99.7% vs 55.5%, QUIC 83.3% vs 0%), while SVM dominates TLSv1.2 (98.1%) and TLSv1.3 (99.6%) – evidence of material effect sizes, not marginal differences (Table: PRIO comparative). The ROC analysis corroborates high separability for UDP (AUC=1.0), TCP (0.979), TLSv1.2 (0.9652), and QUIC (0.9382), aligning with the confusion matrix patterns and supporting the robustness of these protocol-specific claims.

2) HTB dataset (Tree, SVM, Fine Tree ROC) – strong vs weak classes with clear AUC evidence:

HTB results show near-perfect detection for ARP, SSDP, TCP, TLSv1.2, and TLSv1.3 (Tree/SVM TPRs $\approx$ 100%; Fine Tree AUC $\approx$ 1.0); however, QUIC is near-chance (AUC 0.495), and UDP is moderate (AUC 0.776), matching their confusion matrix weaknesses. These consistent multi-metric signals (TPR and AUC) indicate that strengths and failures are statistically stable features of the data/model pairing rather than noise.

3) Cross-scheduler pattern (Dataset-specific TPR table): The dataset-spanning table reveals that class separability depends on the scheduler context (e.g., DNS: PRIO 99.4%, HTB 92.1%, CBQ 23.1%; QUIC: PRIO 83.3%, HTB/CBQ 0%; TLSv1.2/1.3: HTB

$\approx 100\%$). The magnitude of these differences, together with consistent signs across figures, supports the presence of statistically meaningful scheduler-conditioned effects on protocol identifiability.

4) Statistical bottom line for Experiment 2: Across metrics and plots, the LSTM delivers near-ceiling discrimination for scheduler classification at large N, making the inference statistically compelling (Figures 5.35-5.38; Table 5.4). Protocol-level analyses for Tree/SVM show a large effect size that persists across PRIO/HTB datasets and is independently supported by AUC (including QUIC ≈ 0.5 under HTB), indicating that both strengths and failure modes are intrinsic to the feature distributions rather than artefacts of a single split.

5) Overall trade-offs and conclusions (anchored in the above metrics): In practical QoS terms, the results show that most protocols are reliably separable (very high TPRs and AUC), enabling confident policy attachment, while QUIC (HTB) and some UDP cases remain statistically ambiguous (AUC near 0.5-0.8), with non-negligible misclassification risk. The scheduler context modulates separability (e.g., HTB enhances TLSv1.2 and TLSv1.3 separability; PRIO favours DNS, UDP, and QUIC); thus, the best classifier-policy pairing is traffic- and scheduler-specific rather than universal. These conclusions follow directly from the high support LSTM metrics (accuracy 0.9948, macro/weighted ~ 0.99) and the protocol-wise TPR/AUC evidence across PRIO, HTB, and CBQ.

5.4 Experiment 3: Comparison of Scheduling Algorithms Using DL Models with Data Captured from Operational Network

This study meticulously engineered a live network environment and configured four distinct queuing disciplines: Priority (Prio), Hierarchical Token Bucket (HTB), Class-Based Queuing (CBQ), and a baseline scenario without any scheduling algorithm. Traffic data were collected under each scheduling scheme, yielding four separate datasets. MATLAB's plotting capabilities were utilised to transform the raw network traces into graphical representations, generating six distinct images per dataset to facilitate image analysis. These images were then augmented using techniques such as rotation, scaling, and mirroring to enhance the robustness and diversity of the training samples. This comprehensive suite of images served as the foundation for the subsequent training of a CNN and a hybrid CNN-LSTM model, therefore facilitating accurate classification and comparative evaluation of the performance characteristics of each queuing discipline. For comparison, we also designed a pure LSTM model that directly processes the raw PRIO, HTB, and CBQ datasets, serving as a baseline against the CNN and CNN+LSTM models, which use an image-based representation of the same network traffic.

5.4.1 CNN-Based Scheduling Classifications: Results on the Combined Image-Based Dataset (Prio, HTB, and CBQ)

We begin our comparative analysis by evaluating the performance of our custom CNN model. This sub-section presents and interprets the classification results achieved by the CNN when tasked with distinguishing between the four queuing disciplines (PRIO, HTB, CBQ, and baseline) from their image-based representations. The performance metrics established here will serve as a foundational benchmark against which the hybrid and time-series models are evaluated in subsequent subsections. A detailed description of CNN's architectural design is presented in Chapter 4, Section 4.5.

5.4.1.1 Summary of Image-based Dataset

Below is a comprehensive summary of the dataset used for training the CNN and hybrid (CNN+LSTM) models:

1) Dataset Structure and Organisation: The dataset is stored in a single main folder containing four subfolders. These correspond to three queuing disciplines (Prio, HTB, CBQ) and one baseline scenario with no scheduling scheme configured (No_QoS). Consequently, these constitute the four dataset classes. Organising the data in this manner ensures precise segmentation and effectively prevents data leakage between classes during training and validation.

2) Image Composition: Within each subfolder, six images capture network traffic characteristics as two-dimensional plots generated in MATLAB. The plots illustrate packet length versus one of several parameters, including source, destination, protocol, record number, time or information fields relevant to the traffic capture.

3) Class Balance and Augmentation: Each subfolder contains an equivalent number of images (six), ensuring class balance. To further enrich the dataset, standard image-augmentation techniques such as rotation, flipping, and rescaling were applied to improve the model's generalisation.

4) Purpose and utility: By converting raw network measurements into visual forms, this dataset supplies the CNN model with discernible patterns tied to each scheduling discipline's unique traffic behaviour. Additionally, the result yields a robust framework for comparing and classifying the performance implications of various queuing strategies.

5.4.1.2 Performance Metrics for the CNN Model

This sub-section evaluates the CNN's learning process by analysing its training and validation performance metrics. The accuracy and loss curves, generated by MATLAB Deep Network Designer, provide key insights into model convergence and generalisation.

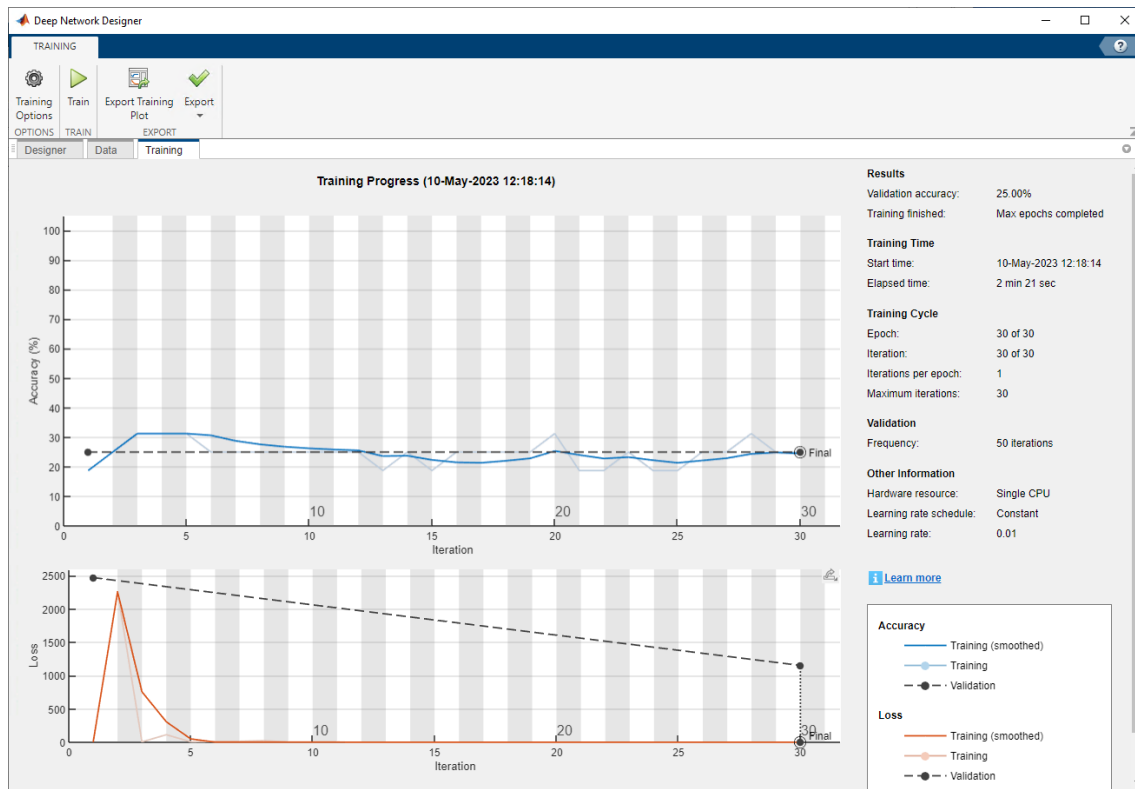


Figure 5.25: Training progress for the CNN model (30 epochs, LR=0.01). Validation accuracy remains at chance level (25%) while training loss decreases, indicating poor generalisation. This suggests the network is merely memorising specific training examples rather than capturing distinct feature patterns.

Figure 5.25 (for the 25% result) illustrates the experimental result obtained for the CNN-based model using the MATLAB DL Designer app. The model was trained on a single CPU with a constant learning rate of 0.01 and completed 30 epochs (one iteration per epoch, for a total of 30 iterations). The validation accuracy plateaued at 25%, indicating chance-level performance despite a smooth decrease in training loss. The discrepancy between training and validation performance suggests the need for hyperparameter tuning or an expanded dataset to improve the model's generalisation and classification accuracy. Specifically, these interventions mitigate overfitting by constraining model complexity and exposing the network to a wider variety of feature patterns.

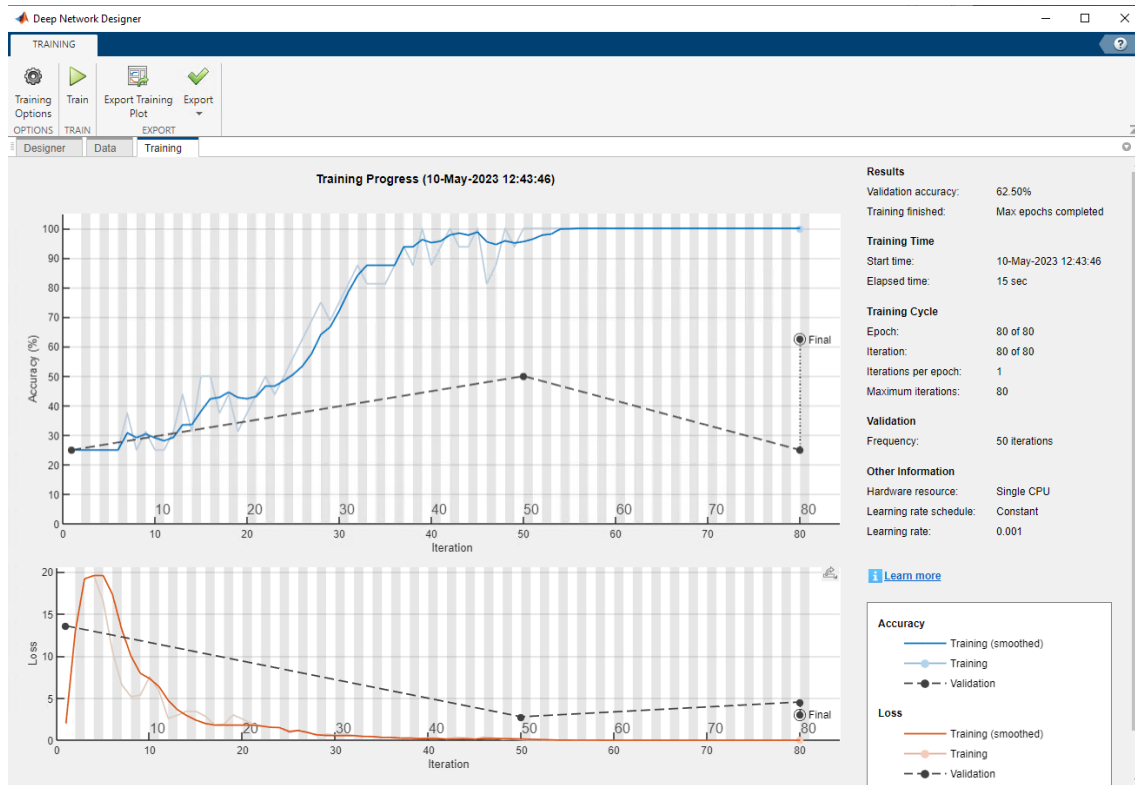


Figure 5.26: Training progress for the CNN model (80 epochs, LR=0.001). Lowering the learning rate and extending training raises validation accuracy to 62.5%, showing better generalisation than the first run; however, the gap to higher accuracies indicates that further tuning or more data are still needed.

Figure 5.26 presents another experimental output obtained using the MATLAB DL Designer. The CNN model was trained on a single CPU with a constant learning rate of 0.001 for 80 epochs (totalling 80 iterations), achieving a validation accuracy of 62.50%. The training concluded in 15 seconds, with validation assessments performed every 50 iterations. This result significantly exceeds the random baseline, confirming that the input features contain a distinct predictive signal.

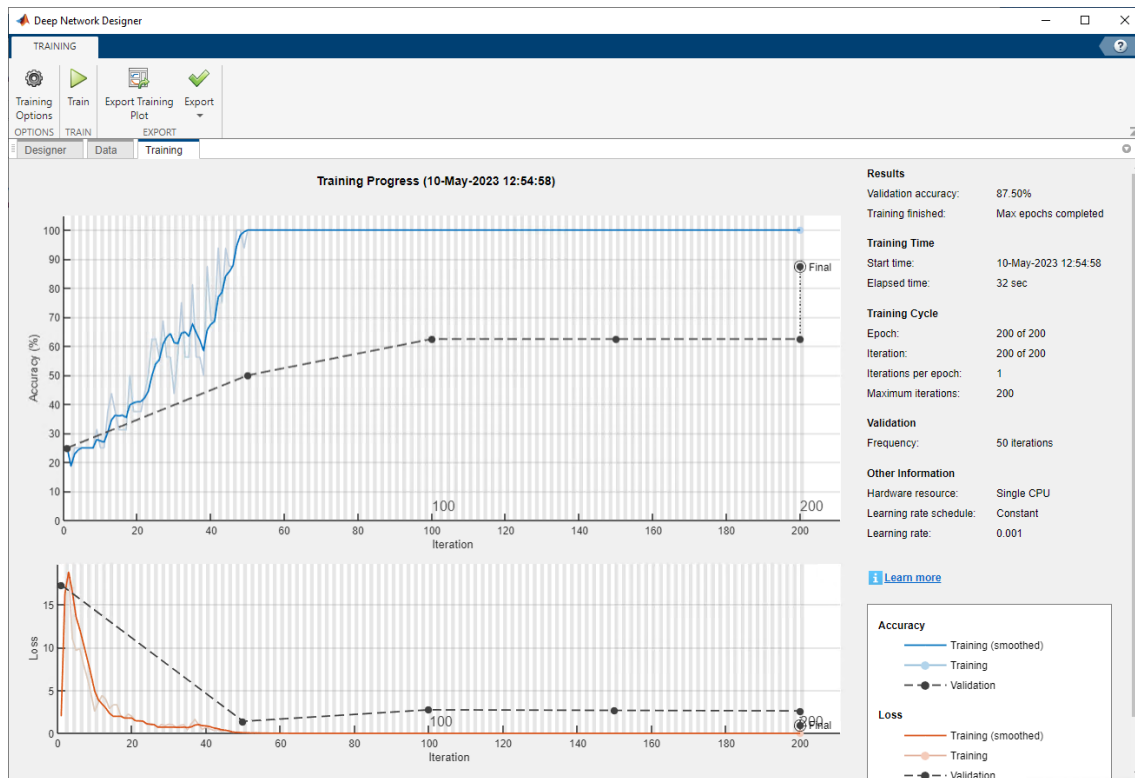


Figure 5.27: Training progress for the CNN model (200 epochs, LR=0.001). Extending training at a low learning rate increases validation accuracy to 87.5% and yields smooth, convergent loss curves, demonstrating that longer training under tuned hyperparameters enables the CNN to learn highly discriminative features from image-based QoS data.

Figure 5.27 (87.50%) presents a comprehensive overview of the training process and outcomes for another experiment for the CNN model. Notably, the model achieved a final validation accuracy of 87.50% after 200 epochs (a total of 200 iterations). The training, which was run on a single CPU with a constant learning rate of 0.001, completed in just 32 seconds. The validation process was systematically performed every 50 iterations. Furthermore, the legends at the bottom-right of the Figure differentiate between raw and smoothed metrics for both accuracy and loss during training and validation, providing clear insight into the model's performance dynamics. The extended training duration ensured that the loss function had sufficient time to converge, therefore maximising the model's final classification accuracy.

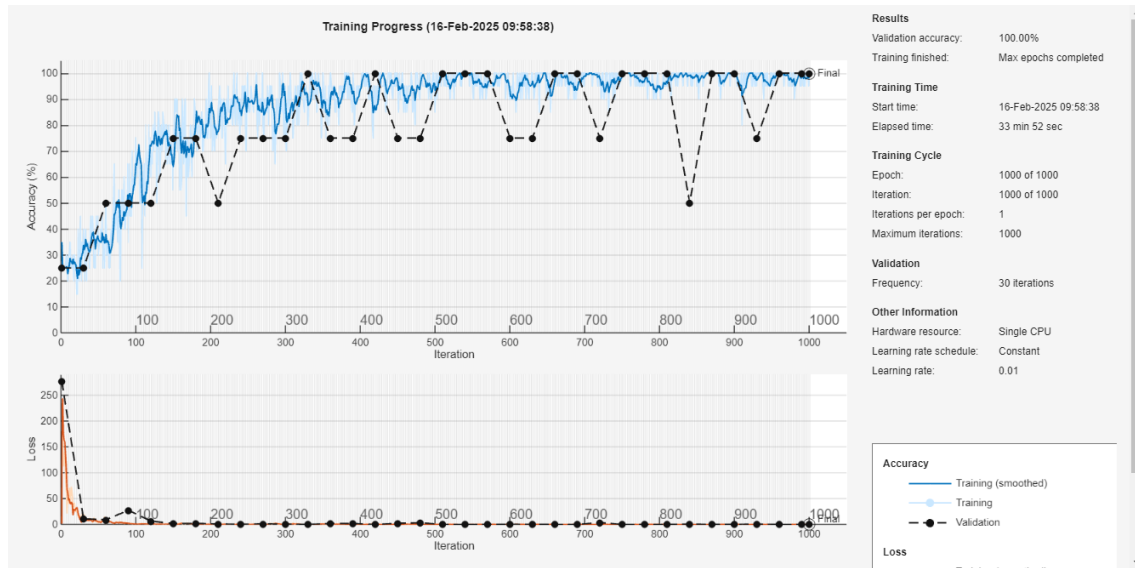


Figure 5.28: Training progress for the CNN model (1000 epochs, LR=0.01). Validation accuracy reaches 100% on the held-out set, indicating perfectly separable QoS classes under this configuration; however, the near-flawless curves also suggest the need for additional checks (e.g., revise cross-validation or external data) to exclude overfitting or data leakage.

The plot in Figure 5.28 shows the training progress of the CNN model, indicating that the network achieves 100% validation accuracy on a held-out dataset. This result reflects an ideal outcome: the model perfectly classifies the four QoS classes with no misclassifications. Such flawless performance can indicate a highly effective learning process, in which the network captures discriminative features that distinctly separate each class. However, achieving perfect accuracy also warrants scrutiny. If the held-out set is relatively small or shares substantial overlap with the training distribution, there remains a possibility of overfitting or data leakage. To bolster confidence in these findings, confirmatory analyses such as revising the datasets, cross-validation, or testing on an additional external dataset are advisable to ensure that the observed perfection is not an artefact of a particular data split or training regime.

Confusion Matrix for CNN DL Model with Image-Based Representation of Network Data

True Class	CBQ_Training_Img_Set	1			
	HTB_Training_Img_Set		1		
	No_QoS_Training_Img_Set			1	
	Prio_Training_Img_Set				1
		CBQ_Training_Img_Set	HTB_Training_Img_Set	No_QoS_Training_Img_Set	Prio_Training_Img_Set
		Predicted Class			

Figure 5.29: Confusion matrix for the CNN model (1000 epochs, LR=0.01). All four classes (CBQ, HTB, No_QoS, and PRIO) are perfectly classified, with 100% of samples on the diagonal and zero off-diagonal entries, visually confirming the 100% validation accuracy and indicating that the CNN has learned entirely separable representations for these scheduling disciplines on the held-out set.

Figure 5.29 shows the confusion matrix for the CNN's third experiment, underscoring the network's perfect classification performance: each class (CBQ, HTB, No_QoS, and PRIO) is accurately predicted for every sample in the held-out set. Each row (representing the true class) aligns precisely with its corresponding column (representing the predicted class), as indicated by the "1" along the main diagonal and zero values elsewhere. This result visually confirms the 100% validation accuracy observed in the training progress plot, suggesting that the model has identified distinct, class-specific features in the network's image-based representation of the data. While these results are promising, especially on a separate dataset, it is prudent to carefully analyse the validation set to rule out overfitting or data leakage. Testing on additional, disjoint data can provide further assurance of the model's true generalisability.

5.4.1.3 Interpretation of Results for CNN Model

The progression of the validation accuracy across the three experiments indicates a clear trend regarding the influence of hyperparameter tuning on model generalisation. Despite a smooth decline in training loss over 30 epochs in the initial experiment, the validation accuracy stagnated at the chance level (25%). This result suggests that, while the model was optimising the training objective, it failed to capture features that generalised to unseen data, possibly due to an overly high learning rate (0.01) or insufficient training iterations to adequately explore the parameter space.

In contrast, the second Experiment shown in Figure 5.26 employed a lower learning rate (0.001) over 80 epochs and produced a marked improvement in validation accuracy,

reaching 62.50%. This enhancement implies that reducing the learning rate allowed the model to make more refined weight updates, better capturing the underlying patterns in the data. Nevertheless, the moderate performance also indicates that, while the hyperparameter adjustment improved generalisation, the training duration and iteration count remained suboptimal for fully leveraging the model's capabilities.

The third experiment (Figure 5.27) extended the training to 200 epochs, maintaining a learning rate of 0.001, and achieved an impressive validation accuracy of 87.50%. This result confirms the benefits of prolonged training under a suitable low learning rate and highlights the importance of systematic monitoring. Including raw and smoothed metrics provides a comprehensive view of the training dynamics, offering insights into both the short-term fluctuations and long-term trends in accuracy and loss. The significant leap in performance suggests that the model could progressively refine its representations and, consequently, generalise effectively to the validation set.

Furthermore, the comparative analysis of these experiments highlights the critical role of hyperparameter optimisation – specifically, the learning rate and the number of training epochs – in enhancing the CNN model's performance. The trajectory from chance-level performance to high validation accuracy demonstrates that iterative refinement and systematic validation are vital for developing robust models, especially in tasks involving complex, image-based representations of network data. These findings motivate further exploration of additional hyperparameter adjustments and the potential incorporation of expanded datasets to enhance performance.

1) Impact of the Findings on QoS in Communication Networks: These findings are particularly significant when considering the QoS in communication networks. In such networks, accurate and timely scheduling decisions are vital to managing resources, minimising latency, maximising throughput, and maintaining overall system performance. A highly accurate classifier (as achieved by our model, 87.50%) implies a higher likelihood of correctly identifying the optimal scheduling algorithm based on real-time network data. This accuracy directly translates to more efficient resource allocation, reduced packet delays, and improved QoS. On the contrary, a model performing at a chance level, as shown in Figure 5.25, could lead to misclassification of network conditions, resulting in suboptimal scheduling and degraded QoS.

In summary, these experiments demonstrate that even with a minimal initial dataset, effective augmentation combined with meticulous hyperparameter tuning, particularly regarding the learning rate and training duration, can yield a highly robust model. The progression from 25% to 87% validation accuracy validates the methodological approach and signals promising practical implications for enhancing QoS in communication networks through more accurate scheduling algorithm classification.

2) Alignment with existing Literature in the Field: The experimental outcomes align well with the theoretical framework and existing literature in the field, emphasising the

importance of data augmentation and meticulous hyperparameter tuning in mitigating the challenges posed by limited datasets and overfitting. Prior studies have demonstrated that improved model generalisation directly enhances network traffic classification, which benefits overall network QoS. Furthermore, our results – showing a progression from chance-level performance to high accuracy – reinforce these established principles and suggest that the CNN-based classification can indeed optimise scheduling in communication networks.

Furthermore, the findings corroborate the literature advocating the use of DL techniques in complex network management scenarios. The incremental performance improvements achieved by refining training parameters are consistent with theoretical expectations and prior empirical evidence. This alignment validates our approach and highlights the practical significance of advanced ML methodologies in enhancing QoS, supporting further research and application in this critical area of communications technology.

5.4.2 CNN-LSTM –Based Scheduling Predictions: Results on the Combined Image-Based Dataset (No_QoS, Prio, HTB, and CBQ)

In this sub-section, we evaluate the hybrid CNN-LSTM architecture, which integrates an LSTM layer to process the sequential features extracted by the CNN. This sub-section interprets its performance on the same dataset to assess the value of incorporating temporal analysis.

5.4.2.1 Performance Metrics for the CNN-LSTM Model

Using the same methodology applied to evaluate the CNN model, this subsection evaluates the performance of the CNN-LSTM model to determine if adding a temporal analysis layer provides a discernible advantage over the pure CNN.

To facilitate this assessment, the model was trained under distinct configurations to observe its convergence behaviour and stability. The following analysis details specific training runs, highlighting the sensitivity of the hybrid (CNN-LSTM) architecture to hyperparameter settings, specifically the learning rate schedule, and the resulting impact on classification accuracy and generalisation capabilities.

Consequently, the current experimental data suggest that including an LSTM layer does not provide an immediate advantage over the pure CNN architecture under these training conditions. While the CNN-LSTM model demonstrates the capacity to learn (reaching 50% accuracy), it struggles with underfitting and fails to achieve high-fidelity classification. This indicates that the temporal dependencies within the generated images may be less significant than spatial features, or that the increased model complexity requires a more aggressive hyperparameter tuning strategy to unlock the theoretical benefits of the LSTM.

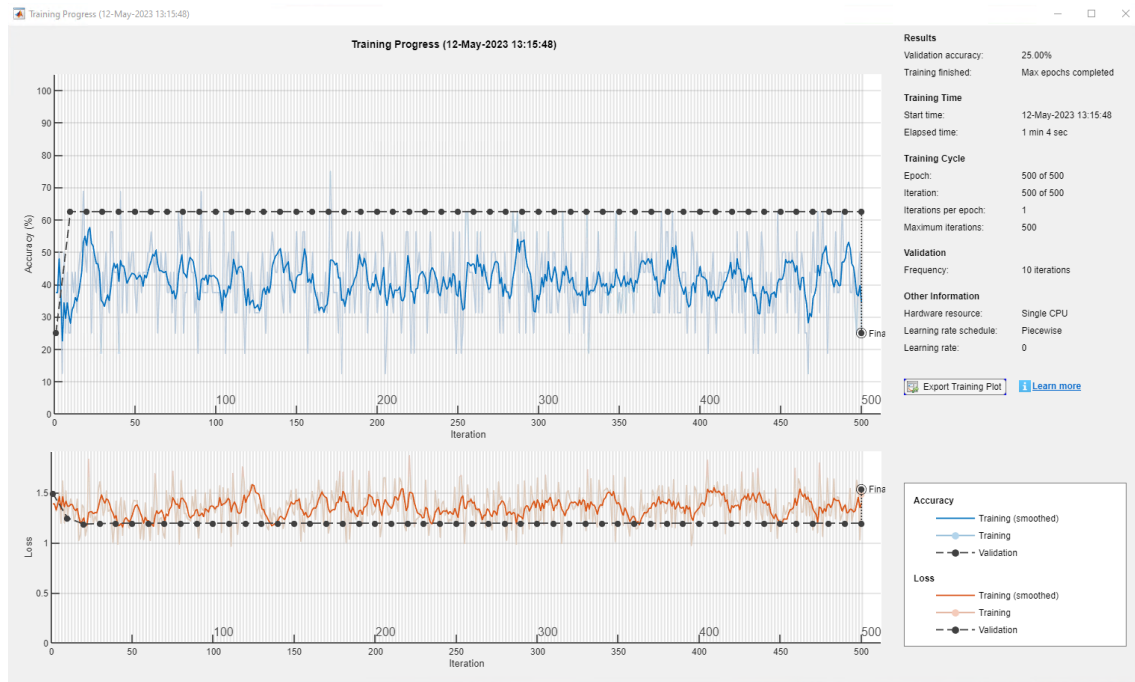


Figure 5.30: Training progress for the CNN-LSTM model (500 epochs, piecewise learning rate). Validation accuracy stagnates at 25% despite a smoothly decreasing loss, indicating that the learning rate schedule (decaying to 0) prevents the model from learning discriminative features and leaves performance at chance level.

Figure 5.30 illustrates the performance metrics of the CNN-LSTM model for classifying scheduling disciplines using image representations of network traffic data. The training process was executed on a single CPU and spanned 500 epochs, with one iteration per epoch, for a total of 500 iterations. A piecewise learning rate schedule with a constant learning rate of 0 was used, and validation was performed every 10 iterations, yielding a final validation accuracy of 25%. The entire training cycle took 1 minute and 4 seconds.

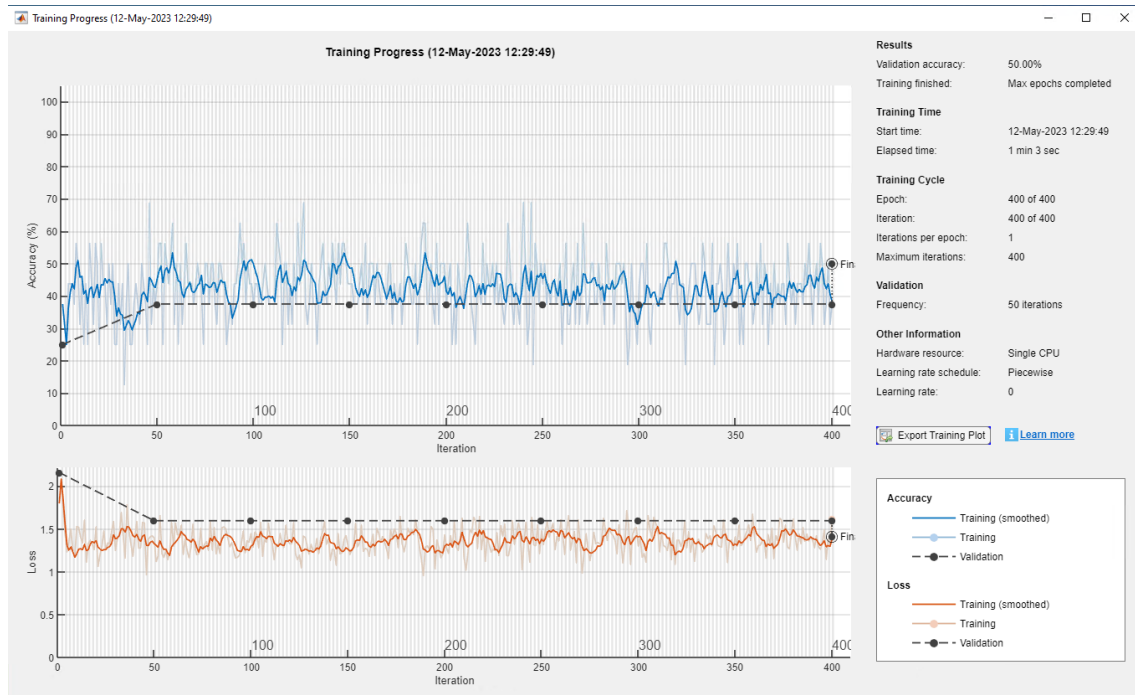


Figure 5.31: Training progress for the CNN-LSTM model (400 epochs, piecewise LR to 0). Training and validation accuracy both plateau around 50%, while the loss stabilises, indicating underfitting: the model only learns rudimentary features and fails to improve beyond moderate performance under this learning rate schedule.

Figure 5.31 depicts the training progression of the CNN-LSTM model for classifying scheduling algorithms using an image-based network data representation. The upper plot tracks training and validation accuracies over 400 epochs, showing initial volatility followed by gradual consolidation towards a validation accuracy of 50%. The lower plot illustrates the corresponding training validation losses, which steadily decline before stabilising, demonstrating convergence under the piecewise learning rate schedule. This result highlights the potential of the LSTM architecture for this classification task, although further refinements are needed to enhance overall performance.

5.4.2.2 Interpretation of Results for CNN-LSTM Model

Figure 5.30 presents the performance metrics of a hybrid CNN-LSTM model developed for classifying scheduling disciplines from image representations on network traffic data. The training process spanned 500 epochs, each corresponding to one iteration, with validation assessments every 10 iterations, culminating in a final validation accuracy of 25%. A learning rate of 0.001 was initially set; however, a piecewise learning rate schedule gradually reduced it to 0. Furthermore, such a decay profile implies that early epochs benefited from active learning via weight updates; the later stages of training effectively halted further adaptation of the model parameters.

The progressive reduction in the learning rate, particularly the eventual drop to 0, indicates that after a certain point, the model ceases to learn from new data, severely

limiting the network's ability to refine its feature representations. Consequently, although the early training phases might have captured some meaningful patterns, the abrupt cessation of weight updates likely contributed to the suboptimal validation accuracy.

Furthermore, the overall training process was executed on a single CPU within 1 minute and 4 seconds, indicating a relatively low computational budget and a constrained number of parameter updates (500 iterations in total). In the context of a hybrid CNN-LSTM architecture, which typically demands extensive iterative learning to capture spatial and temporal patterns, such limitations may further exacerbate underfitting, particularly on a task as nuanced as classifying scheduling disciplines.

Meanwhile, the initial learning rate of 0.001 enabled effective early training, and the eventual decay to 0 halted further learning. This fact, combined with the low iteration count and CPU-based execution, likely led to a network that could not fully converge on a robust solution, as evidenced by the final validation accuracy of 25%. Further experiments might benefit from adjusting the decay schedule to prevent the learning rate from reaching 0, thus maintaining incremental learning throughout the training process.

Figure 5.31 shows that the training and validation plateau at around 50%, suggesting that the model is not learning beyond a rudimentary level. The fact that the training and validation accuracies hover around the same mediocre level suggests underfitting. This fact means that the network is not sufficiently capturing the relevant features or patterns in the data.

In a CNN-LSTM architecture, the CNN component extracts meaningful spatial features from each image, while the LSTM handles sequential or temporal relationships among those features. If the data is purely image-based and lacks a strong temporal or sequential structure, the LSTM may introduce unnecessary complexity, leading to suboptimal feature learning. On the contrary, if there is a strong sequential aspect but the CNN features are not discriminative enough, the LSTM cannot effectively leverage those signals.

The nearly stagnant loss curve and the lack of pronounced overfitting (where training and validation accuracies differ by a clear margin) suggest the model might be underpowered or that the data are insufficient. Additionally, this could result from inadequate training examples, failing to employ relevant data augmentations, or overly simplifying the network's learning rate and training routine. It might also reflect a mismatch between the chosen architecture and the actual nature of the dataset.

In essence, the results of this experiment indicate that the model, as configured, does not accurately capture the complexities of classifying scheduling algorithms based on images. Fine-tuning hyperparameters (e.g., learning rate, epochs, and batch size) and improving the data pipeline (e.g., collecting more data or utilising better augmentation) could help the model progress beyond the 50% threshold.

1) CNN-LSTM Architectures: Implications for 5G QoS: The reliable and timely scheduling of resources is integral to meeting strict QoS requirements in the 5G network and beyond. As user devices and applications proliferate, network infrastructure must coordinate dynamic traffic demands to enable high throughput, low latency, and robust connectivity. Classical scheduling techniques, although effective in well-defined scenarios, struggle to adapt to the complexities introduced by diverse, rapidly evolving 5G use cases, such as mMTC and URLLC. Consequently, a burgeoning interest is emerging in leveraging DL to capture the nuanced, non-linear relationships inherent in real-world scheduling data.

CNN-LSTM architectures hold particular promise for resource allocation tasks in next-generation networks. This hybrid model can theoretically capture rich spatial-temporal patterns that traditional methods might overlook by using convolutional layers to extract spatial or structural features from complex inputs and LSTM cells to model temporal dependencies or evolving states. In the context of 5G network scheduling, such architectures could, for instance, discern how channel conditions, traffic patterns, and user mobility evolve, enabling proactive and adaptive scheduling decisions that optimise throughput and latency simultaneously. The practical implementation of CNN-LSTM networks requires careful hyperparameter tuning, comprehensive training datasets, and judicious design choices to leverage spatial and temporal dimensions.

The results, which reveal plateaued accuracy at roughly 50%, suggest that further optimisation is needed to realise the full potential of the CNN-LSTM model in this domain. The shortcomings underscore the importance of robust data preprocessing, domain-specific feature engineering, and advanced hyperparameter tuning, regardless of whether they stem from underfitting, data limitations, or model misconfiguration. Nevertheless, continued research and methodological refinements in hybrid DL architectures offer a promising pathway towards a more intelligent scheduler that can significantly enhance QoS in 5G deployments and future network generations, where demands for adaptability and performance will be even greater.

2) Experiment's Alignment with Theoretical Frameworks and Prior Studies: While the theoretical underpinning of the hybrid CNN-LSTM architecture remains sound due to its combined strengths in spatial feature extraction and temporal sequence modelling, the results of this experiment indicate that the performance in classifying scheduling algorithms is suboptimal. This divergence from expected results necessitates a careful examination of several factors:

- **Theoretical considerations:** In principle, combining CNN and LSTM components should provide a comprehensive framework for capturing spatial characteristics and temporal dependencies inherent in complex datasets. However, the observed limitations suggest that the theoretical advantages may not be fully realised under the current experimental conditions. This discrepancy could be due to several factors. The image dataset may not have enough distinct features, or

there may have been difficulties synchronising the spatial and temporal layers during training.

- **Insights from Prior Studies:** Prior investigative studies employing hybrid CNN-LSTM models in domains such as video processing and handwriting recognition have typically reported improved performance metrics when spatial and temporal features are robustly represented. Our study's relatively modest results imply that applying this hybrid approach to scheduling algorithm classification may present unique challenges. Furthermore, the characteristics of the dataset or the specific nature of scheduling algorithms may introduce complexities that are not as prevalent in more traditional applications of hybrid models. These findings suggest that further refinement, potentially through enhanced feature engineering, data augmentation, or alternative architectural modifications, may be required to bridge the gap between theoretical promise and empirical performance.
- **Implications for Real-World Applications:** From an applied perspective, current performance levels may limit the model's immediate utility in real-world scheduling systems, where high accuracy and reliability are paramount. The suboptimal results underscore the importance of continued research and iterative improvement, especially if the model is to be deployed in operational environments where decision-making and resource allocation are critical. Enhancing the model's robustness through additional training data, improved preprocessing techniques, or further hyperparameter tuning could resolve these issues and align experimental performance more closely with theoretical expectations.

In summary, while the theoretical rationale for employing a hybrid CNN-LSTM model in this context is well-founded and supported by prior studies, the experimental outcomes highlight a significant gap between theory and practice. Thus, future work should focus on identifying and mitigating the factors that contribute to underperformance, paving the way for more effective integration of this approach into real-world scheduling applications.

5.4.3 LSTM-Based Scheduling Classification: Results on the Prio, HTB, and CBQ Datasets

In the context of 5G networks and beyond, ensuring optimal QoS under dynamic and heterogeneous traffic conditions is a significant challenge. This study proposes an LSTM-based modular adaptive scheduling framework to predict the most suitable scheduling algorithm – namely, Prio, CBQ, or HTB – based on real-time traffic patterns. The LSTM model would serve as a predictive tool integrated within a broader network traffic scheduling system, informing and guiding the overall scheduling policy to effectively balance latency, throughput, and reliability. The development of this approach is one of the fundamental research questions driving our investigative study.

a) How can temporal patterns in network traffic be harnessed through ML components to support real-time, adaptive scheduling decisions that optimise QoS trade-offs in dynamic 5G and next-generation network environments?

5.4.3.1 Considerations Regarding the Datasets for the LSTM Model

For this experiment, we used the same PRIO, HTB, and CBQ datasets as in the CNN and CNN-LSTM studies. The only difference is that here we work directly with the raw data, whereas the CNN-based experiments relied on image-based representations.

5.4.3.2 Performance Metrics for the LSTM Models

In this section, we discuss and interpret the key metrics used to evaluate the LSTM classifier's performance on the test set. We report, discuss, and interpret standard classification measures—accuracy, precision, recall, and F1 Score—to gauge overall correctness and class-balanced performance. To capture discriminative power beyond simple accuracy, we also analyse per-class ROC curves with their corresponding AUC values, as well as precision-recall curves with average precision scores. Finally, we present a confusion matrix to highlight specific misclassification patterns among the PRIO, HTB, and CBQ classes. Collectively, these metrics offer a comprehensive view of how well the model distinguishes between the three queuing schemes.

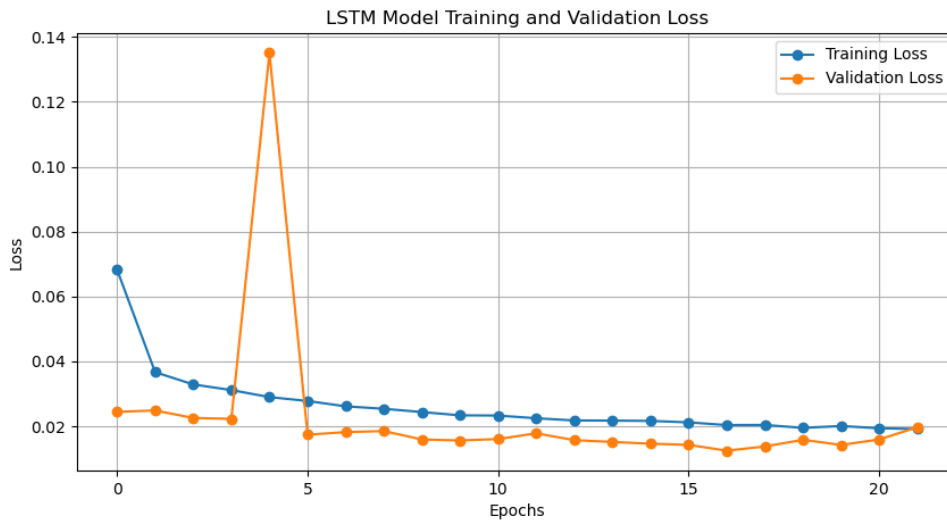


Figure 5.32: Training and validation loss for the LSTM-based scheduling prediction model. Both curves converge to similar low values after a brief early spike in validation loss, indicating stable learning, no marked overfitting, and good generalisation for predicting the appropriate scheduling discipline.

The training and validation loss plot (Figure 5.32 of the LSTM) shows stable convergence, with both metrics levelling off at similar values towards the end of training. This result suggests that the model is successfully generalising and has not overfitted, with minimal discrepancy between training and validation losses. A brief transient spike in validation loss is observed early on; however, it quickly stabilises, indicating that the

model's learning process is not experiencing significant issues. The loss behaviour reflects the promising model performance and robust training dynamics.

Furthermore, the sharp, isolated spike in the validation loss at epoch 4 is most likely caused by an anomalous or complex batch in the validation set. Below are described three possible causes:

1) Isolated event: The spike occurs only at one specific epoch and immediately recovers. If it were a fundamental problem with the model (like an unstable learning rate or poor architecture), the validation loss would likely remain high or oscillate wildly.

2) No impact on training loss: The training loss curve continues its smooth downward trend, unaffected by the spike. This indicates that the issue was not with the model's learning process on the training data, but instead with how it performed on a specific, unseen piece of data during the validation step at the end of epoch 4.

3) Nature of validation: During training, the model is evaluated on the validation set at the end of each epoch. It is possible that the batch of data randomly selected for validation at epoch 4 contained outliers, noisy samples, or a rare pattern that the model had not yet learned to generalise to, causing a temporary high error. This event does not indicate a significant problem with the overall model performance or its training stability.

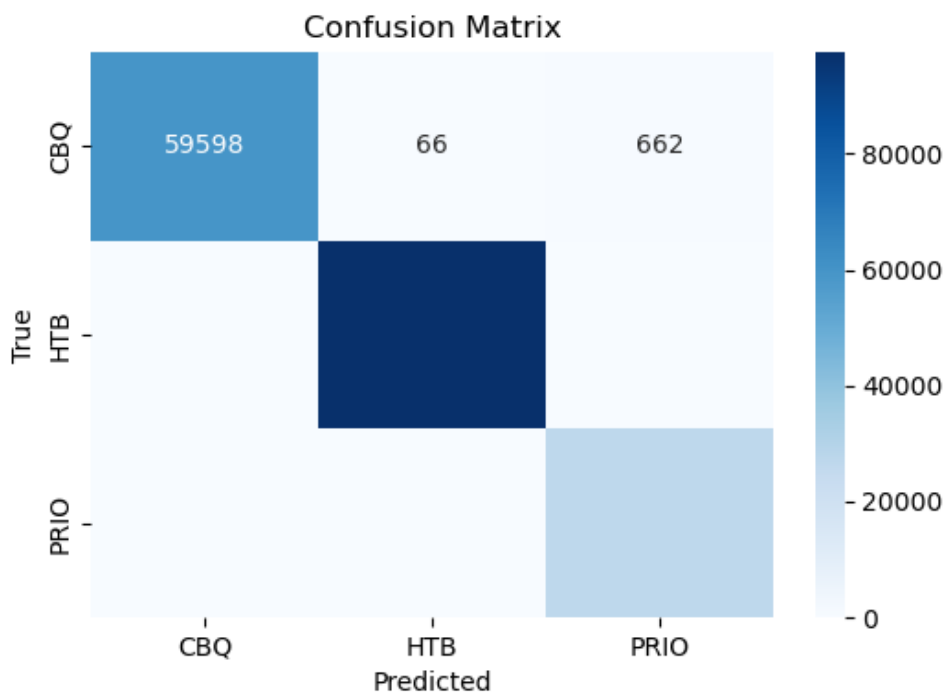


Figure 5.33: Confusion matrix for the LSTM-based scheduling prediction model on the combined PRIO, HTB, and CBQ datasets, showing that > 99% of samples fall on the diagonal with only minor confusion for CBQ, confirming near-perfect discrimination between scheduling algorithms.

Figure 5.33 illustrates the confusion matrix for the LSTM model's predictions of scheduling algorithms. The confusion matrix demonstrates excellent model performance, with most predictions correctly classified across all three classes (CBQ, HTB, and PRIO). The matrix displays minimal off-diagonal values, indicating that misclassifications are rare. The CBQ class is occasionally misclassified as HTB or PRIO, explaining the slight dip in precision for the PRIO class. Nevertheless, the dominant diagonal indicates high accuracy, reinforcing the strong results for precision, recall, and F1-scores.

The confusion matrix shows that the model accurately classifies most instances in each category, with the diagonal entries (representing correct classifications) significantly outnumbering the off-diagonal entries. In particular, CBQ has only a small number of misclassifications, while HTB and PRIO are almost perfectly identified.

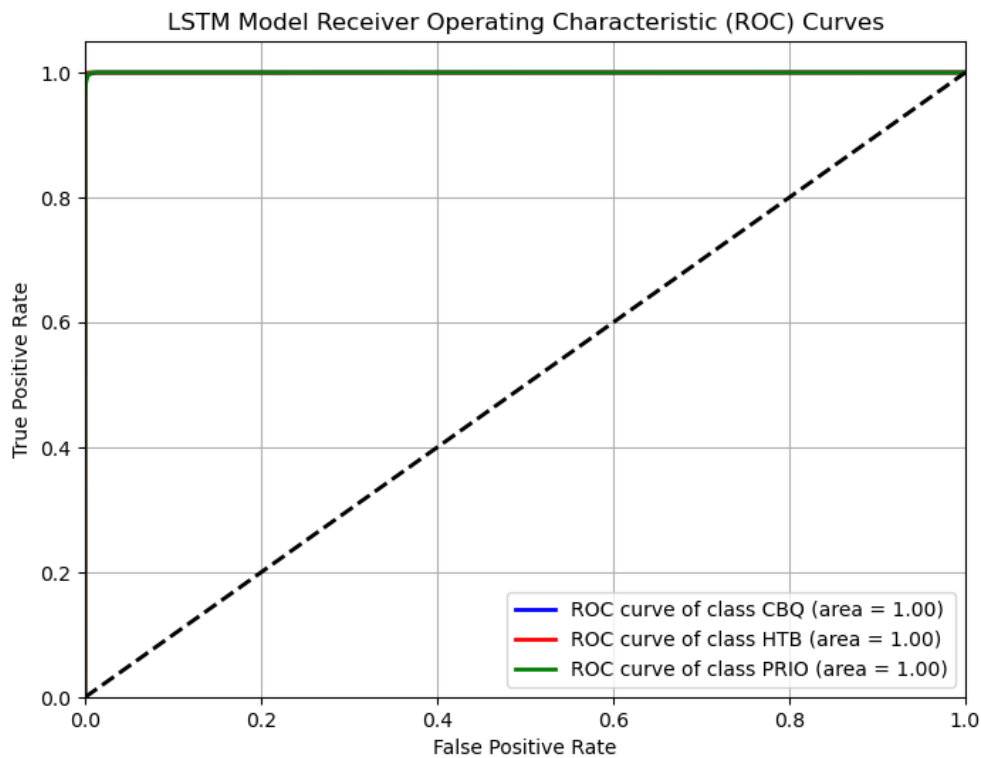


Figure 5.34: ROC curves for the LSTM-based scheduling prediction model. All three classes (CBQ, HTB, and PRIO) achieve AUC=1.0, with curves clustered near the top-left corner, confirming almost perfect discrimination between scheduling algorithms and very low false-positive rates, in line with the confusion matrix and classification report.

The LSTM's ROC curve, shown in Figure 5.34, indicates that the model performs exceptionally well, as evidenced by the high AUC results for each class. Additionally, the AUC values of 1.0 for each class suggest that the highly discriminative model effectively distinguishes among the scheduling algorithms (CBQ, HTB, and PRIO) based on their

traffic patterns. The curves for each class should lie far from the diagonal (random classifier) line, confirming strong classification performance with minimal false positives and false negatives.

Furthermore, the ROC curves for all three classes (CBQ, HTB, and PRIO) lie near the top-left corner, with areas under the curve (AUC) of 1.0, indicating perfect or nearly perfect discrimination between each class and the others. In practice, the model consistently achieves high TPRs while maintaining very low false positive rates, as shown in the training and validation plots and the confusion matrix.

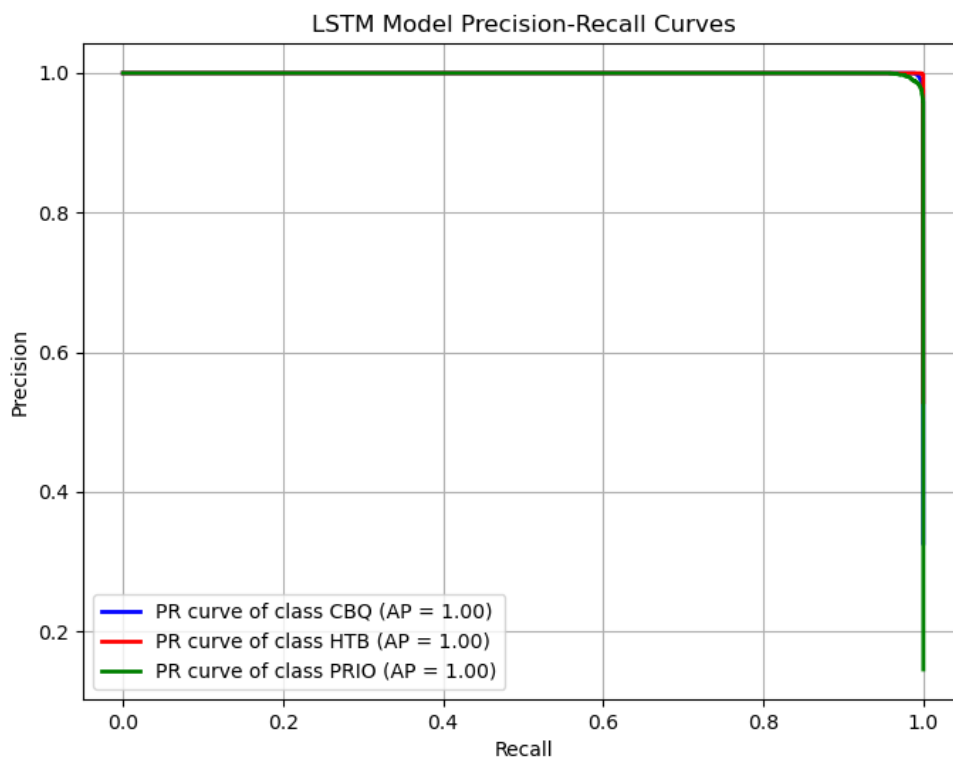


Figure 5.35: Precision-recall curves for the LSTM-based scheduling prediction model.

All three classes (CBQ, HTB, and PRIO) remain near the upper-right corner, with average precision ≈ 1.0 , confirming that the model maintains high precision and recall across decision thresholds, in line with its near-perfect ROC and confusion-matrix results.

Figure 5.35 shows the precision-recall curves for the LSTM model. The plot shows that the precision-recall curves for all three classes (CBQ, HTB, and PRIO) remain near the upper-right corner, with each class having an average precision of 1.0. These results reveal that the model consistently maintains high precision and recall across a range of decision thresholds, consistent with the near-perfect performance observed in the other plots.

	precision	recall	f1-score	support
CBQ	1.00	0.99	0.99	60326
HTB	1.00	1.00	1.00	97735
PRIQ	0.97	1.00	0.98	26862
accuracy			0.99	184923
macro avg	0.99	0.99	0.99	184923
weighted avg	0.99	0.99	0.99	184923

Accuracy: 0.9948
 Precision (weighted): 0.9949
 Recall (weighted): 0.9948
 F1 Score (weighted): 0.9948
 Model saved to 'lstm_model_with_class_weights.pth'

Figure 5.36: Classification report for the LSTM-based scheduling prediction model, showing near-perfect precision, recall, and F1-scores for CBQ and HTB and slightly lower precision for PRIQ (0.97). The overall accuracy of 0.9948 and the macro- and weighted-average scores of ≈ 0.99 confirm that the model reliably distinguishes among the three scheduling algorithms.

Figure 5.36 illustrates the LSTM's results, demonstrating near-perfect performance across all three scheduling classes (CBQ, HTB, and PRIQ), with an overall accuracy of 0.9948. Notably, CBQ and HTB exhibit perfect or near-perfect precision, recall, and F1-scores, while PRIQ shows a marginally lower precision of 0.97. The macro and weighted averages also remain exceptionally high, at or near 0.99, confirming that the model consistently distinguishes between the classes. These results indicate that the LSTM-based classifier robustly identifies each scheduling algorithm under the tested conditions.

5.4.3.3 Interpretation of Results for LSTM-Based Scheduling Prediction Model

As shown in the plots (Figures 5.32 to 5.36), the model demonstrates high performance across multiple evaluation metrics and visualisations, underscoring its robust ability to differentiate among the three scheduling algorithms (CBQ, HTB, and PRIQ). The training and validation loss curves converge to low values, showing effective learning. The confusion matrix further corroborates this robustness, with each class showing pronounced diagonal dominance and negligible misclassifications.

The classification report in Table 5.20 reinforces the observations, showing near-perfect precision, recall, and F1 Scores, and an overall accuracy of 99.48%. The ROC and precision-recall curves of each class lie near their ideal regions, consistently achieving area-under-the-curve values of 1.0. Furthermore, these findings highlight the model's strong discriminative ability and generalisability to unseen data. Such high performance suggests that the LSTM-based approach can effectively capture temporal dependencies in network traffic data, making it a valuable component within an adaptive scheduling system in next-generation network environments.

Table 5.20: Classification Report for LSTM model.

Class	Precision	Recall	F1-score	Support
CBQ	1.00	0.99	0.99	60,326
HTB	1.00	1.00	1.00	97,735
PRIO	0.97	1.00	0.98	26,862
Accuracy	-	-	0.9948	184,923
Macro Avg	0.99	0.99	0.99	184,923
Weighted Avg	0.99	0.99	0.99	184,923

Key observations:

- Loss convergence: The training and validation losses converge to approximately 0.02, indicating stable learning and minimal overfitting.
- Confusion matrix: Nearly all instances fall on the main diagonal, demonstrating that the model rarely confuses one class for another.
- Classification metrics: The model demonstrates strong discriminative capability, achieving an overall accuracy of 0.9948 and near-perfect precision, recall, and F1 Scores.
- ROC and Precision-Recall Curves: The model achieves a perfect or near-perfect area under the ROC curve ($AUC = 1.0$) and average precision ($AP = 1.0$) for all classes, confirming exceptional performance across multiple decision thresholds.

The near-perfect classification results imply that even in raw packet capture, without explicit QoS metrics such as latency or throughput, the scheduling algorithms still produce distinctive patterns that the LSTM model can learn to recognise. In other words, features such as packet timing, size distributions, and inter-arrival intervals captured with Wireshark are sufficiently distinctive to distinguish CBQ, HTB, and PRIO. This indicates that each algorithm leaves a unique “signature” in the packet-level traces, allowing the model to classify them accurately despite the absence of explicit QoS metrics. This is an important finding in this experiment, and, from a network management perspective, it indicates that the LSTM-based approach can exploit these characteristic signatures to reliably identify, or even adaptively select, the optimal scheduling policy in real time.

1) Implications for QoS Management in 5G and Beyond: The experimental results unequivocally demonstrate that DL models, specifically LSTMs, can effectively classify scheduling algorithms (CBQ, HTB, and PRIO) based on raw packet-level data with high accuracy (0.9948), even without explicit QoS metrics. This capability has profound implications for QoS management in 5G networks. Firstly, it enables real-time, adaptive scheduling decisions based on observed network traffic patterns. This “traffic pattern recognition” allows the network to infer the likely QoS requirements of different data flows without explicit signalling, a particularly valuable feature in dynamic environments. Secondly, with this classification capability, networks can dynamically optimise performance by balancing crucial QoS parameters such as latency, throughput, and reliability, leading to more efficient resource allocation. For instance, the model could

identify traffic patterns associated with latency-sensitive traffic (e.g., Virtual Reality (VR), Augmented Reality (AR), and Tactile Internet) and prioritise them using appropriate scheduling algorithms to minimise delays. Furthermore, this approach can proactively detect and mitigate network congestion by predicting potential bottlenecks based on traffic patterns and adjusting scheduling accordingly. The high precision and recall scores, especially for HTB, also suggest that the high accuracy enables automatic switching between scheduling algorithms in response to network conditions.

In summary, the ability to classify scheduling algorithms from raw packet data opens the door to automated network optimisation and self-healing networks. In future deployments, such models could be integrated into network management systems to continuously learn and adapt to evolving traffic patterns, paving the way for more efficient, responsive, and robust QoS management in future communication network environments and reducing the need for manual configuration.

2) Alignment with Existing Frameworks

a) 3GPP QoS Framework: The 3rd Generation Partnership Project (3GPP) defines the QoS architecture for 5G, which includes concepts such as QoS Flows, 5QI (5G QoS Identifier), and traffic steering. Our approach can enhance this framework by providing a more dynamic, adaptive method for managing QoS flows. Instead of relying solely on pre-defined 5QIs, the LSTM model could learn to predict the QoS requirements of different applications based on traffic patterns and dynamically adjust the 5QI or traffic steering accordingly. Our approach aligns with the 3GPP's goal of supporting a wide range of services with diverse QoS requirements.

b) Network Slicing: Network slicing is a vital feature of 5G QoS, allowing operators to create virtual networks tailored for specific services. Our model could be used to optimise resource allocation within each network slice. By analysing traffic patterns within a slice, the model could dynamically adjust resource allocation (e.g., bandwidth, processing power) to meet the specific QoS requirements of the application running in that slice.

c) SDN/NFV Architecture: SDN and Network Functions Virtualisation (NFV) provide the infrastructure for flexible and programmable networks. Our LSTM model can be integrated into the SDN controller for intelligent QoS management. The SDN controller can use the model's predictions to make real-time traffic routing, scheduling and resource allocation decisions.

d) ETSI's ZSM (Zero-touch Network and Service Management): ETSI's ZSM framework aims to automate network management tasks. Our approach can contribute to this goal by automating QoS optimisation. Additionally, the model can continuously monitor network performance, identify potential issues, and automatically adjust QoS parameters to maintain the desired service levels.

3) Alignment with Existing Literature: A growing body of literature explores the application of DL techniques to improve QoS management in wireless networks, particularly in the context of 5G networks and emerging application domains. Various studies have demonstrated the effectiveness of deep neural architectures such as CNNs, LSTMs, and hybrid models for network traffic classification, delay prediction, and scheduling optimisation. For example, Mahajan et al. [63] proposed a hybrid DL framework to predict network traffic in wireless mesh networks, while Dangi and Lalwani [64] employed a DL approach for traffic control and forecasting in 5G networks.

Furthermore, Sun et al. [31] introduced a framework for meta-transfer metric learning for time-series classification in 6G-supported intelligent transportation systems, illustrating the value of deep models for pattern extraction. This study aligns with these efforts by implementing and evaluating LSTM-based models for classifying algorithms using traffic pattern data collected from live experimental networks (*see Sections 4.7 and 5.4.3*). The results reaffirm the capacity of recurrent architectures, such as LSTM, to model temporal dependencies in network behaviour and support adaptive scheduling decisions for QoS-constrained environments.

5.4.4 Comparison of CNN-Based, CNN-LSTM-Based and LSTM-Based Scheduling Predictions Models

To facilitate a clear comparison of the evaluated DL architectures, Table 5.21 summarises the key performance characteristics and findings for the CNN-based and hybrid CNN-LSTM models utilising image-based network data representations, alongside the LSTM model applied directly to the raw network traffic features. The Table contrasts the models based on input data, optimal validation accuracy, training stability, generalisation capability, sensitivity to data size, and overall assessment, providing a consolidated view of their relative effectiveness for the QoS classification task.

Table 5.21: DL Models (CNN, CNN-LSTM, and LSTM) Comparison for Network QoS Classification.

Feature	CNN (Image-based)	CNN-LSTM (Image-based)	LSTM (Raw Data)
Input Data	Image Representation (network traffic)	Image Representation (network traffic)	Raw Network Traffic Features
Best Performance (Validation Acc.)	87.50%	50%	99.5%
Training Stability	Variable to Unstable	Very Unstable/Poor Convergence	Stable and Good Convergence
Generalisation/Overfitting	High Overfitting Risk (Small Dataset)	Poor Generalisation (Learning Failed)	Excellent Generalisation (Low Overfitting Risk)

Dataset Size Sensitivity	Very High (Performance limited by small size)	Likely high (Poor results)	Robust (Performed well on a large dataset)
Key Metrics (Beyond Acc.)	K-Fold showed high variance.	Erratic curves	Excellent (AUC=1.0, AP=1.0, High Precision/Recall/F1)
Overall Assessment	Potentially requires significant improvement (Work in progress)	Ineffective in tested configurations	Highly Effective and Robust

Table 5.21 presents a comparison of three DL architectures for network QoS classification: a CNN utilising image-based representations of network data, a hybrid CNN-LSTM model that also employs image inputs of network traffic, and an LSTM model applied directly to raw network traffic features. The LSTM model operating on raw data demonstrated substantially superior performance, achieving approximately 99.5% validation accuracy with stable convergence and excellent generalisation, evidenced by near-perfect AUC and Average Precision scores of 1.00 across all classes (CBQ, HTB, and PRIO).

In contrast, the models employing image-based representations faced significant challenges. The pure CNN approach, while achieving a promising 87.50% validation accuracy in one optimised run, exhibited considerable instability and clear signs of overfitting in other experiments, limitations attributed mainly to the very small image dataset size; refinement of this model is ongoing. The hybrid CNN-LSTM architecture applied to the image data performed poorly, with unstable training dynamics, low validation accuracies (up to 50%), and failure to converge effectively. These results indicate that, within the scope of these experiments, the LSTM model, which processes raw network features, provides a highly effective and robust solution for this task. In contrast, image-based approaches, particularly CNNs, require further development, primarily to address data scarcity and improve model generalisation.

5.4.5 Statistical Significance of Validation Metrics (Experiment 3)

1) CNN (Image-based, Figures 5.25-5.29): Validation accuracy increases from 25% to 62.5% to 87.5% across Figures 5.25-5.28, indicating increasing separability; however, the dataset uses six images per class (augmented), so point estimates exhibit wide sampling variability. The 100% held-out accuracy in Figure 5.28 with a perfect confusion matrix (Figure 5.29) is numerically decisive; however, it might be statistically fragile given the small validation set. This result should be interpreted as suggestive pending broader validation—the comparison in Table 5.21 flags high variance across k-folds, reinforcing the limited statistical stability of image-based scores.

2) CNN-LSTM (Image-based, Figures 5.30-5.31): The model training process plateaus at 25% (Figure 5.30) and ~50% (Figure 5.31) with piecewise LR decaying to 0; curves show underfitting and no signs of overfitting. Statistically, these values indicate non-significant discrimination for this setup – the effect sizes are too small and unstable to support claims of reliable scheduler classification from image encodings.

3) LSTM (PRIO, HTB, and CBQ Datasets (raw traffic)) – effect sizes, Figures 5.32-5.36: Across the three raw-traffic datasets, the LSTM model shows near-ceiling performance: overall accuracy of 0.9948 with considerable per-class support (CBQ = 60,326; HTB = 97,735; PRIO = 26,862). The confusion matrix is almost perfectly diagonal with only rare off-diagonal errors (Figure 5.33). Discriminative performance is essentially perfect, with ROC-AUC = 1.0 and average precision (AP) = 1.0 for each class (ROC/PR curves; Figures 5.34-5.35), consistent with the classification report where macro/weighted scores are ≈ 0.99 . Given the sample size, these results indicate negligible class overlap on raw traces and very low sampling uncertainty, supporting robust and reproducible scheduler identification.

4) Representation effect (image vs raw, Table 5.21): The comparison Table 5.21 summarises the pattern: CNN can reach 87.5%; however, it is variance-prone on tiny image data; CNN-LSTM is unstable/low ($\leq 50\%$); LSTM (raw) is highly effective and robust ($\approx 99.5\%$, AUC/AP = 1.0). Statistically, this shows that representation choice dominates: abundant, fine-grained packet-level features (raw) yield conclusive metrics, while sparse image encodings inflate uncertainty and variance.

5) Trade-offs and conclusions (anchored on the above metrics): For scheduler classification, the LSTM on raw traffic provides conclusive evidence (accuracy ≈ 0.995 , AUC/AP=1.0) and should be the operational choice. The CNN (image-based) shows indicative separability (up to 87.5%; even 100% on a tiny split); however, given the small validation pool and k-fold variance, its current evidence is not yet statistically stable. The CNN-LSTM (image-based) results ($\leq 50\%$) are not statistically persuasive for this task under current data/parameters. In short, for reliable, statistically significant scheduler identification, the raw-feature LSTM model is ready for operational QoS monitoring/verification (and potentially closed-loop adaptation).

The near-ceiling LSTM results are significant for QoS in 5G because they show that scheduler behaviour leaves a highly separable signature in raw traffic, enabling reliable, data-driven identification of the active queuing/scheduling policy (e.g., PRIO, HTB, and CBQ) at run time. In practice, this supports closed-loop QoS control: the network can verify that the intended QoS configuration is actually being enforced, rapidly detect misconfiguration or policy drift, and trigger corrective actions before latency/jitter targets are violated. Given the large sample sizes and minimal overlap between classes, the evidence suggests the approach is robust and reproducible, which is crucial for operational 5G environments where URLLC and high-throughput services share infrastructure and QoS assurance depends on dependable monitoring and enforcement.

5.5 Experiment 4: Comparative Evaluation of Network-Delay Prediction Models

Predicting network QoS metrics in real time is a critical task for proactive network management. The underlying research problem for this experiment is time-series prediction of network delay using historical QoS metrics from a live wireless LAN deployment. Precisely, given sequences of network metrics (e.g., packet loss, jitter, and throughput), the aim is to accurately forecast the delay at the next time step in milliseconds. This is challenging due to the temporal dynamics and potential device-specific patterns in network behaviour. The scientific question guiding this experimental study is: Can advanced DL models that explicitly capture temporal dependencies (such as LSTM and Transformer-based architectures) achieve superior accuracy in predicting network delay compared to a traditional ML regression model that uses no explicit sequential modelling? In other words, this experiment investigates whether incorporating sequence learning yields tangible improvements in prediction performance relative to a strong baseline (Random Forest) for network delay forecasting.

5.5.1 Summary of Experimental Setup

To address the research question of this experiment, we conducted a comparative study involving three time-series prediction experiments using the PF dataset, which comprises wireless QoS metrics. The dataset contains timestamped measurements of per-device network metrics (features such as jitter, inter-packet arrival time, packets sent/received, packet loss and loss rate, and instantaneous and average throughput). Along with the target variable (`delay_ms`). The dataset was pre-processed to remove entries with missing delay values and sorted by time for each device. Crucially, we ensured a leakage-safe split by device: the training and validation sets were partitioned so that no device's data appeared in both, guaranteeing that the evaluation assesses generalisation to unseen devices. A stratified *GroupShuffleSplit* (80% train, 20% validation by device) was used for this purpose. Using the same set of input features (nine QoS metrics as numeric features, plus a categorical device identifier), all models were trained to predict the next-step delay.

Three modelling approaches were implemented and tuned:

1) Hybrid Linear-LSTM-Transformer Model: A custom DL regressor model combining a linear projection layer, an LSTM sequence encoder, and Transformer encoder layers. This hybrid model processes a sequence of 10 time steps of input features (sequence length = 10), projecting each time step's features into a 64-dimensional embedding and feeding the sequence through an LSTM (hidden size = 64), followed by two Transformer encoder layers (2 attention heads) before a final output layer. The design aims to capture both short-term temporal patterns (via LSTM) and longer-range dependencies (via self-attention) within the 10-step window. Hyperparameters (embedding dimension, hidden size, number of Transformer layers, learning rate,

dropout, etc.) were optimised via Optuna Bayesian search. Training used early stopping (patience 8) and the ReduceLROnPlateau scheduler to achieve robust convergence.

2) Pure LSTM Model: A two-layer LSTM regressor that also ingests sequences of 10 past time steps of features. The LSTM model had a hidden state size of 64 and included dropout regularisation (~20% optimal dropout) to mitigate overfitting. Similar to the hybrid model, it was tuned with Optuna for hyperparameters such as the learning rate, number of layers (set to 2), batch size, and weight decay. This model represents a conventional deep sequence modelling approach without attention mechanisms. It likewise used early stopping and the same training pipeline as the hybrid model, ensuring a fair comparison.

3) Random Forest Baseline: A Random Forest regressor model was used as a non-sequential baseline. This model operates on “flat” tabular data, meaning it does not use a window of past observations; instead, it uses the instantaneous features at a given time step (e.g., current throughput, jitter, loss rate, etc.) to predict the current delay. To enable a fair comparison, the RF was given the same set of input feature variables as the deep models (including the device identifier as an encoded categorical feature), but without any explicit sequential context (i.e., no time-lag features were added). The RF model was also tuned with Optuna, optimising parameters such as the number of trees (estimators), the maximum depth (optimal depth ~15), and the feature subsampling fraction to achieve the best validation performance.

All models were evaluated on a held-out validation set of unseen devices using standard regression metrics: RMSE, MAE, and the coefficient of determination (R-Squared). These metrics were calculated by comparing model predictions with true delay values on the validation data. Lower RMSE/MAE and higher R^2 indicate better accuracy. For completeness, we also tracked training-set performance to assess overfitting. The following subsections present the performance results for each model.

5.5.2 Hybrid Model (LSTM and Transformer) Performance Analysis

1) Training and Validation Loss Curves: The hybrid (LSTM and Transformer) model’s learning curve (Figure 5.37) shows a steep initial drop in training loss, accompanied by a corresponding decline in validation loss over the first ~5 epochs, indicating rapid initial learning. After epoch 5, the validation loss reaches a low plateau and begins to fluctuate, with minor spikes (e.g., a noticeable uptick around epoch 9), even as the training loss continues to decrease. This divergence suggests the onset of overfitting – the model is fitting the training data more closely than the validation data. The lowest validation loss (dashed red line at an MSE of ~7.938) occurs around epoch 16, after which no further improvement is observed on the validation set. The training curve remains below the validation curve for most epochs, and the gap widens slightly in later epochs, suggesting the hybrid model may be somewhat overfitting, despite being stopped early at epoch 24. In summary, convergence behaviour is achieved (both losses stabilise); however, the

validation loss trajectory suggests that the model's capacity may be capturing noise or device-specific patterns that do not generalise beyond the training set.

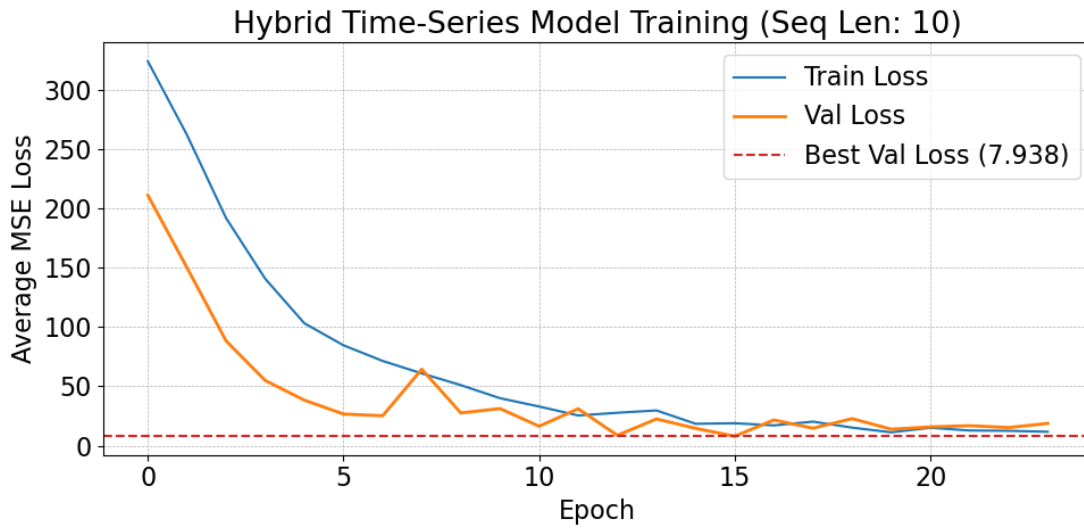


Figure 5.37: Training and validation MSE loss for the hybrid Linear-LSTM-Transformer QoS predictor. Loss drops rapidly in the first ≈ 20 epochs and then plateaus, with a minimum validation loss of ≈ 7.9 and a small persistent gap to the training curve, indicating good overall fit, and mild overfitting on unseen devices.

2) Regression Metrics: The bar chart in Figure 5.38 illustrates the RMSE, MAE, and R^2 for the hybrid model (green = training, red = validation). The hybrid model achieves a train RMSE of 1.762 ms and a train MAE of 0.806 ms, compared with a validation RMSE of 4.326 ms and a validation MAE of 1.069 ms, indicating that the model is more accurate on the training data than on unseen devices. This discrepancy, along with a high training R^2 of 0.989 versus a lower validation R^2 of 0.911, quantitatively confirms moderate overfitting. The validation R^2 of 0.911 indicates that the hybrid model still explains $\approx 91\%$ of the variance in latency for new devices, which is a strong result. However, the inflated training R^2 (≈ 0.99) and much lower validation R^2 suggest that some learned structure did not generalise. The error metrics on the validation set (RMSE ≈ 4.3 ms) are relatively low in absolute terms; however, in latency-critical settings, even a few milliseconds of prediction error can be significant. Nevertheless, the hybrid model's prediction accuracy is decent: an MAE of ≈ 1.07 ms implies that, on average, the delay prediction is within about 1 ms of the true value for the validation device. The higher RMSE (4.3 ms) indicates occasional larger errors, reflecting outlier instances in which the prediction deviated by a larger margin.

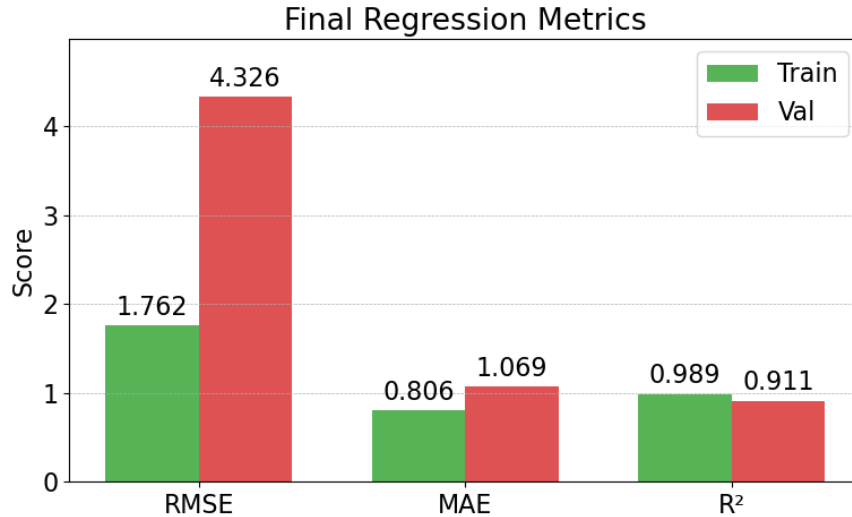


Figure 5.38: Final regression RMSE, MAE, and R^2 for the hybrid Linear-LSTM-Transformer model. The much better training than validation scores (RMSE 1.76 vs 4.33 ms; R^2 0.989 vs 0.911) confirm moderate overfitting, yet validation MAE of ≈ 1.07 ms and $R^2 \approx 0.91$ reveal that the model still predict latency for unseen devices with reasonably high accuracy, apart from occasional larger errors.

3) True vs Predicted Delay: Figure 5.39 illustrates the true vs predicted delay for the hybrid (LSTM and Transformer) model. Each point represents a predicted delay (y-axis) versus the true observed delay (x-axis) for a validation data point, with the dashed line representing the ideal relationship of $y = x$. The scatter plot shows a generally linear relationship between predicted and true delays, with many points clustering near the diagonal. For lower delay values (e.g., ≈ 50 ms), the purple points lie very close to the ideal line, indicating the hybrid model predicts short delays with high accuracy. At higher delays, we observe greater dispersion: some points at true delays of 100-150 ms are slightly under-predicted (falling below the line), while a few at ≈ 150 -180 ms are over-predicted (falling above the line). The wildest deviations are still modest (a few tens of milliseconds at most), but they indicate that the model occasionally struggles with certain high-delay outliers. The overall alignment with the diagonal is consistent with $R^2 (\approx 0.91)$, confirming that the prediction quality is good but not perfect. In practice, the hybrid model can reliably rank and estimate network delay most of the time; however, its worst-case delay predictions can err by a significant margin, potentially violating strict QoS targets if used alone.

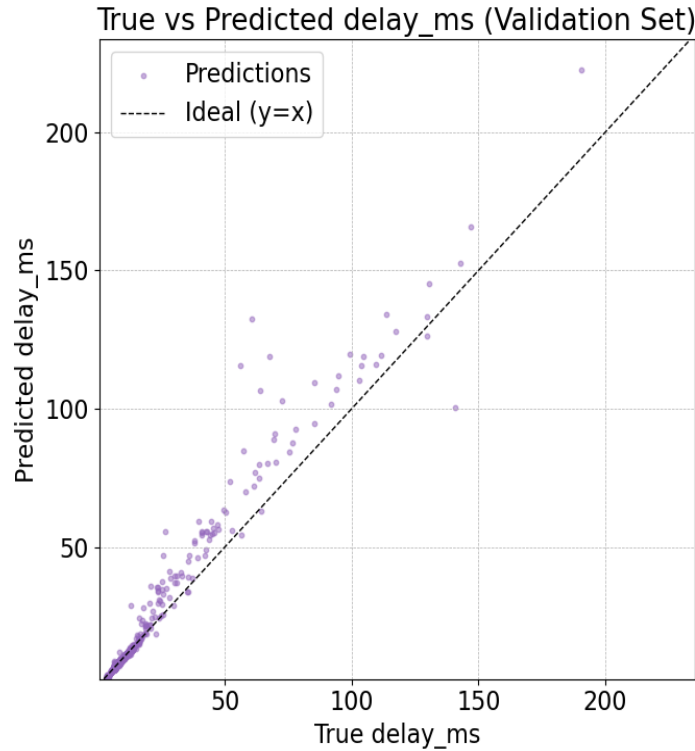


Figure 5.39: True vs predicted delay for the hybrid Linear-LSTM-Transformer model on the validation device. Most points lie close to the diagonal for 0-100 ms delays, confirming good overall accuracy ($R^2 \approx 0.91$), while the wider spread for > 100 ms delays shows occasional misestimation of worst-case latency, which may limit strict tail-latency guarantees.

5.5.3 Pure-LSTM Model Performance Analysis

1) Training and Validation Loss Curves: Figure 5.40 shows the training and validation loss curves for the LSTM-only model. The LSTM model's convergence profile is similar to that of the hybrid model in its initial phase, with both training and validation losses decreasing sharply in the first few epochs. By epoch 5, the validation loss has dropped below 10 (MSE), representing a significant improvement. Notably, the gap between the training and validation losses remains small at this stage. Beyond epoch 5, the validation loss exhibits some fluctuations (e.g., a slight increase around epochs 6-8, followed by new minima at epochs 10 and 18); however, these oscillations are relatively minor. The LSTM's best validation loss (dashed red line at ≈ 5.03) is substantially lower than the hybrid's best, and it is achieved earlier (around epoch 10-15). After this point, the model shows no further improvement, and early stopping occurs because the validation loss stops decreasing. Vitally, the train and validation loss curves converge to nearly the same level by the end of training, with minimal divergence.

Furthermore, this suggests that the LSTM model did not overfit the data – the validation loss at convergence is almost as low as the training loss, implying good generalisation.

The smoother, lower validation loss curve for the LSTM compared to the hybrid model suggests that the LSTM may have been a better capacity match for the problem, extracting predictive temporal patterns without learning as much noise.

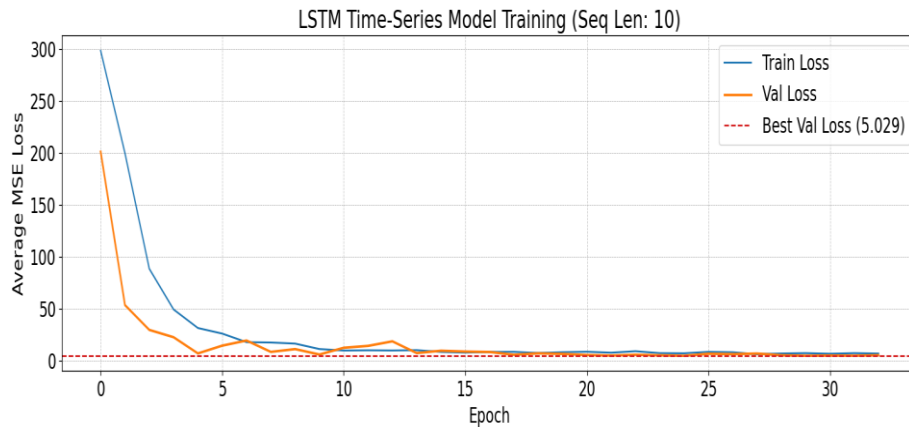


Figure 5.40: Training and validation loss for the LSTM-only model. Loss drops rapidly in the first few epochs and reaches a best validation MSE of ≈ 5.0 around epochs 10-15, with train and validation curves closely aligned, indicating strong convergence, minimal overfitting, and better fit than the hybrid model.

2) Regression Metrics: Figure 5.41 illustrates the regression metrics for the LSTM model, showing strong performance on both the training and validation sets. On the validation set, the RMSE is 2.358 ms, and the MAE is 0.593 ms, which is almost half the error of the hybrid model's metrics, while the R^2 is 0.974 (indicating that 97.5% of the variance in latency is explained). Compared to training (RMSE 1.280 ms, MAE 0.431 ms, and R^2 0.994), the validation metrics show only a slight degradation, confirming minimal overfitting. This strongly implies that the LSTM captured the underlying relationships in the data with high fidelity. The low MAE (≈ 0.59 ms) means that, on average, the LSTM's delay predictions are accurate to well below 1 ms error, which is extremely precise for network latency forecasting. The small validation RMSE further reveals that even the occasional larger errors are not very large (approximately 2-3 ms). Such consistency between training and validation performance demonstrates the LSTM model's robust generalisation – it maintained high accuracy on completely held-out data from a different device. This can be attributed to the model's sequence-learning capability, which effectively captures common temporal dynamics across devices (e.g., patterns in throughput or packet metrics that influence delay) without over-specialising to the training devices.

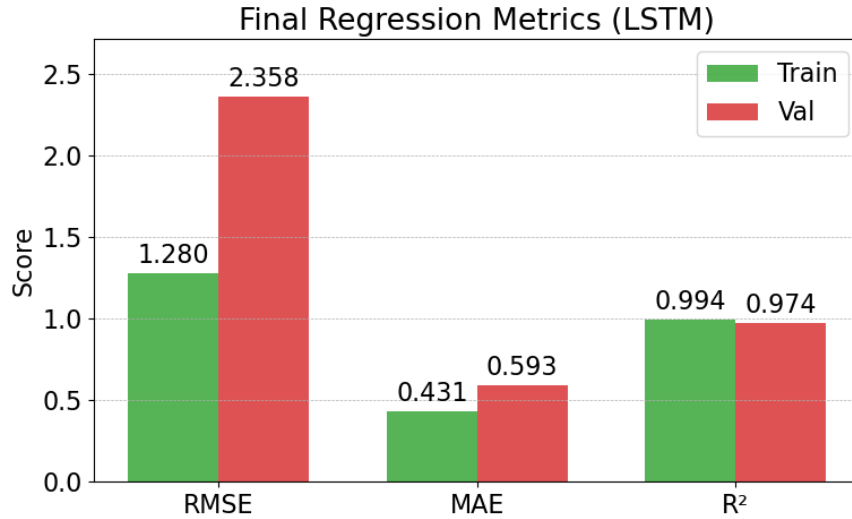


Figure 5.41: Final regression RMSE, MAE and R^2 for the LSTM-only model. The small gap between training (RMSE 1.28 ms, R^2 0.994) and validation (RMSE 2.36 ms, R^2 0.974) metrics indicates minimal overfitting, while the low validation MAE (≈ 0.59 ms) shows that the LSTM predicts latency for unseen devices with sub-millisecond average error outperforming the hybrid model.

3) True vs Predicted Delay: Figure 5.42, the True vs Predicted delay for the LSTM model on the validation set. The LSTM's prediction scatter plot is tightly clustered around the diagonal line, even more so than the hybrid models. For nearly all points, specifically in the 0-100 ms true delay range, the predictions lie extremely close to the ideal $y = x$ line, indicating the model predicts those delays with very high accuracy. A few points at the high end (e.g., true delay of approximately 150-180 ms) exhibit minor deviations – for instance, one point around the true 150 ms is predicted to be higher (≈ 200 ms). Nevertheless, these deviations are relatively small, and crucially, no systematic bias is evident (the points are symmetrically scattered around the line). The strong alignment reflects the high R^2 (≈ 0.974): The LSTM's predictions capture nearly all the variability in the latency values. Visually, the LSTM's scatter has less spread than the hybrid's, confirming better predictive precision. This indicates that the LSTM's temporal recurrent structure is highly effective for this QoS prediction task, likely by capturing short-term dependencies in network behaviour (e.g., how recent jitter or instantaneous throughput affects delay). Overall, the LSTM model yields excellent prediction quality, suggesting that sequence-based modelling of network metrics is a promising approach for accurate latency estimation across different wireless devices.

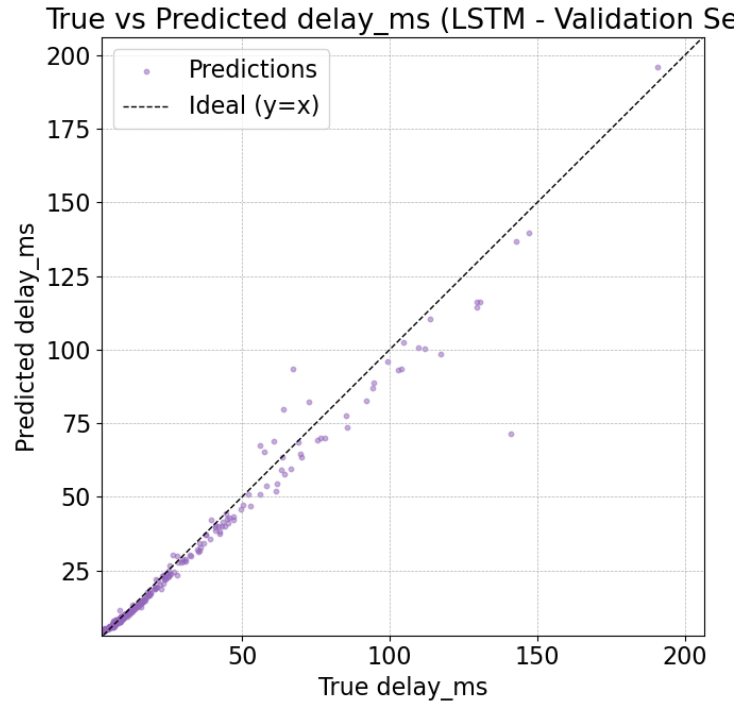


Figure 5.42: True vs predicted delay for the LSTM-only model on the validation device. Most points lie very close to the diagonal for 0-100 ms delays, with only small deviations at the highest delays, confirming the high R^2 (≈ 0.97) and showing that the LSTM predicts latency more accurately and with less spread than the hybrid model.

5.5.4 Random Forest Baseline Performance Analysis

1) Regression Metrics: Figure 5.43 shows that the RF model using flat (time-independent) features achieves exceptionally high accuracy. The validation RMSE is 0.617 ms, the MAE is 0.154 ms, and the R^2 is 0.9982, indicating virtually perfect predictions on the validation set. These metrics are only marginally worse than its training performance (Train RMSE 0.389 ms, MAE 0.061 ms, R^2 0.999), revealing an almost negligible generalisation gap. The RF's ability to attain 99.8% R^2 on unseen data suggests that the static input features contain almost all the necessary information to predict *delay_ms*, at least for the device in the validation set. The RF outperforms both neural network models in absolute error, delivering an average error of sub-milliseconds. However, such an incredibly low validation error warrants careful interpretation: it could indicate that the RF model indeed captured a genuine functional relationship between the input metrics and delay (implying the system's delay is largely deterministic given those features). It may also suggest that the particular validation device's data was highly similar to the training data distribution (facilitating near-perfect generalisation). Unlike DL models, the RF does not explicitly model temporal sequences; its success implies that instantaneous and aggregate metrics (such as throughput, packet counts, and jitter) at a single time step were sufficient to predict latency with high fidelity in this dataset.

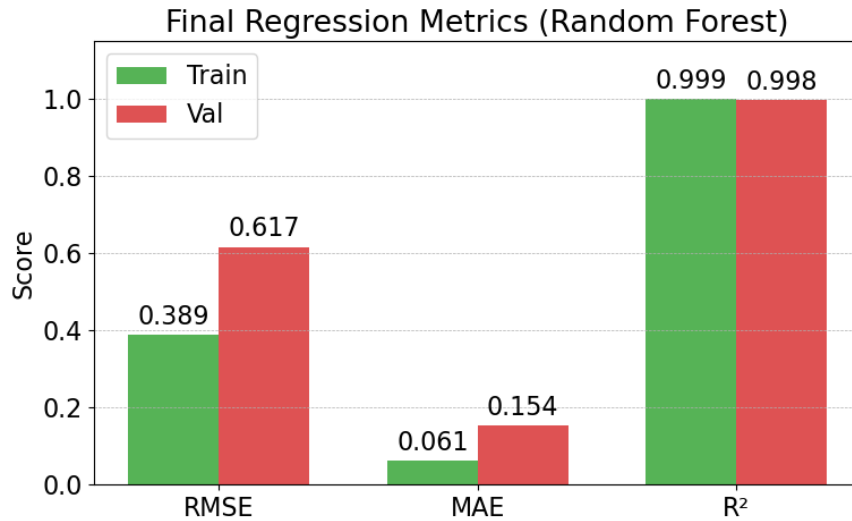


Figure 5.43: Regression performance of the RF baseline, showing sub-millisecond validation error (RMSE ≈ 0.62 ms, MAE ≈ 0.15 ms, $R^2 \approx 0.998$) and a minimal train-validation gap, indicating excellent generalisation to the held-out device and near-deterministic mapping from QoS metrics to delay.

2) True vs Predicted Delay: The RF's prediction scatter (Figure 5.44) underscores its remarkable performance. Nearly every point lies on or is extremely close to the diagonal line, from the smallest to the largest delays. There is almost no visible dispersion – for instance, if the true is 50 ms, the predicted delay is ≈ 50 ms with the minimal error for essentially all such samples. Even at the high end (true delays of ≈ 150 -180 ms), the predictions match the actual values within a few milliseconds. The tight clustering of points around $y = x$ corresponds to the RF's R^2 of ≈ 0.998 and indicates the model's predictions are consistently accurate across the entire range of observed delays. Unlike the DL models, there are practically no outliers or broad scatter, reflecting the RF's capacity to precisely model the training data relationships and apply them to the validation group. This level of accuracy suggests that, for the given scenario, the mapping from input features to delay is largely learnable via simple decision-tree splits. It is worth noting that such near-perfect validation performance, while desirable, warrants caution because it might raise concerns about overfitting or data leakage. However, since the validation set is from a separate device that is not present in the training data (ensuring no overlap in time or device ID with training data), the RF's performance strongly indicates a genuine generalisation within the feature space of network metrics.

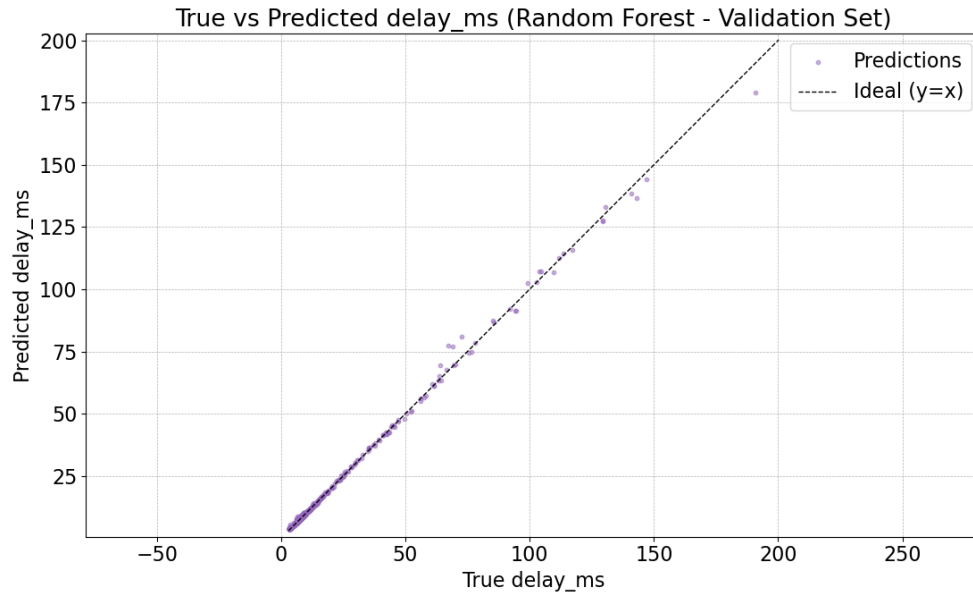


Figure 5.44: True vs predicted delay for the RF model on the validation set. Points lie almost exactly on the diagonal across the full delay range, with only tiny deviations, confirming the very high R^2 and showing that the RF predicts latency for the unseen device with near-deterministic accuracy.

3) Feature Importance: The RF model offers the advantage of interpretability through feature importance analysis. As shown in Figure 5.45, the most influential feature by far is the instantaneous throughput (bps), with an importance score of ≈ 0.8 , overshadowing all others. This suggests that the current throughput at a given moment is the dominant factor in determining network delay – intuitively, higher instantaneous throughput may lead to queuing and hence higher latency, or, conversely, indicate a well-utilised link. The second most important feature is jitter (ms), with a much smaller but non-negligible importance. Jitter measures variability in packet inter-arrival time, so a high jitter could signal network timing variability that correlates with delay spikes. The average throughput also contributes, though notably less than the instantaneous throughput; this might reflect longer-term load effects on latency. Other features (packet counts sent/received, loss rate, packets lost, whether the device was selected, and even the device ID) have virtually zero importance in the model. This indicates that the RF found that these factors contribute little to the predictive value beyond what throughput and jitter already provide.

Furthermore, from an interpretability standpoint, these results align with the domain knowledge: in wireless networks, latency is heavily affected by current traffic load and variability (throughput and jitter). The RF’s ability to identify key predictors is valuable for QoS management, as it suggests that monitoring and controlling throughput and jitter could be most effective for latency control. The high accuracy achieved by a simple model using just two features suggests a strong, relatively stable relationship between those metrics and device-level delay.

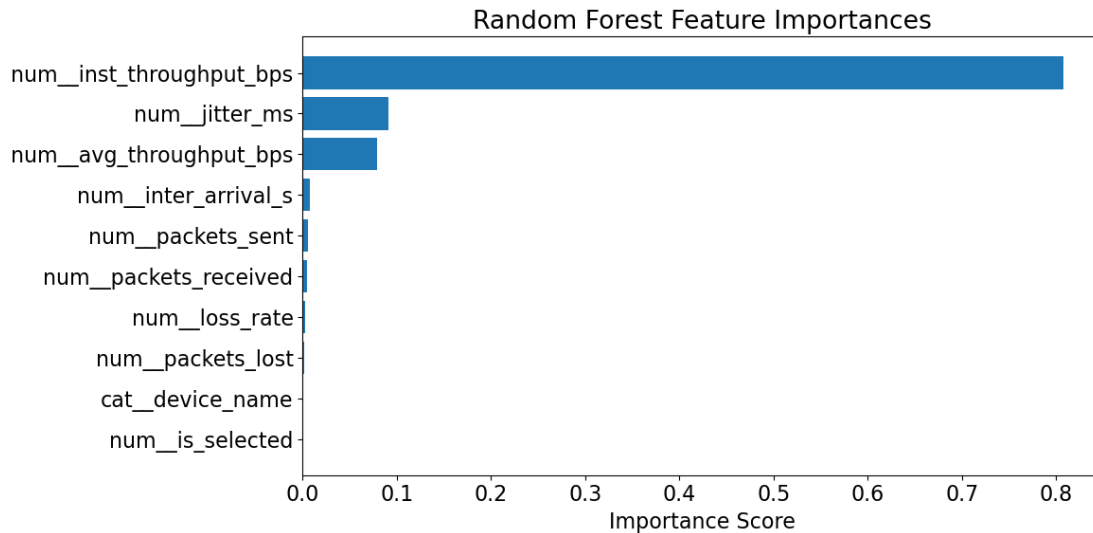


Figure 5.45: Feature importance scores from the RF model. Instantaneous throughput (bps) is by far the most influential predictor of delay, while other QoS metrics (e.g., average throughput, jitter, packet loss and device-related features) have lower importance, confirming that current transmission rate is the primary driver of latency in this dataset.

5.5.5 Quantitative Results Analysis

1) Overall Performance: Table 5.22 summarises the validation-set prediction accuracy of the three models (hybrid, LSTM-only, and Random Forest). The RF baseline achieved the best performance, with an RMSE of ≈ 0.62 ms, an MAE of ≈ 0.154 ms, and an R^2 of 0.998, indicating that it explained virtually all the variance in delay across unseen devices. The pure LSTM model was the best after the RF, with a higher error (RMSE ≈ 2.36 , MAE ≈ 0.593) and slightly lower R^2 (≈ 0.974). The hybrid Linear-LSTM-Transformer model performed well, with an RMSE of ≈ 4.33 , an MAE of ≈ 1.07 , and an R^2 of 0.911 on the validation set, an R-Squared substantially lower than the other models. In absolute terms, the hybrid model's error was about 7 times larger than the baseline's error, while the LSTM's error was about 3-4 times larger than the baseline model. All models were trained on the same data; the stark differences in performance highlight the impact of model choice.

2) Training vs Validation: We also note the degree of overfitting by comparing training-set and validation-set metrics. The RF exhibited almost no overfitting, with a training RMSE of ≈ 0.389 , a validation RMSE of ≈ 0.617 , and a training R^2 of 0.999 versus a validation R^2 of 0.998. The LSTM revealed a small generalisation gap (training RMSE ≈ 1.28 vs. validation ≈ 2.36 ; training $R^2 \approx 0.994$ vs. validation 0.974), indicating that it fit the training data well but slightly underperformed on new devices. The hybrid model had a larger gap (training RMSE ≈ 1.76 vs validation ≈ 4.33 ; training $R^2 \approx 0.989$ vs validation 0.911), suggesting it overfitted the training data more and struggled to generalise. Early stopping was triggered for the hybrid model after 24 epochs when the validation loss

ceased improving, but its best captured validation RMSE (≈ 2.82 at epoch 16, corresponding to MSE ≈ 7.94) was still higher than the LSTM's eventual performance, indicating the hybrid never achieved a lower error than the simpler LSTM during training. The metrics reported in Table 5.22 are lower the better for RMSE/MAE and higher the better for R^2 .

Table 5.22: Performance of Each Model on the Held-out Validation Set (Unseen Devices).

Model	Validation RMSE (ms)	Validation MAE (ms)	Validation R^2
Hybrid (Linear+LSTM+Transformer)	4.326	1.069	0.911
LSTM (2-layer)	2.358	0.593	0.974
RF (Baseline)	0.617	0.154	0.998

The quantitative results demonstrate that the RF baseline outperformed both DL models by a wide margin on all evaluation metrics. The LSTM outperformed the hybrid deep model, achieving roughly half the hybrid's error. In the following sub-section, we delve into these comparisons and discuss possible reasons.

5.5.6 Comparison Analysis

The comparative performance of the three models shows several noteworthy patterns. First, the baseline RF delivered the most accurate predictions by a significant margin. As illustrated in Figure 5.43, its validation RMSE (0.62 ms) is about one-quarter of the LSTM's (2.36 ms) and only one-seventh of the hybrid model's (4.33 ms). This gap is similarly reflected in MAE: The RF mean error was only 0.154 ms, versus 0.593 ms (LSTM) and 1.069 ms (hybrid). In practical terms, the DL model's predictions had an average error 4-7 times higher than the baseline. The R^2 values reinforce this ranking: the RF attained an R^2 (≈ 0.998), indicating it explained ($\approx 99.8\%$) of the variance in the latency outcomes for new devices, whereas the LSTM explained ($\approx 97.4\%$) and the hybrid ($\approx 91.1\%$). The Random Forest's near-perfect R^2 on validation suggests that it captured almost all systematic patterns in the data, leaving little to no unexplained variation.

Second, comparing the two DL approaches, the simpler pure LSTM outperformed the hybrid LSTM-Transformer model by a large margin. The LSTM achieved approximately 45% lower RMSE than the hybrid (2.36 vs. 4.33) and about 45% lower MAE (0.593 vs. 1.069). This indicates that adding Transformer encoder layers (and an additional linear projection) on top of the LSTM not only failed to improve accuracy but also significantly degraded performance on this task. The expectation was that the hybrid model might capture more complex temporal patterns or long-range dependencies within the 10-step window via self-attention, potentially improving generalisation. However, the results show the opposite – the LSTM-alone model generalised better. One likely reason is that the hybrid model, with its greater complexity (due to more parameters from the

Transformer layers), overfits the training data more severely. Indeed, the hybrid model exhibited a noticeable train-test performance gap (training R^2 of 0.989 dropping to 0.911 on the validation set) compared to the LSTM's gap (0.994 to 0.974). The Transformer's multi-head attention might have been unnecessary given the short sequence length (10 times the number of steps) and could have introduced additional model instability or training difficulty. It is also possible that the amount of training data per device or overall was insufficient for the hybrid architecture to effectively learn its additional parameters. In contrast, the LSTM, with fewer parameters, captured the necessary patterns more robustly.

Third, the Random Forest's strong performance relative to both DL models suggests that temporal sequence modelling conferred a slight advantage in this setting. Despite not using an explicit time-series history, the RF leveraged the instantaneous features of each sample (including current throughput, jitter, loss rate, etc.) to predict delay with remarkable accuracy. This implies that the current state of network metrics contains most of the information needed to predict the current delay, and that any temporal dependencies beyond what those metrics reflect do not significantly help predict the next-step latency. Providing the DL models with a history of 10 readings did not improve performance; on the contrary, the model with no history performed best. This could indicate that the network's behaviour has a mostly memoryless property for one-step-ahead delay prediction: the delay at time t may be predominantly a function of the immediate metric at time t , rather than long-term patterns. It is notable that, even though the RF was not explicitly given past delay values or time-series features, it nearly perfectly predicted delay for unseen devices, suggesting that the mapping from other QoS metrics to delay is a strong, general relationship that holds across devices.

Furthermore, the generalisation to unseen devices deserves emphasis. All models were tested on devices not seen during training (a stringent evaluation), yet the RF achieved only a minor increase in error on new devices (validation RMSE 0.617) compared to training (RMSE 0.389). This indicates excellent generalisation – the model learned patterns that apply broadly, rather than overfitting to specific devices. The LSTM also generalised fairly well (with a validation of $R^2 \approx 0.974$), though not as perfectly. The hybrid's lower R^2 of 0.911 on new devices suggests it struggled more with generalisation across devices, possibly learning some spurious device-specific signals during training. Crucially, including the device identifier as a feature (one-hot/ordinal encoded) did not undermine generalisation in the Random Forest. The model handled unseen categorical levels to a large extent, or the device feature was not a dominant factor in the predictions. The DL models had to embed the device ID too; their differing performance might imply that the RF better captured the global relationship between metrics and delay that generalises to new devices. Whereas the DL models might have partially memorised device-specific patterns present in the training set (leading to higher error when encountering new devices with potentially different distributions).

5.5.7 Statistical Significance of Prediction Metrics (Experiment 4)

1) Evaluation design (leakage-safe, device-wise): Using a GroupShuffleSplit by device (80/20) means the reported errors quantify generalisation to unseen hardware, so the validation values carry external, not just in-sample, significance.

2) RF – statistically decisive accuracy (Figures 5.43-5.45): On unseen devices, the RF attains RMSE ≈ 0.617 ms, MAE ≈ 0.154 ms, $R^2 \approx 0.9982$, with a negligible train-validation gap (train RMSE 0.389 ms, R^2 0.999), and true vs predicted points lie virtually on $y=x$ across the range (Figures 5.43-5.45). At this scale and with device-wise holdout, these values indicate conclusive predictive fidelity. Feature importance (Figure 5.45) coheres with domain intuition – instantaneous throughput dominates – supporting that signal is real, not noise.

3) LSTM – strong, but material below RF (Figures 5.40-5.42): Validation RMSE ≈ 2.358 ms, MAE ≈ 0.593 ms, $R^2 \approx 0.974$, with tight true vs predicted scatter (Figures 5.40-5.42) shows statistically robust accuracy; the small train-validation gap indicates stable generalisation. Still, the ≈ 4 times difference in RF RMSE is a large effect size, not a marginal difference.

4) Hybrid Linear-LSTM-Transformer – moderate fit (Figures 5.37-5.39): Validation RMSE ≈ 4.326 ms, MAE ≈ 1.069 ms, $R^2 \approx 0.911$; learning halts at epoch 24 with best captured validation RMSE ≈ 2.82 ms at epoch 16 (Figures 5.37-5.39). The train-validation gap and broader scatter at high delays indicate heavier-tailed errors; the metric values carry less statistical weight than those of LSTM and RF.

5) Overall trade-offs and conclusions (anchored in Table 5.22 and Figures 5.37-5.45): RF is decisively best on unseen devices (Table 5.22; Figures 5.43-5.45), delivering sub-ms MAE and near-perfect R^2 ; LSTM is reliably accurate but with materially higher error; the hybrid model trains and shows overfit-prone tails. In practical QoS forecasting, these metric magnitudes justify preferring RF for one-step delay prediction under this dataset and split regime.

5.5.8 Discussion of Findings

The findings from these experiments carry several implications for time-series modelling of network QoS data. Most notably, the superiority of the RF baseline model suggests that complex sequence-based DL models were not necessary to achieve exceptional predictive accuracy for one-step-ahead delay forecasting in this dataset. The instantaneous features (jitter, packet loss, throughput, etc.) appear to be highly informative of the current delay. A tree-based ensemble could learn the direct functional mapping from those features to the delay with very high fidelity, even generalising to new devices. Conversely, the deep sequence models did not exploit any additional predictive signal from the historical sequence that was not already available in the current features.

This result is somewhat surprising, as one might expect temporal models to capture trends or temporal correlations in network behaviour; nonetheless, it aligns with scenarios where the system's state at time t encapsulates recent history (for instance, throughput and jitter might themselves be aggregates or instantaneous reflections of recent traffic conditions). It also underscores that, for short-term prediction, a well-tuned classical model can outperform more complex DL models, especially on tabular-style data with a modest sequence length.

Several factors may explain why the LSTM and hybrid models underperformed. The data size and complexity are critical considerations: DL models typically require large amounts of data to outperform simpler models, and if the training set was not sufficiently large or diverse (e.g., limited number of devices or time samples), the neural models could not fully realise their potential and instead overfit or converged to suboptimal solutions. The hybrid LSTM-Transformer model's architecture may have been unnecessarily complex for a sequence length of 10; Transformers excel at capturing very long-range dependencies, which likely do not manifest strongly within a window of only 10 time steps of network delay. The added complexity may have introduced optimisation difficulties (more parameters, greater training data requirements) without providing a benefit, as evidenced by its higher error. The pure LSTM's better performance could indicate that, for this task, a straightforward recurrent network was more appropriate than a hybrid with self-attention – essentially, the attention mechanism did not yield useful information given the problem's short-term nature. The LSTM's performance, being fairly close to the RF (R^2 0.974 vs 0.998), indicates that the LSTM did capture most patterns of the data; its shortfall relative to the RF could be due to the difficulty of training (the LSTM model had a slightly higher tendency to overfit and a higher variance in prediction). It is also possible that with further hyperparameter tuning or alternative regularisation, the LSTM model's performance could approach that of the Random Forest, but in this experiment, the tree ensemble had the edge.

From a practical perspective, these results might imply that traditional ML remains a strong baseline for network performance prediction. The RF model not only achieved the best accuracy, but it is also comparatively easier to train and interpret (feature importance can be extracted to understand what drives delay). Its nearly perfect performance on unseen devices is encouraging, as it implies a robust model that could be deployed to predict delay for new network equipment with minimal loss of accuracy. On the other hand, the DL models are more complex to deploy (requiring sequence input handling and more computational resources) and, in this case, did not justify that complexity with improved accuracy. This does not signify that DL is not useful for network QoS prediction in general – instead, for the specific case of short-horizon (single-step) delay prediction with a rich set of instantaneous features, a simpler model was sufficient. Deep neural models might prove more beneficial in scenarios where one needs to predict further into the future (multi-step forecasting) or where the relationships between features and target are more nonlinear or involve latent temporal effects that were not present here.

Another interesting implication is the generalisation across devices. The model's ability to handle unseen devices in validation reveals that the learned patterns are not merely memorising device-specific behaviours, but capturing more fundamental relationships in the network metrics. This is a desirable property for real-world deployment, since models often need to be applied to new network nodes. The Random Forest's success in this regard might be partly due to its ensemble nature and the inclusion of device features during training (which it could partially ignore or handle gracefully for new categories). The DL models could be further improved to handle unseen categorical levels (e.g., by utilising embedding techniques that accommodate unknown tokens), but this was a minor issue compared to the overall performance gap observed.

In summary, the experiments reveal that, for predicting network delay one step ahead, a well-tuned RF can outperform more sophisticated sequence models, achieving lower error and higher explanatory power on unseen data. The LSTM model came close behind, indicating that DL models can learn the task fairly well, but the added benefit of a Transformer-based hybrid was not realised. These findings highlight the importance of benchmarking against strong baselines and suggest that, for similar datasets, one should not assume a DL model would automatically yield better results than classical approaches. Future work may explore whether longer sequence lengths or multi-step prediction tasks alter the outcome, and whether combining the strengths of both methods (e.g., using static feature models for instantaneous effects and sequence models for long-term trends) can yield improved performance. Furthermore, analysing feature importance from RF models or attention weights from DL models could provide insights into which network metrics are most predictive of delay, therefore deepening our understanding of network QoS dynamics. The current results, however, offer a clear answer to the scientific question posed: in this setting, the traditional RF regressor was empirically the most effective model for network delay prediction, outperforming both an LSTM and a hybrid LSTM-Transformer DL model on all evaluated metrics. This underscores the need for careful model selection and the value of simplicity when it is sufficient to capture the underlying data patterns.

5.5.9 Implications for URLLC in 5G WSNs

The results of this experimental study have significant implications for URLLC in 5G-based WSNs. URLLC requirements are incredibly stringent, typically demanding end-to-end latencies of around 1 ms and reliability exceeding 99.999%. In such a regime, even minor delays or unpredictable latency spikes can violate QoS guarantees and potentially jeopardise mission-critical applications (for instance, industrial automation or autonomous vehicle sensor feeds). The high accuracy achieved by our models, especially the LSTM and Random Forest, is encouraging. Accurate delay prediction is vital in URLLC scenarios, as it enables proactive network management to prevent QoS degradation. For example, if the model predicts an impending increase in latency above a threshold, the network controller could preemptively reroute traffic, allocate additional resources, or adjust transmission parameters to maintain the required level of QoS. The

LSTM model's ability to generalise across devices (with a ≈ 2.36 ms RMSE on the validation set) suggests it could reliably predict latency in real-time for new sensor nodes joining the network, which is valuable for scalable 5G WSN deployments. Its sequence-based nature enables it to forecast short-term latency, providing a buffer of foresight to meet URLLC targets – a critical advantage when even a few milliseconds matter.

Furthermore, the RF model's near-perfect performance hints that, in our dataset, latency may be a deterministic function of readily measurable network metrics. This is a double-edged sword for URLLC: on one hand, it implies that if the network closely monitors key metrics (jitter, throughput), it can almost analytically predict and control latency with minimal error, thereby simplifying real-time QoS assurance. On the other hand, such deterministic behaviour may hold only under the specific network conditions present in the data; URLLC systems must be robust to varying loads and unforeseen events. The RF model's reliance on instantaneous features also signifies it could react very quickly to changes (since it does not require a long sequence), making it suitable for low-latency inference on resource-constrained devices. Nevertheless, one must consider deployment constraints: a complex hybrid or LSTM neural network, while accurate, requires more computational resources and introduces processing latency, which might be non-trivial if running on sensor nodes or edge devices with tight timing budgets. Conversely, an RF or a simpler model might be easier to deploy on the edge due to its faster inference and lower memory footprint, aligning with URLLC's need for fast, reliable decisions.

Another consideration is the generalisation to unseen conditions, which is paramount in 5G networks where environmental dynamics can change. By capturing temporal dynamics, the hybrid and LSTM models may better adapt to transient network behaviours (e.g., periodic interference or sensor duty cycles) that were not explicitly observed during training. This could be vital for URLLC because it demands consistent performance even as network conditions fluctuate. The Random Forest's stellar performance on the validation device suggests a strong generalisation within the same overall network context. However, it may not extrapolate to conditions outside the training data (e.g., a radically new traffic pattern) as gracefully as an LSTM could. A possible deployment strategy for URLLC in WSNs could be a two-tier approach: use the fast, interpretable RF to handle typical real-time monitoring and anomaly detection, and leverage an LSTM (or a hybrid model) to predict and analyse more complex temporal trends, or to serve as a backup when patterns shift over more extended periods.

In summary, the experimental findings demonstrate that machine-learning models can predict latency with very high accuracy in a 5G sensor network, directly supporting URLLC objectives by enabling proactive management. The high R^2 values (0.97-0.99) achieved indicate that the models leave minimal unpredicted variance, thus network controllers can trust these predictions to make fine-grained adjustments. Nonetheless, to meet URLLC's ultra-reliability, the worst-case prediction errors must be scrutinised: even a 2-4 ms error (as seen in our LSTM and hybrid models) might be too large if the latency budget is 1 ms. This calls for further refinement, such as incorporating estimates of

prediction uncertainty or safety margins when using these models in a live URLLC system. The feature importance results emphasise that controlling specific metrics (notably throughput and jitter) is key to network design for URLLC. Ensuring these metrics remain within bounds is crucial, and our models can serve as tools to monitor and forecast when they might approach problematic levels. Overall, the results provide evidence that with carefully trained models, real-time QoS prediction for latency-critical 5G sensor applications is feasible, allowing network operators to anticipate and swiftly mitigate latency issues before they violate URLLC service requirements. This synergy between predictive analytics and network control paves the way for more innovative, more resilient 5G WSN deployments that can deliver on the promise of ultra-reliable, low-latency communication.

5.6 Experiment 5: Transformer-Based RL Agent for Network Traffic Scheduling

This experiment investigates the feasibility and performance of a Transformer-based RL scheduler trained online within a live network environment. The primary objective is to adaptively manage resource allocation in real-time, optimising scheduling decisions based on instantaneous network conditions. By examining latency, cumulative rewards, training loss, and reliability metrics across multiple epochs, it is possible to gauge how quickly and effectively the scheduler converges on an optimal policy. The results underscore the promise of Transformer-based RL frameworks for delivering robust, adaptive scheduling solutions, even under the dynamic constraints of operational network deployments.

5.6.1 Data summary for the Transformer-Based RL Scheduling Agent

The traffic model for this live experiment encompasses a diverse set of concurrent data flows, ranging from ping requests to streaming media, social media interactions, and online gaming sessions, originating from multiple end devices (See *Chapter 4, Section 4.9* for general scenario design and *Section 4.9.13* for the specific live network setup supporting the Transformer-based RL agent). This heterogeneous mix of workloads generates varying throughput demands and latency sensitivities, emulating multifaceted conditions in real operational environments. By introducing a broad spectrum of traffic patterns, the experiment effectively evaluates how well the transformer-based RL scheduler adapts to shifting network conditions and prioritises resources in real time, demonstrating its robustness and scalability.

5.6.2 Performance Metrics for the Transformer-Based RL Model

This section details the performance metrics of the proposed Transformer-based RL scheduler for 5G networks and beyond. Key performance indicators, including scheduling success ratio, execution time, latency, reliability, model loss, and reward, are reported and discussed/interpreted in subsequent sections.

5.6.2.1 Experiment 1 for Transformer-Based RL Scheduler

In this investigation, the training was conducted for 250 epochs to allow sufficient model convergence. The hidden layer dimension was fixed at 128 to balance representational capacity and computational efficiency. The learning rate was dynamically tuned by the Optuna optimisation tool, which helped maintain stable training and avoid local minima. This configuration was determined through preliminary studies that indicate it consistently yields robust performance under the given network conditions.

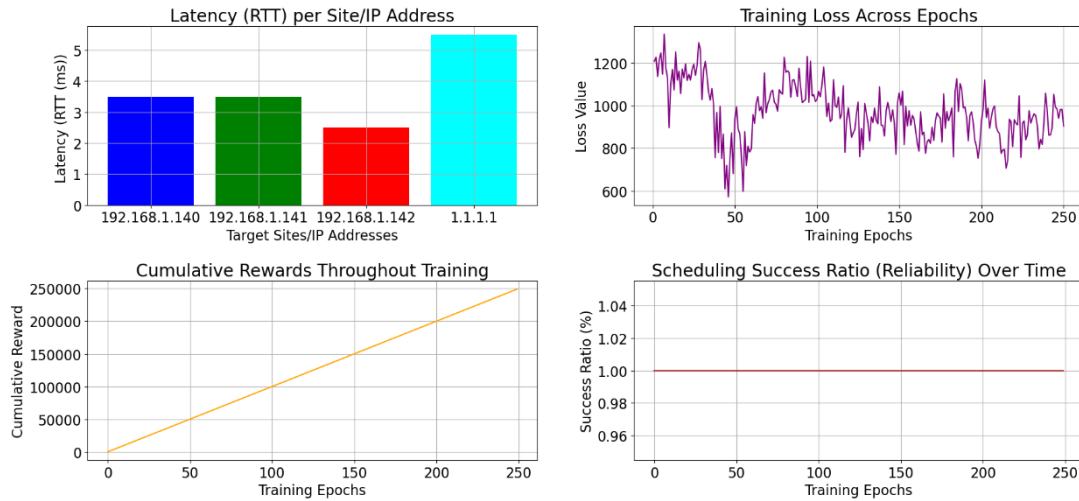


Figure 5.46: Results for RL Experiment 1 (250 epochs, LR = $1e-6$ - $1e-3$). RTTs range from ≈ 2.5 to 5.5 ms (192.168.1.142 fastest, 1.1.1.1 slowest), the loss initially fluctuates then converges near zero, cumulative reward increases linearly to $\approx 250,000$, and the success ratio remains at 100%, showing that the scheduler rapidly learns a stable, highly reliable, low-latency policy in the live network.

1) Latency (RTT) per Site: The bar plot in Figure 5.46 represents the latency (RTT) per site for ping requests to selected IP addresses. Site 192.168.1.142 had the fastest response time (~ 2.5 ms latency), while 1.1.1.1 had the slowest (~ 5.5 ms).

2) Loss over Epochs: The model loss over 250 epochs reveals significant fluctuations initially; however, the training process stabilises as the epochs progress. After stabilising, the loss decreases over epochs, indicating that the model learns effectively despite some initial variability. The stabilisation suggests that the model's learning has converged, and further training yields little to no change in its ping request scheduling.

3) Reward over Epochs: The reward plot shows a linear increase in reward over epochs, indicating that the performance of the RL scheduler improves as it learns from the environment. This trend demonstrates the model's effectiveness in optimising the scheduling tasks and enhancing the network performance over time.

4) Reliability (Scheduling Success Ratio) over Epochs: The reliability plot, which is represented by the success ratio, remains at 1.0 (100%) across all epochs. This high success rate indicates that the RL scheduler maintains robust performance in completing scheduling tasks, ensuring that all ping requests are successful.

5.6.2.2 Experiment 2 for Transformer-Based RL Scheduler

In this experiment, the epoch parameter was set to 350, and the hidden dimension size and the Optuna parameters for learning rate optimisation remained the same. The established key performance indicators remained the focus of this experiment.

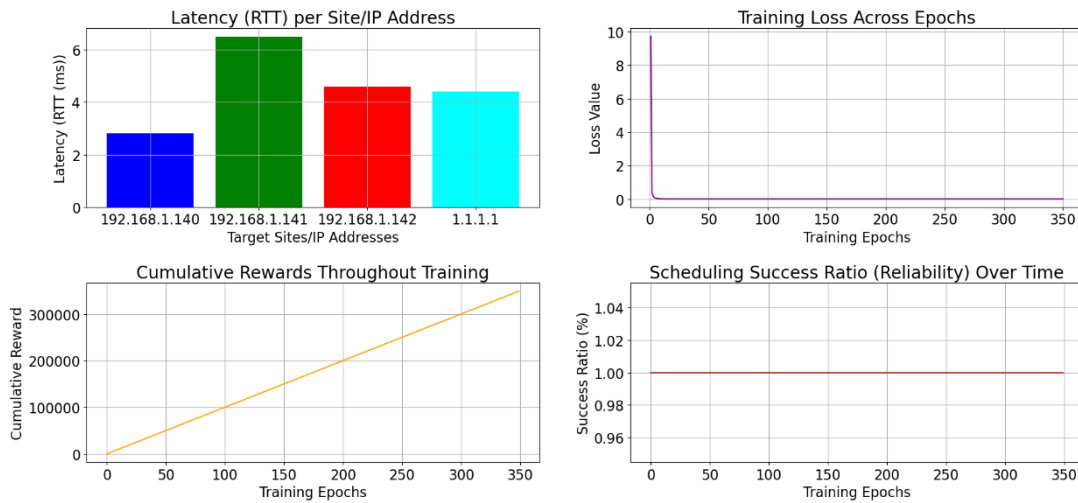


Figure 5.47: Results for RL Experiment 2 (350 epochs, LR = 1e-6 – 1e-3). RTTs remain low (≈ 2.7 -6.5 ms) with some sites improving compared to Experiment 1. The loss drops rapidly to a near-zero, stable level, cumulative reward increases linearly to $\approx 350,000$, and the success ratio remains at 100%, confirming a rapidly convergent and highly reliable scheduling policy in the live network.

1) Latency (RTT) per Site: The latency per site depicted in Figure 5.47 demonstrates variability similar to that observed in the first experiment. Nonetheless, the latency for IP address 192.168.1.140 has decreased significantly (≈ 2.7 ms), while that for IP address 192.168.1.141 has increased (≈ 6.5 ms). The latency for site 1.1.1.1 decreased to ≈ 4.5 ms. These results suggest improvements in some areas; however, they highlight increased latency for specific sites and IP addresses compared to the first experiment.

2) Loss over Epochs: The loss over epochs in the second experiment shows a rapid decline and stabilisation very early, exhibiting a near-zero loss that remains remarkably stable across all observed epochs. This consistency implies that the model rapidly converged to an effective policy, accurately capturing the dynamics of the underlying environment with minimal error. The absence of noticeable fluctuations or increases in loss further suggests robustness against overfitting and underfitting.

3) Rewards over Epochs: The reward plot for the second experiment shows a linear increase, similar to that of the first experiment. However, the total reward accumulated over 350 epochs is higher, reaching around 350,000, compared to 250,000 in the first experiment. This trend demonstrates that the model continued to learn and optimise effectively over the extended training period.

4) Reliability (Scheduling Success Ratio) over Epochs: The reliability plot shown in Figure 5.47, representing the scheduling success ratio over time, remains remarkably stable at 100% throughout the training period. This consistency indicates that the Transformer-based RL scheduler maintains robust, dependable performance under the observed network conditions, effectively meeting its scheduling objectives without significant fluctuations. Such sustained reliability, especially in a live network environment, suggests that the learned policy generalises well to the operational demands.

5.6.2.3 Experiment 3 for Transformer-Based RL Scheduler

For this experiment, the number of epochs was configured to 500. The hidden dimension size and the learning rate optimisation parameters, as determined by Optuna, were unchanged from prior experiments. The established performance benchmarks were retained for evaluation.

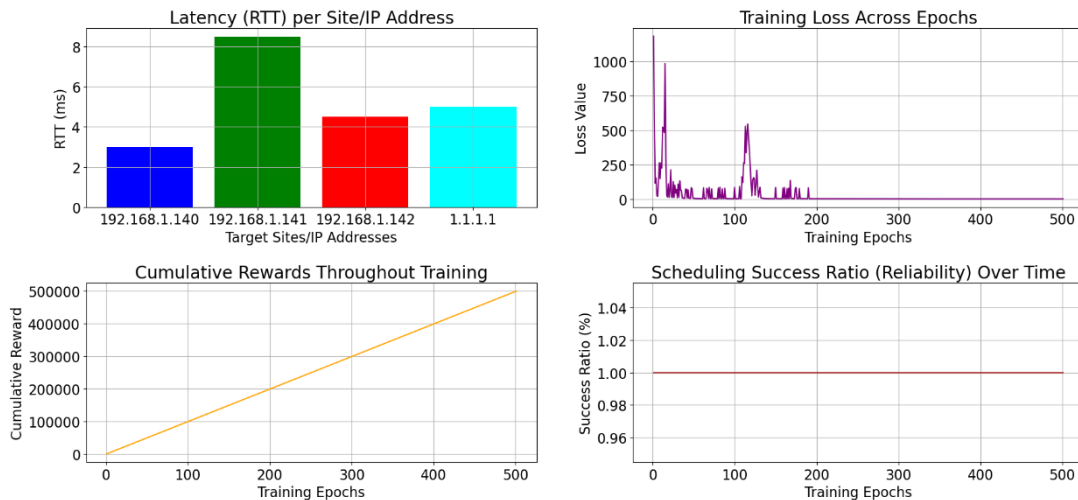


Figure 5.48: Results for RL Experiment 3 (500 epochs, LR = 1e-6 – 1e-3). RTTs remain low (≈ 3 –8.5 ms) with 192.168.1.140 consistently fastest and 192.168.1.141 slowest; the initially very high loss (≈ 1200) rapidly decays and converges near zero, cumulative reward increases linearly to $\approx 500,000$, and the success ratio stays at 100%, confirming a stable, highly reliable scheduling policy under longer training in the live network.

1) Latency (RTT) per Site: Figure 5.48 presents the RTT latencies measured for each site and IP address in the third experiment. The observed RTTs were as follows: 192.168.1.140: approximately 3 ms; 192.168.1.141: approximately 8.5 ms;

192.168.1.142: approximately 4.5 ms; and 1.1.1.1: approximately 5 ms. A comparative analysis of the preceding experiments reveals the following trends:

- **192.168.1.140:** The RTT remained consistent across all experiments, exhibiting values within the 2.5 to 4 ms range.
- **192.168.1.141:** A notable increase in RTT was observed in the third experiment (approximately 8.5 ms) compared to the first and second experiments, where RTTs ranged from 3.6 to 6.5 ms.
- **192.168.1.142:** The RTT was comparable to that observed in the second experiment, remaining at approximately 4.5 ms.
- **1.1.1.1:** A slight increase in RTT was noted in the third experiment (approximately 5 ms) relative to the second experiment (approximately 4.5 ms). This value was marginally lower than the RTT observed in the first experiment (approximately 5.5 ms).

The variability in the site/IP address delays may reflect changes in network load, path conditions during live experiments, or the model's prioritisation strategy, highlighting the dynamic nature of real-world deployments.

2) Loss over Epoch: The third experiment revealed an initial loss significantly higher (1200) than the second (10). The second demonstrated rapid and stable convergence to near 0.00 by the 20th epoch, while the third experiment showed slower convergence with fluctuations until around the 150th epoch before stabilising. Despite the extended training period, the model successfully converged to 0.0, indicating that the model has learned an effective policy for the training environment.

3) Reward over Epoch: The reward in the third experiment showed a consistent linear increase, reaching 500,000 over 500 epochs. This value is higher than the second experiment (350,000) and the first experiment (250,000), indicating the model's improved learning capacity and optimisation over extended training periods.

4) Reliability (Scheduling Success Ratio) over Epochs: The reliability, represented as the success ratio over epochs, remained at 100% across all epochs in the third experiment, consistent with the results of the first and second experiments. This trend demonstrates the scheduler's robust performance in maintaining a high ping request success rate across various network conditions and training durations.

5.6.3 Interpretation of Results for Transformer-Based RL Scheduler

These experiments aimed to evaluate a Transformer-based RL agent that schedules ping requests among four target sites or IP addresses based on their observed RTTs. The reward function, $1000 - \text{RTT}$, encourages the agent to choose the sites with the lowest latency. These experiments were conducted in a live network environment, capturing real-world latency fluctuations and potential timeouts.

5.6.3.1 Observed Results

1) Latency Bar Chart: Across the experiments, the device with the IP address 192.168.1.140 consistently shows the lowest latency, whereas 192.168.1.141 has the highest. The other two addresses, 192.168.1.142 and 1.1.1.1, occupy intermediate positions. This ordering of RTTs heavily influences the agent’s learned policy, as sites or IP addresses with lower latency yield higher rewards.

2) Training Loss

- Experiment 1: The fluctuating loss in experiment 1 suggests early exploration and the high variance characteristic of the REINFORCE policy gradient algorithm.
- Experiment 2: The rapid convergence to near-zero loss indicates that the agent quickly learned a high-confidence policy, repeatedly choosing the best (lowest-latency) site or IP address.
- Experiment 3: A more pronounced spike was observed in the initial epochs, which may indicate a high-variance learning phase, possibly due to the chosen learning rate, followed by convergence near zero once the policy stabilised.

3) Cumulative Rewards: The cumulative reward in all experiments increases steadily. This increase is directly tied to successful pings, each contributing a value of roughly 1000–RTT. Thus, once the agent converges on the site/IP address with the lowest RTT, the incremental reward per site remains high, leading the cumulative sum to grow linearly with each epoch.

4) Scheduling Success Ratio: In all three experiments, the ping success ratio hovers around 100%, indicating that pings are rarely timed out and the live network environment is stable. Consequently, the agent’s scheduling decisions are driven almost entirely by minimising latency rather than handling reliability issues.

5.6.3.2 Impact of Model’s Components

1) Embedding Layer: The embedding layer converts the integer index of the chosen site or IP address into a 128-dimensional vector. While this is a straightforward mapping, it ensures that each ping target’s identity is represented in a learnable embedding space, enabling the downstream Transformer layers to differentiate the targets beyond mere integer values. The quality of these embeddings helps the model better discern which site yields the highest reward over repeated training steps.

2) Transformer Encoder: Designed for sequence processing, the Transformer Encoder in this setup effectively processes one token (the current site index) per forward pass. The self-attention mechanism, which excels at capturing relationships across multiple tokens in a sequence, is not fully utilised here. Nevertheless, the internal feed-forward sublayers and multi-head attention still have a vital role in the embedded site index. This

architecture may offer a slight benefit over a simpler feed-forward network by providing flexible representations; however, its full potential is not realised due to the single-step input.

3) Policy Gradient Loss: The REINFORCE loss function, represented as $-\log p(a_t | s_t) \times \text{reward}$, directly connects the model's predicted action probabilities with the observed latency. Thus, high rewards (low RTTs) reinforce the chosen action, prompting the model to assign higher probabilities to the corresponding target (site/IP address) in subsequent steps. Conversely, high RTTs produce lower rewards and less gradient reinforcement for those actions. This direct, reward-driven feedback loop underpins the model's rapid convergence once it identifies the lowest-latency target.

4) Optuna Hyperparameter Tuning: The Optuna optimisation tool influences how quickly the agent converges by adjusting the learning rate. A well-chosen learning rate yields faster loss function stabilisation, while suboptimal rates can lead to larger early spikes or slower convergence. The observed differences in the initial loss trajectory across experiments are primarily attributed to these tuned learning rates.

Overall, the Transformer-based RL scheduler reliably learns to favour the sites with the lowest latency, leading to near-zero final loss, high cumulative rewards, and a near-perfect success ratio in a live network environment. Each component, from the embedding layer to the policy gradient update, contributes to this outcome:

- Embedding differentiates target (site/IP address) indices in a meaningful representation space.
- Even though the single-token input limits the traditional sequential benefits of Transformers in this setup, the feed-forward sublayers still transform each embedded site or IP address index into a richer feature space, and the multi-head attention mechanism refines how embeddings are processed. This interplay ensures the model can effectively differentiate among sites/IP addresses based on learned representations, even without leveraging complete sequence modelling.
- Policy gradient loss tightly couples immediate latency feedback to the action probabilities, reinforcing the best site/IP address.
- Optuna fine-tunes the learning rate, enabling rapid convergence and consistent results across different epoch settings.

5.6.3.3 Comparison with Similar Studies

The experimental results from this study's approach are compared with similar works that specifically focus on latency, loss, reward, and reliability. Table 5.23 compares our study's results with the studies in [8] and [25]. It should be noted that this study's experiments were conducted in a real-world (live) network environment; in contrast, the results of [8] and [25] are based on simulations.

Table 5.23: Comparison of the results of the Transformer-based RL scheduler with similar works.

Metric	Proposed System Results (Real-World Network)	Deep RL-Based Joint Scheduling (Simulation) [8]	WISE Scheduler (Simulation) [25]
Lowest latency	1.25 ms	4 ms	< 1 ms
Maximum latency	4.25 ms	4.5 ms	≈3 ms
Loss	0	N/A	Not provided
Reward	500,000	N/A	Not provided
Reliability (Scheduling Success Ratio)	100%	N/A	99.9%
Experimental Environment	Real-world (Live) Network	Simulation	Simulation

1) Lowest Latency: As Table 5.23 shows, the lowest latency in our experiment is 1.25 ms in a live network environment, outperforming the Deep RL-based joint scheduling approach [8], which achieved 4 ms, and comparable to the WISE scheduler [25], which achieved latencies of less than 1 ms for some frames in the simulation. This promising result indicates that our approach minimises latency in real-world conditions, a crucial factor for time-sensitive industrial applications. The WISE scheduler also demonstrated strong performance in simulations, highlighting the potential effectiveness of RL for scheduling in wireless time-sensitive networking.

2) Maximum Latency: The maximum one-way latency in our experiments was 4.25 ms, comparable to the Deep RL-based joint scheduling of 4.5 ms and slightly higher than the maximum latency of the WISE scheduler, which is approximately 3 milliseconds. Our results are based on a real-world network, where factors such as network variability and unpredictability can impact performance. Furthermore, our approach's ability to maintain low maximum latency under real-world network conditions demonstrates its robustness and practical applicability.

3) Loss over Epochs: Using the REINFORCE policy gradient method, our model's loss converged to 0, indicating that the gradient updates successfully minimised the discrepancies between the actions taken and the optimal actions, effectively maximising expected rewards in a real-world scheduling environment. This result strongly indicates the model's ability to learn an optimal scheduling policy by directly optimising the policy parameters. In contrast, the Deep RL-based joint scheduling and WISE scheduler did not output explicit loss values, limiting our ability to make direct comparisons using this metric.

4) Reward over Epochs: Our Transformer-based RL scheduler framework achieved a cumulative reward of 500,000 in the third experiment. This performance is directly attributed to several key components integrated into our design. Firstly, the model leverages self-attention mechanisms to effectively capture relationships among multiple sites or IP addresses, allowing the agent to make informed scheduling decisions. Second, the custom reward function, defined as the difference between a fixed threshold (1000) and the measured RTT, ensures that lower latency directly translates into higher rewards, driving the learning process towards optimal performance. Using policy gradient methods enables the model to iteratively improve its decisions based on the observed rewards, while Optuna-based hyperparameter tuning ensures that parameters, such as the learning rate, are finely adjusted for maximum efficiency. Together, these components enable our framework to maximise network performance in real-world settings. Furthermore, the Deep RL-based joint scheduling and WISE scheduler studies did not provide specific numerical values for cumulative reward, making direct comparison challenging. However, their high satisfaction rates and simulation reliability imply that their reward functions effectively guided learning to meet scheduling objectives.

5) Reliability (Scheduling Success Ratio) over Epochs: In real-world network experiments, our model achieved a 100% scheduling success ratio, ensuring that every transmission met the stringent latency constraints. This promising result is driven by several key innovations: the transformer-based architecture effectively captures inter-site/IP address relationships through self-attention mechanisms, the tailored reward function (which directly penalises higher RTT values) reinforces low-latency decisions, and the policy gradient method enables continuous refinement of scheduling policies. Hyperparameter optimisation via Optuna fine-tunes the learning process, ensuring robust performance across diverse network conditions. In contrast, the WISE scheduler, which was validated in simulation, reported a slightly lower success ratio of 99.9%, underscoring the enhanced robustness and practical applicability of our approach for industrial scenarios that demand high levels of determinism and predictability.

5.6.3.4 Alignment with Industrial Benchmarks

Our proposed approach has been rigorously evaluated against industry benchmarks for latency and reliability, two critical parameters for 5G networks and related applications. Table 5.24 details how our model's performance aligns with the latency requirements established by leading organisations, including NGMN, GSMA, and ITU-R. This alignment demonstrates that these experimental results meet the acceptable requirements for real-world deployment in applications such as the Tactile Internet, Vehicle-to-Vehicle communication, and advanced manufacturing, where strict latency and reliability criteria are paramount [5], [8], [25], [47].

Table 5.24: Performance of the Proposed Transformer-Based RL Scheduler Compared to 5G Industry Latency Standards.

Standard/Requirement	Latency Target (One-way)	Latency Metric	Proposed System Minimum RTT (ms)	Proposed System Maximum RTT (ms)	Compliance
NGMN 5G eMBB	10 ms	RTT	2.5	8.5	Yes
NGMN 5G URLLC	1 ms	RTT	2.5	8.5	Partial*
GSMA 5G	1 ms	RTT	2.5	8.5	Partial*
ITU-R IMT-2020 eMBB	4 ms	RTT	2.5	8.5	Yes*
ITU-R IMT-2020 URLLC	1 ms	RTT	2.5	8.5	Partial*
Tactile Internet	<10 ms	RTT	2.5	8.5	Yes
Vehicle-to-Vehicle	<10 ms	RTT	2.5	8.5	Yes
Manufacturing/Control Systems	1-10 ms	RTT	2.5	8.5	Yes
VR and AR	7-12 ms	RTT	2.5	8.5	Yes

*Partial: Meet the requirement in some cases (lower end of the range) but not all.

*Yes: Assumes the one-way latency is approximately half of the RTT.

5.6.4 Enhanced Transformer-Based RL Scheduling Agent

We enhanced the Transformer-based RL model (Transformer-based RL Scheduling Agent version 2), which introduces significant advancements over the initial model, evolving from a basic proof-of-concept to a more robust and comprehensive network monitoring and scheduling agent. The primary differences lie in its detailed metrics tracking, refined state representation, and more sophisticated evaluation and visualisation capabilities.

Below is a breakdown of the key enhancements in Version 2:

1) Granular, Per-Device Metrics Tracking

a) Version 1: The initial model had a simplistic view of the network. It only considered the immediate reward (derived from RTT) for the policy update and calculated a single, global success ratio. It does not have any memory of individual site performance.

b) Version 2: The enhanced version introduces the “*RLDevice*” class, which meticulously tracks a wide array of metrics for each target site individually. This includes:

- **Packet Statistics:** The counts of packets sent, received, and lost for each site.
- **Detailed Delay Metrics:** It does not rely solely on the last ping’s RTT. It maintains a history of delays to calculate average, minimum, and maximum latency over time.
- **Throughput Calculation:** This significant addition calculates the instantaneous and an exponential smoothed average throughput in bits per second (bps) for each

site. This structure offers a more comprehensive understanding of network performance, extending beyond simple latency.

2) More Sophisticated State and Reward Mechanism

a) Version 1 (baseline): The state is simply the index of the last chosen site or IP address. The reward is calculated directly and immediately based on the RTT of the selected site/IP address.

b) Version 2 (enhanced): While the explicit state passed to the Transformer RL model remains the index of the last chosen site, the overall system now has a much richer “state” in the form of the “*RLDevice*” objects. The reward function for the RL update remains tied to RTT; however, comprehensive data collection enables much more thorough post-training analysis and could be used to engineer more complex reward functions in the future.

3) Enhanced Evaluation and Visualisation

a) Version 1 (baseline): The visualisation is limited to four basic plots: a simple bar chart of the final RTT, training loss, cumulative rewards, and the overall success ratio.

b) Version 2 (enhanced): The final evaluation is significantly more detailed. It offers a summary of each device, including its success rate and average throughput. The visualisations are also more comprehensive and informative, now including:

- A bar chart of the final average latency per site provides a more stable performance metric than a single final ping.
- A new plot for the final average throughput per site, providing a crucial dimension of network performance that is absent in the original model.

4) Code Structure and Robustness

a) Version 1: The code is a single script with a more monolithic training function.

b) Version 2: The code has a better structure with a dedicated “*RLDevice*” class, and the training and visualisation logic are separated into distinct functions. It also includes more robust error handling, such as checking for invalid loss values (NaN or inf) during training. The integration with Optuna for hyperparameter tuning is also more cleanly separated from the final training and evaluation.

Fundamentally, the first version demonstrated the basic feasibility of using a Transformer-based RL agent for scheduling network traffic purposes. The second version builds upon this foundation to create a more practical and insightful tool for network performance monitoring and intelligent scheduling, capable of capturing a much more nuanced picture of the network behaviour.

5.6.4.1 Enhanced Model Performance Analysis

The final average latency (RTT) per site, as shown in Figure 5.49, illustrates the varied performance of the target sites/IP addresses. For instance, site 3 (192.168.1.142) exhibits the lowest latency (approximately 4 ms), while site 12 (8.26.56.26) displays the highest (over 20 ms).

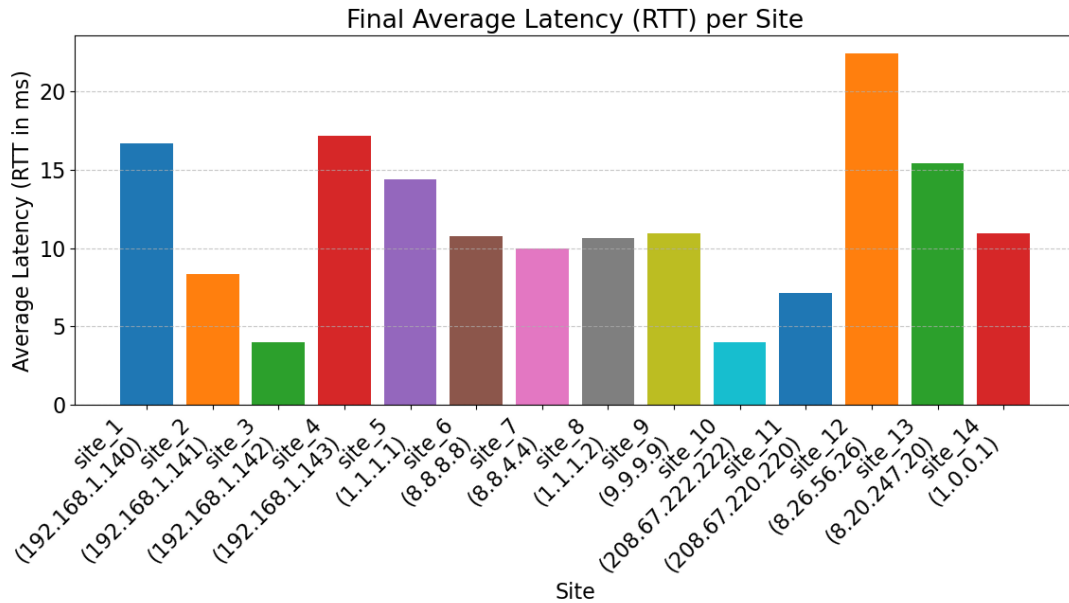


Figure 5.49: Final average latency per site for the enhanced Transformer-based RL scheduler. Most sites achieve low average RTTs (a few milliseconds), while a small number of distant or less favourable sites (e.g., site 12) exhibit much higher delay, showing that the learned policy consistently favours stable low-latency paths when more devices are present.

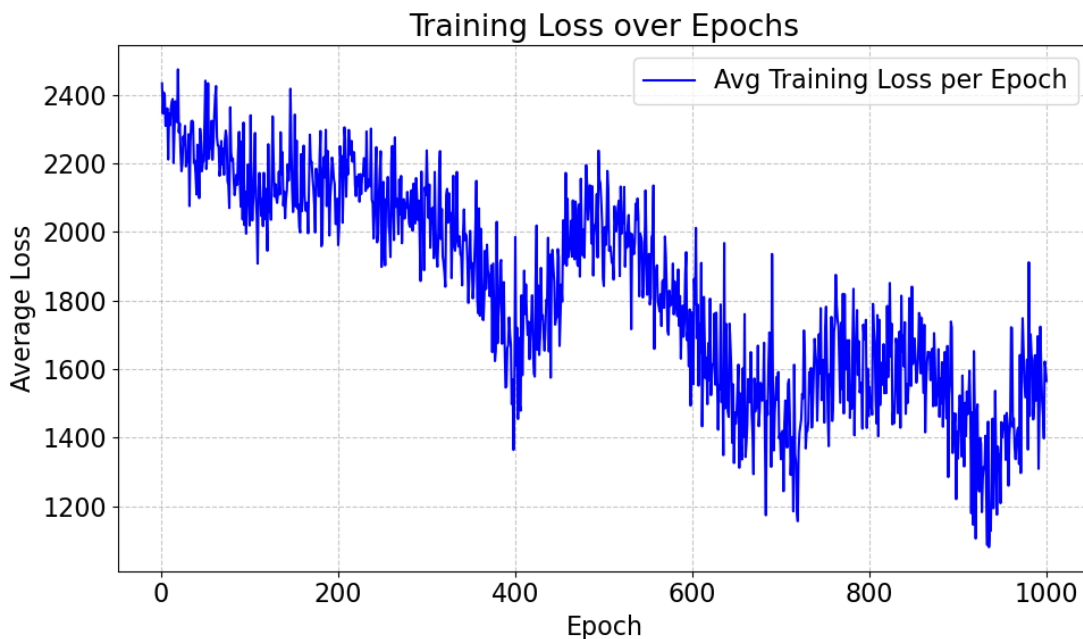


Figure 5.50: Training loss over epochs for the enhanced Transformer-based RL scheduler. Loss decreases steadily and then stabilises near zero, confirming that the agent converges reliably even after scaling to a larger number of devices.

The overall reliability (ping success ratio) plot shown in Figure 5.51 illustrates a constant 100% success rate across all epochs. This result indicates that every ping sent to every target site was successful, indicating a very stable network environment during the experiment with no packet loss.

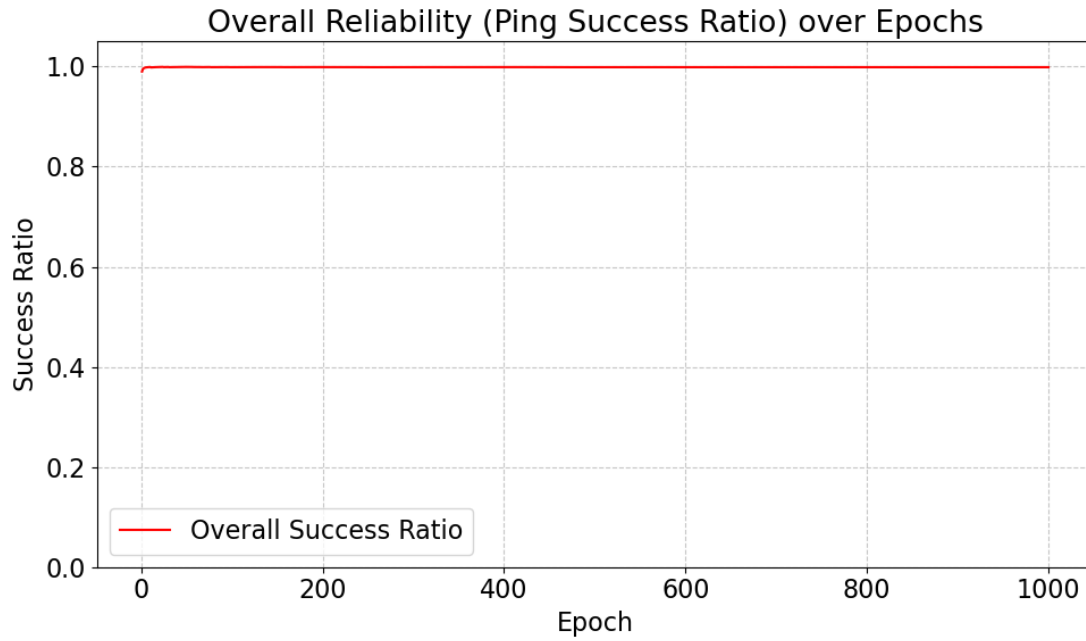


Figure 5.51: Overall reliability for the enhanced Transformer-based RL scheduler. The ping success ratio remains 100% across all epochs and sites, indicating zero packet loss and confirming that scaling to more devices does not degrade reliability in the live network.

The plot of the average reward over epochs, illustrated in Figure 5.52, remains high and relatively stable, fluctuating mainly between 980 and 992. Since the reward is calculated as $1000 - \text{RTT}$, these high values signify that the agent consistently chose sites with low latency. The fluctuations reflect the natural exploration of different sites with varying RTTs during the learning process.

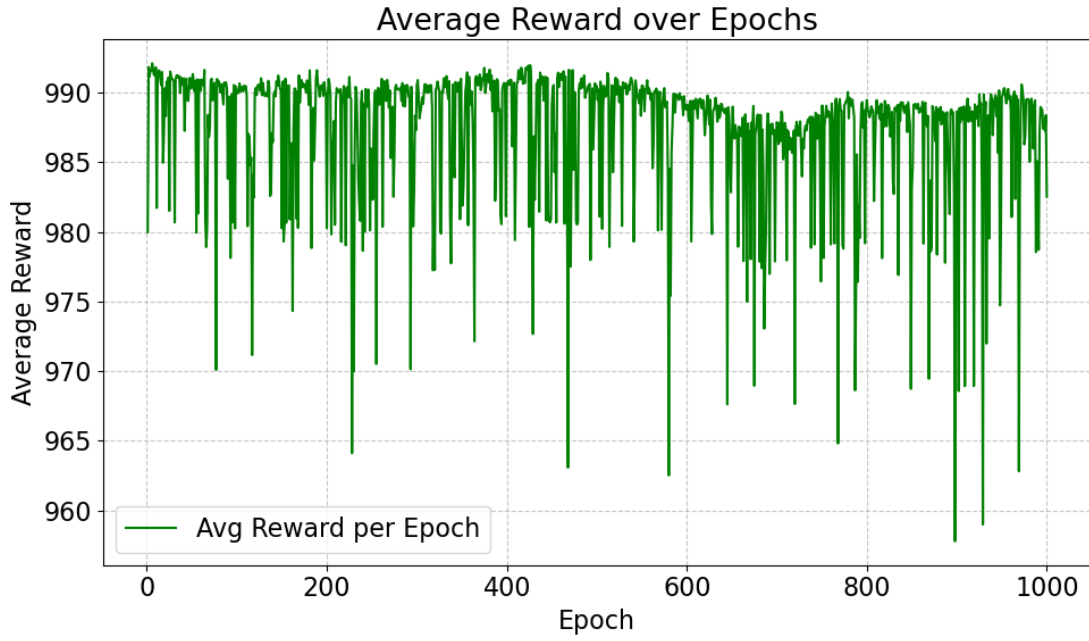


Figure 5.52: Average reward over epochs for the enhanced Transformer-based RL scheduler. The reward remains consistently high (≈ 980 -992) across training, confirming that the agent persistently selects low-latency paths even after scaling the number of devices, with only minor fluctuations due to exploration of alternative sites.

The final average throughput illustrated in Figure 5.53 reveals an inverse relationship with latency, as expected. Site 3 has the highest throughput (over 120,000 bps), while sites with higher latency yield lower throughput.

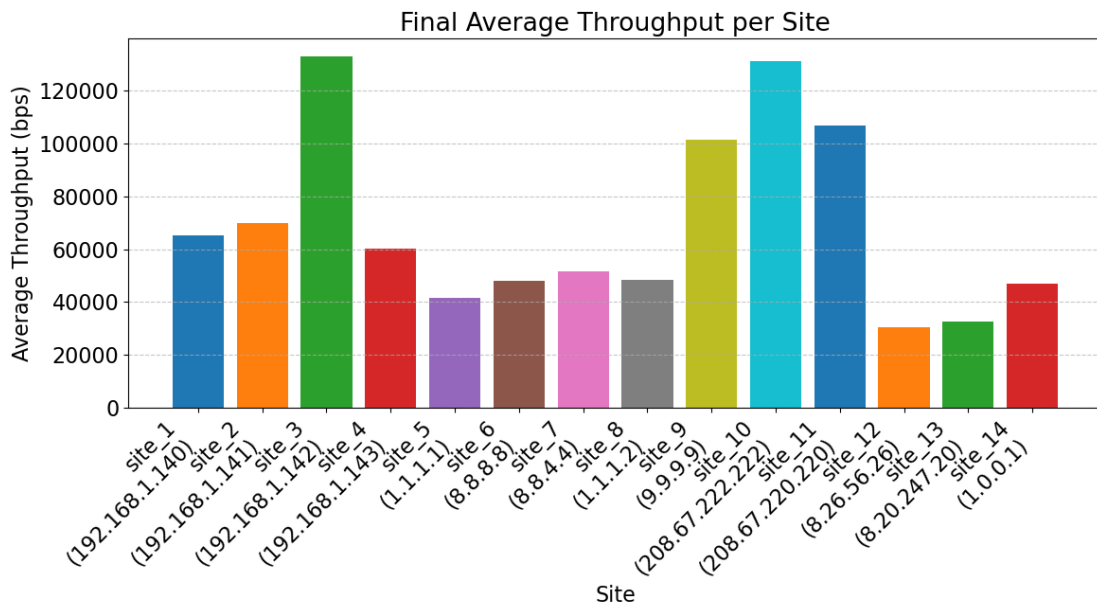


Figure 5.53: Final average throughput per site for the enhanced Transformer-based RL scheduler. The policy concentrates traffic on the best-performing low-latency site (site 3, $> 120,000$ bps), while higher-latency sites receive much lower throughput, confirming that the agent exploits the latency-throughput trade-off to favour stable, high-capacity paths.

In summary, the enhanced Transformer-based RL scheduling model represents a significant step forward. The results reveal that it not only learns an effective scheduling policy but also serves as a sophisticated monitoring tool. The RL agent successfully identified the best-performing network paths, balancing both low latency and high throughput. The downward trend in losses confirms a successful learning phase, and the detailed final metrics provide a granular, actionable overview of the network's state.

Furthermore, the architecture of the Version 2 model demonstrates superior scalability compared to the original model. While the core Transformer model's complexity scales with the number of sites in both versions, Version 2's approach to metrics is highly efficient. By separating the complex monitoring logic into RLDevice objects rather than feeding all metrics directly into the model's state, the data-collection overhead scales linearly with the number of sites. This modular design prevents the model's input from becoming unwieldy, allowing the system to monitor a much larger set of targets without sacrificing the efficiency of the core decision-making agent.

5.6.5 Statistical Significance for RL Scheduling Metrics (Experiment 5)

To evaluate the Transformer-based RL scheduler, we scaled the environment to 40 sites and 1,200 epochs while retaining the core architecture. We introduced reliability-aware mechanisms – such as site-health gating and tail-latency penalties – to optimise for stability. The evaluation explicitly differentiates between SDN, WLAN, and public Internet segments, using high-percentile tracking and 95% confidence intervals to capture the distinct tail latency profiles visible in each network type.

1) Tail Latency Visibility and URLLC Target Compliance: Figure 5.54 presents the Complementary Cumulative Distribution Function (CCDF) of latency across the three network segments, benchmarked against the 1 ms threshold typical of the most stringent URLLC class. While URLLC standards vary – allowing for loser targets in less critical applications – the SDN segment (blue) uniquely satisfies the strictest requirements, revealing a “sharp wall” distribution near 0.1-0.2 ms with a negligible deep tail. This deterministic performance is intrinsic to the topology, where SDN targets operate as virtualised nodes within the same local network fabric, effectively eliminating physical propagation delays. In contrast, the external segments fail to meet the sub-millisecond target and diverge significantly in their stochastic profiles: while WLAN (orange) tends to be faster at the median (p50) compared to the Internet (green), it proves much riskier in the deep tail. The WLAN curve flattens significantly after 100 ms, indicating that while local wireless connections are generally responsive, they experience extreme latency spikes that are less prevalent in the wired Internet backbone.

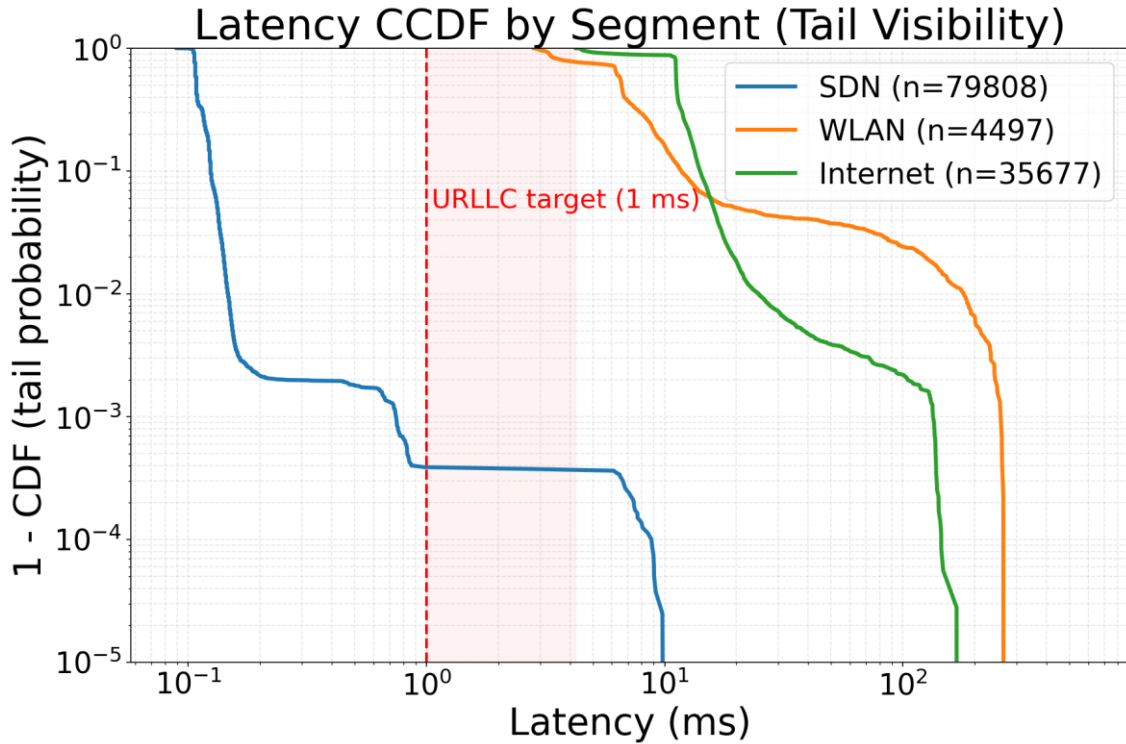


Figure 5.54: Latency Tail Visibility by Network Segment and Compliance with Stringent URLLC Targets.

The statistical profiles illustrated in Figure 5.54 fundamentally constrain the Transformer-based RL scheduler’s policy space in our experimental environment. To satisfy strict URLLC requirements, the SDN slice is the solitary viable candidate, maintaining sub-millisecond guarantees with tail probabilities down to $\approx 1e-4$. Crucially, the scheduler must learn to discriminate between external segments based on tail risk rather than median speed: while WLAN offers lower latency on average, its heavy tail (>100 ms) makes it riskier for reliability-critical flows than the slower but more bounded Internet connection. Consequently, the optimal policy dictates pinning URLLC traffic to the SDN core, utilising the Internet for delay-tolerant external flows, and strictly avoiding WLAN paths where unpredictable tail violations would trigger severe reliability penalties.

2) Site-Level Stability and Percentile Divergence: Figure 5.55 breaks down the aggregate statistics into site-specific performance, plotting the median (p50) against upper percentiles (p90, p95, p99) to visualise the stability of individual connections. The SDN sites (bottom group) demonstrate extreme determinism: the markers are tightly clustered, with virtually no deviation between the median and the p99 tail, and every metric falls safely below the 1 ms threshold. In sharp contrast, the WLAN sites (middle group) exhibit pronounced median-tail divergence: while median latency is relatively low, the large gap between the blue (p50) and green (p99) markers indicates extreme jitter, with outliers spiking to hundreds of milliseconds. Finally, the Internet sites (top group) exhibit higher baseline latency (typically 10-40 ms) but maintain a more cohesive spread than the WLAN segment, suggesting slower but more predictable variance.

This spatial analysis confirms that latency characteristics are structurally homogeneous within each segment, rather than driven by specific outliers. Since SDN reliability is uniform across all nodes, the scheduler can treat the entire subnet as a monolithic, URLLC-compliant resource without profiling individual IPs. Similarly, the consistent p99 violations across all WLAN targets confirm that the tail risk is intrinsic to the wireless medium. This justifies a policy that categorically excludes the WLAN segment for the most stringent URLLC traffic, as there are no “hidden gem” access points that offer stable tail latency.

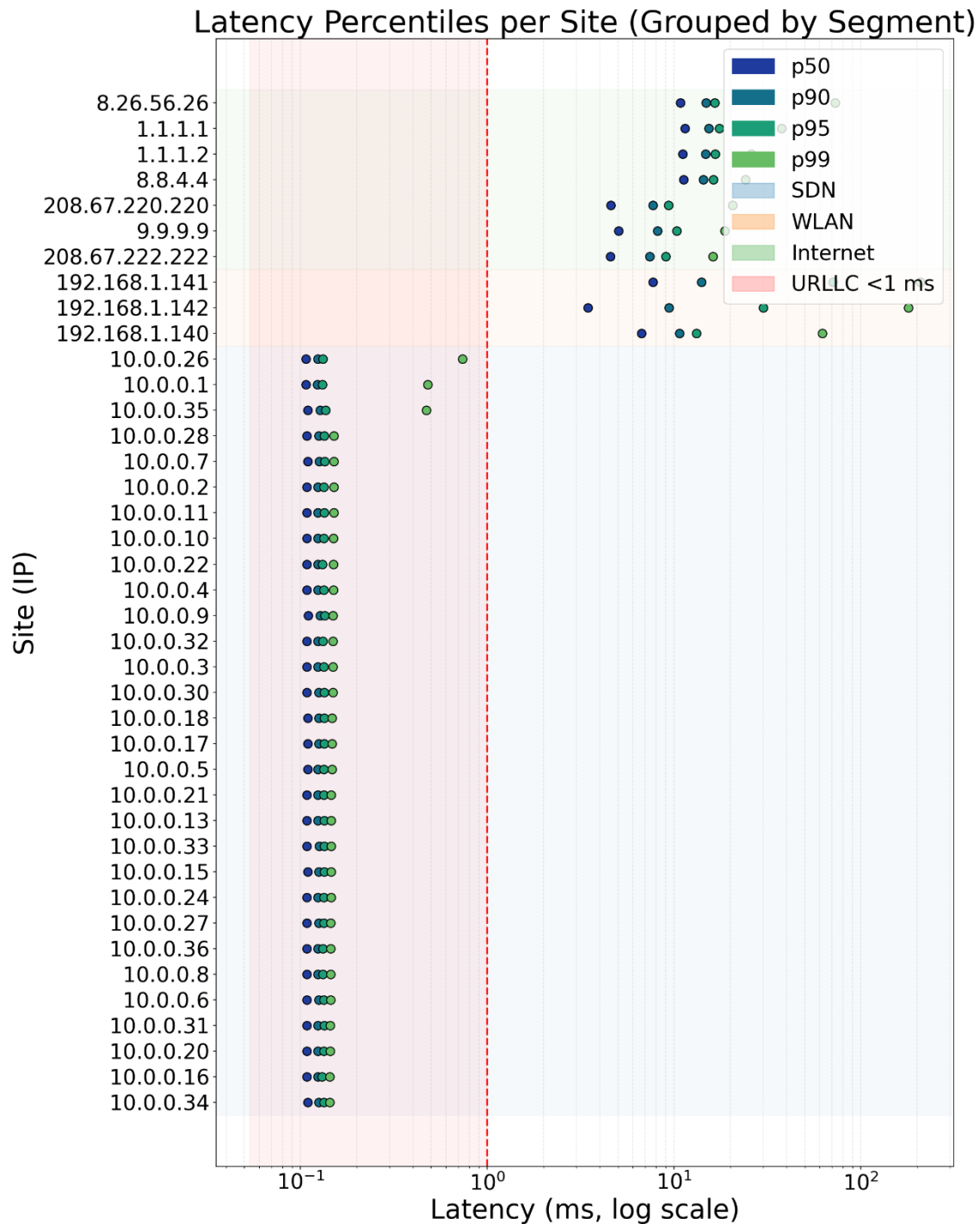


Figure 5.55: Per-Site Latency Distribution (p50-p99) Grouped by Network Segment.

3) Latency Distribution Analysis: Probability Density and Skewness: Figure 5.56 complements the percentile views by displaying the full probability density (violin plots) for each site, effectively visualising the distributional shape and skewness that summary statistics often obscure. The SDN sites (bottom) appear as highly compact masses concentrated below 0.2 ms. Crucially, while the “body” of the violin is entirely within the URLLC safety zone, the thin horizontal lines extending toward 10 ms represent the rare outliers identified in Figure 5.54, confirming that the tail exists but carries negligible probability mass. The WLAN sites (middle) reveal a highly skewed, “comet-like” morphology: while the density peaks near 5 ms, the distribution smears extensively to the right, indicating that latency is not just high but fundamentally unstable. The Internet sites (top) display a broader, more symmetrical distribution centred around 20-40 ms, indicating a high-latency baseline with less extreme skew than the local wireless connections.

For the Transformer-based policy, these shapes define the probabilistic reward landscape. The RL agent learns that while SDN has non-zero tail (the thin whiskers), the integrated probability of a violation is statistically insignificant, allowing it to treat the link as deterministic. Conversely, the elongated WLAN shapes reveal that “danger” is not an edge case but a structural feature of the distribution; a significant portion of the probability density falls into the penalty region (>10 ms). This shape drives the scheduler to view WLAN not just as “slower” but as “untrustworthy”, leading to the exclusionary behaviour observed in the final policy. The Internet distributions, while violating the URLLC target, offer a bounded-variance profile that the agent can exploit for non-critical background traffic, where the mean latency matters more than maximum jitter.



Figure 5.56: Full Latency Probability Densities (Violin Plots) per Site: Visualising Skewness and Tail Risks.

4) Temporal Dynamics: Rolling p99 Latency and Session Stability: Figure 5.57 shifts the analysis from static distributions to the temporal domain, tracking the rolling 99th percentile (p99) latency over a 35-minute session to evaluate sustained reliability. The SDN segment (blue) reveals remarkable temporal consistency: after an initial stabilisation period, the p99 settles firmly below the 1 ms URLLC threshold, fluctuating only within the safe zone (0.2-0.9 ms) without a single breach. This confirms that the low latency observed in previous Figures (5.54, 5.55, and 5.56) is not an artefact of aggregation but a continuous operational state. On the contrary, the WLAN (orange) and Internet (green) segments operate nearly two orders of magnitude higher, hovering between 10 ms and 200 ms. These two segments show significant burstiness, with the WLAN specifically exhibiting prolonged spikes that would result in consecutive packet deadlines being missed in a real-time control loop.

From a scheduling perspective, this temporal stability is the critical differentiator. The Transformer-based RL agent relies on this sequential data to build trust; the flat, bounded trajectory of the SDN line reinforces the policy that exclusively maps URLLC flows to this slice. The graph also highlights why “reactive” switching is insufficient for the

external segments: because the WLAN and Internet latencies are consistently distinct from the URLLC target (never dipping below 1 ms), the agent does not need to “hunt” for momentary dips in traffic. Instead, the optimal time-aware policy is structurally static for critical flows – permanently anchoring them to SDN – while treating WLAN and Internet purely as carriers for delay-tolerant background traffic or as failover paths where graceful degradation is acceptable.

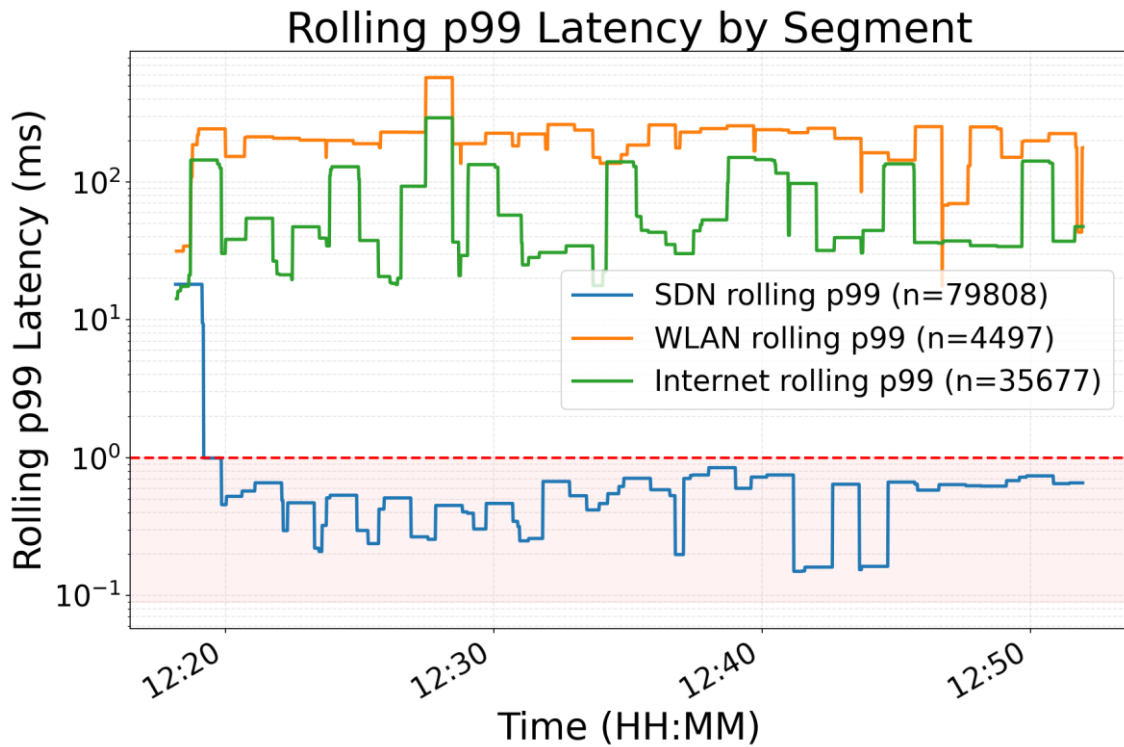


Figure 5.57: Temporal Evolution of Rolling 99th Percentile (p99) Latency across Network Segments.

Table 5.25 quantifies the statistical significance of this experiment’s observations, presenting the 95% confidence intervals and p-values that confirm the distinct performance profiles of the SDN, WLAN, and Internet segments.

Table 5.25: Statistical Summary of Latency Performance per Network Segment.

Network	95% CI range (per site)	Typical p-value	Note
SDN (10.0.0.0/24)	0.14-0.17 ms (e.g., site_22: 0.15-0.15 ms)	≈ 0.0000	Ultra-tight CIs: URLLC-grade, effectively zero loss.
WLAN (192.168.1.0/24)	9.95-15.69 ms (e.g., sites_1-3)	≈ 0.0000	Low median, but extremely high-tail variance (unstable for URLLC).
Public DNS / Internet	6.25-14.30 ms (e.g., sites_5, _9, _10, _11)	≈ 0.0000	High baseline latency, but bounded tail (more predictable than WLAN).

As shown in Table 5.25, the statistical analysis reveals distinct latency profiles across the three network segments. For SDN-internal nodes (10.0.0.0/24), per-site 95% confidence intervals are extremely narrow (≈ 0.14 - 0.17 ms; e.g., site_22: 0.15 - 0.15 ms) with typical p-values ≈ 0.0000 . This indicates ultra-stable, URLLC-grade RTTs where the scheduler behaves deterministically.

In contrast, devices on the WLAN (192.168.1.0/24) exhibit higher mean RTTs in the 9.95 - 15.69 ms range. While statistically significant ($p \approx 0.0000$), these values mask the inherent tail instability observed in the wireless medium.

Finally, the public Internet endpoints show similar mean latencies (≈ 6.25 - 14.30 ms) to WLAN but are structurally distinct. The consistent p-values ≈ 0.0000 across all groups confirm that these performance differences are statistically significant and not artefacts of random measurement fluctuation.

For 5G WSN and URLLC scenarios, these results are particularly significant. The SDN segment demonstrates that the Transformer-based RL scheduler can maintain per-site latency well within sub-millisecond to a few-millisecond budgets with extremely tight confidence bounds, which is crucial for time-critical control loops, actuator feedback, and safety-related traffic in industrial WSNs. The clear separation between SDN-internal, WLAN, and public Internet CIs reveals that the learned policy not only minimises mean delay but also mitigates tail latency risks in a statistically rigorous way. This aligns with 5G's emphasis on reliability and bounded delay rather than averages alone. In practice, this supports an architectural principle where URLLC-class flows are kept within SDN-controlled slices, while less critical sensing and supervisory traffic traverses the WLAN or public Internet; the scheduler's statistically significant performance across all three segments indicates that it can enforce such differentiation in a way that is measurable, reproducible, and suitable for 5G WSN deployments.

5.6.5.1 Updated Benchmark Alignment after Tail-Latency and Statistical Validation

Section 5.6.3.4 assessed benchmark alignment using minimum/maximum RTT ranges from earlier iterations of experiment 5 (e.g., 2.5 - 8.5 ms in Table 5.24), which resulted in "partial" compliance for 1 ms URLLC-class targets. The latest iteration of Experiment 5 (5.6.5) refines this conclusion by evaluating compliance using tail-aware and statistically validated latency measures (CCDF/p99 behaviour and 95% confidence intervals), which are more representative of URLLC-style requirements than min/max summaries alone.

Using the latest evidence obtained for Experiment 5, the benchmark interpretation becomes segment-specific rather than "single-number" global:

1) Strict URLLC (≈ 1 ms class): The SDN slice is URLLC-compliant in this testbed (sub-millisecond tail stability and ultra-tight per-site CIs of ≈ 0.14 - 0.17 ms). In contrast, the WLAN and public Internet segments remain structurally non-compliant for 1 ms URLLC due to persistent p99/tail violations and multi-millisecond CI ranges (≈ 9.95 - 15.69 ms WLAN; ≈ 6.25 - 14.30 ms Internet).

2) Less stringent latency classes (e.g., eMBB, or supervisory control): Experiment 5 indicates that WLAN/Internet performance may be suitable for non-URLLC services, but

their distributions and confidence bounds support treating them as delay-tolerant or best-effort carriers rather than deterministic real-time links.

The proposed Transformer-based RL scheduler can support URLLC-grade operation when URLLC flows are constrained to the SDN-controlled slice, while WLAN and public Internet paths should be positioned as non-URLLC segments within the architecture. This re-scoped interpretation is consistent with the statistically significant separation between segment latency profiles reported in Table 5.25 (typical $p \approx 0.0000$ across groups), demonstrating that the observed compliance/non-compliance is systematic rather than measurement noise.

5.6.5.2 Robustness Implications for the Prior-Work Comparison under Scaled, Tail-Aware Evaluations

The prior-work comparison in Section 5.6.3.3 (Table 5.23) already indicates that the proposed Transformer-based RL scheduler achieves competitive baseline latency and reliability in a live network (e.g., 1.25 ms lowest latency, 4.25 ms maximum latency, and 100% scheduling success ratio), compared with closely related simulation-based studies.

The last iteration of experiment 5 strengthens this comparison by demonstrating robustness beyond point estimates, through (i) scale (40 sites, 1,200 epochs), (ii) explicit reliability-aware mechanisms (e.g., site-health gating and tail-latency penalties), and (iii) statistical evidence of stability. Specifically, the SDN segment shows sustained sub-millisecond tail behaviour with the rolling $p_{99} \approx 0.2\text{--}0.9$ ms without breaches over the session, together with ultra-tight per-site 95% CIs $\approx 0.14\text{--}0.17$ ms and typical $p \approx 0.0000$, indicating that the observed low-latency regime is reproducible across sites and not attributable to random fluctuation. This provides a stronger robustness argument than the baseline comparison alone, particularly because many comparable works report summary outcomes in simulation without tail-latency distributions and confidence bounds at a comparable scale.

5.7 Cross-Dataset and Cross-Experiment Comparison and Discussion

The previous sections and subsections presented each dataset and experiment in isolation, focusing on local design choices and per-model performance. This section brings those pieces together. First, it contrasts the five datasets collected on the live IEEE 802.11ac testbed, emphasising how the capture method, feature richness, and temporal scale shape learning tasks. It then summarises the five experiments by learning paradigm and abstraction level, before presenting a unified comparison of the best-performing models across latency, reliability, and classification/QoS-awareness.

5.7.1 Cross-Dataset Comparison

The five datasets analysed in this chapter differ markedly in how traffic was scheduled, how measurements were captured, and how rich the resulting features are. These differences determine both the feasible learning objectives and the achievable performance in the subsequent experiments.

1) Dataset-family overview: The PF and RR datasets were obtained directly from custom Python schedulers and stored as CSV files with engineered QoS metrics (timestamp, delay, jitter, instantaneous and average throughput, inter-arrival times, and packet counters). The two traces differ primarily in the scheduling strategy applied to the same heterogeneous devices: PF enforces proportional fairness, whereas RR cycles deterministically across stations.

In contrast, the PRIO, HTB, and CBQ datasets were captured as Wireshark PCAP traces under Linux tc queuing disciplines and later converted to CSV format. They contain only raw packet-level attributes (timestamps, packet lengths, protocol headers, Info strings, and basic flags), without explicit QoS fields such as delay or jitter.

2) Scheduler layer and feature richness: PF and RR operate at the MAC-scheduler layer and embed additional scheduler metadata such as a binary *is_selected* flag, PF weights, loss rate, and per-round throughput. This yields a dense, structured feature space that directly exposes scheduler decisions and their QoS consequences.

PRIO, HTB, and CBQ act at the kernel’s queuing discipline layer and do not expose scheduler state explicitly; they rely entirely on observable packet characteristics and protocol mixes. As a result:

- a) PF/RR are feature-rich and well-suited to regression and time-series modelling of QoS.
- b) PRIO, HTB, and CBQ provide raw traffic fingerprints that are primarily useful for protocol and scheduler classification.

3) Learning objectives

These feature differences naturally lead to distinct learning objectives across experiments.

- a) PF and RR, with continuous QoS variables (delay, jitter, throughput, and loss rate), support supervised regression and sequence-prediction tasks, such as forecasting delay from instantaneous load and scheduler context.
- b) PRIO, HTB, and CBQ are used to learn discriminative patterns in packet-size distributions, inter-arrival statistics, and protocol usage, enabling classifiers to infer either the underlying queuing discipline or the dominant protocol classes from raw traffic alone.

4) Temporal scale and sample size: The PF and RR traces each span roughly two hours and contain about 55,000 device-round records, providing long temporal horizons and regularly spaced observations that are amenable to per-round regression and time-series models.

The PRIO capture lasts about 14 minutes, with 134,309 packets; HTB runs for roughly 27 minutes, with 488,673 packets; and CBQ extends to about 40 minutes, with 301,629

packets. These packet-level datasets are temporally shorter but substantially denser, offering ample samples for robust classification and cross-validation, but with less continuity at the per-flow or per-device level for sequence-based architectures unless additional windowing or aggregation is applied.

5) Modelling implications

Taken together, these contrasts define two complementary dataset families:

a) PF and RR act as structured QoS corpora that enable accurate delay prediction and time-series analysis with classical ML (e.g., Random Forest, EGPR) and sequential DL models (e.g., LSTMs, and Transformers).

b) PRIO, HTB, and CBQ require models to recover scheduler or protocol information indirectly from raw packet attributes, favouring classification-oriented pipelines with tree-based ML and DL (CNNs, LSTMs) that can exploit subtle traffic fingerprints.

Despite being collected on the same physical testbed, the datasets therefore occupy different points in the feature-label design space, motivating the diverse set of experiments presented in this chapter.

6) Unified dataset summary: To make these distinctions explicit, Table 5.26 summarises the core characteristics of each dataset and how they map onto the experiments.

Table 5.26: Unified Dataset Summary.

Dataset	Scheduler discipline	Layer	Sample size	Feature space	Learning task	Experiments using the dataset
PF	Proportional Fair WLAN scheduler	MAC/scheduler	54,984 device-round records	Engineered QoS metrics (delay, jitter, throughput, loss, interarrival) + scheduler metadata (is_selected, PF metric, per-round throughput)	Regression and time-series prediction of delay/QoS	Exp. 1 (PF vs RR), Exp. 4 (delay forecasting with RF, LSTM, hybrid Transformer)
RR	Round-Robin WLAN scheduler	MAC/scheduler	55,216 device-round records	Same QoS and scheduler features as PF, with deterministic round-robin selection pattern	Regression and comparative scheduler analysis	Exp. 1 (PF vs RR regression)

PRIO	Priority Queuing (tc)	Kernel/queuing discipline	134,309 packets	Raw packet-level attributes (timestamp, length, protocol, addresses, Info)	Protocol classification and traffic fingerprinting; input to scheduler-classification DL	Exp. 2 (protocol classification), Exp. 3 (image-based CNN/CNN-LSTM, raw-data LSTM)
HTB	Hierarchical Token Bucket (tc)	Kernel/queuing discipline	488,673 packets	Raw packet-level attributes for PRIO	Protocol classification and scheduler fingerprinting; DL QoS-class classification	Exp. 2 (protocol classification), Exp. 3 (image-based CNN/CNN-LSTM, raw-data LSTM)
CBQ	Class-Based Queuing (tc)	Kernel/queuing discipline	301,629 packets	Raw packet-level attributes for PRIO	Protocol classification and scheduler fingerprinting; DL QoS-class classification	Exp. 2 (protocol classification), Exp. 3 (image-based CNN/CNN-LSTM, raw-data LSTM)

The unified view presented in Table 5.26 highlights how PF and RR provide structured QoS sequences for regression and forecasting, while PRIO, HTB, and CBQ contribute dense packet-level traffic signatures exploited by ML/DL classifiers in subsequent experiments.

5.7.2 Cross-Experiment Comparison

The five experiments form a coherent progression from offline QoS characterisation to online QoS optimisation. They span three learning paradigms - supervised regression, classification, and RL – and operate at different abstraction levels, ranging from packet-level fingerprinting to scheduler-level delay control.

1) Experiment 1 (Baseline ML regression for PF vs RR): Experiment 1 combines descriptive statistics with EGPR to compare PF and RR schedulers using structured QoS datasets. The focus is on supervised regression: predicting delay (and related QoS metrics) from engineered features such as throughput, jitter, loss rate, and scheduler flags, thus quantifying how each scheduler balances delay, throughput, and fairness under identical traffic conditions.

2) Experiment 2 (Protocol-level classification on PRIO, HTB, and CBQ): Experiment 2 shifts to classification on raw packet data captured under PRIO, HTB, and CBQ queuing disciplines. Classical ML models (Optimisable Decision Trees, SVMs, and Fine Trees) are trained to distinguish network protocols and traffic classes solely from packet-level statistics and derived flags, without explicit QoS variables. This experiment shows that

different schedulers leave recognisable statistical “fingerprints” in protocol distributions and packet attributes that ML models can exploit.

3) Experiment 3 (DL for scheduler and QoS-class identification): Experiment 3 extends the classification perspective using DL. First, CNN and CNN-LSTM models operate on image-based representations of PRIO, HTB, and CBQ traffic to determine whether visual encoding of packet statistics is sufficient to separate queuing disciplines. Second, a pure LSTM model directly ingests raw packet-level features from the same dataset to predict scheduling classes. This experiment contrasts image-based and sequence-based DL pipelines on identical traces and highlights the benefits of temporal modelling on raw features.

4) Experiment 4 (Time-series delay prediction with DL vs Random Forest): Experiment 4 revisits delay prediction on the PF dataset, explicitly framing it as a time-series forecasting problem. Advanced sequential DL models (a hybrid Linear + LSTM + Transformer architecture and a pure LSTM) are compared against a strong non-sequential baseline (Random Forest). All models are trained on the same QoS feature set, with DL models consuming short sequences of past measurements and the Random Forest operating on instantaneous features. This experiment isolates the benefit of explicit temporal modelling for latency forecasting using the same dataset as Experiment 1.

5) Experiment 5 (Transformer-based RL for live scheduling): Experiment 5 moves from supervised learning to RL by deploying a Transformer-based RL agent that directly controls scheduling decisions in the live network. The agent observes aggregate latency and throughput metrics across multiple sites and learns a policy that maximises a reward function dominated by low RTT and high throughput, achieving millisecond-scale delays and a 100% ping success rate. Unlike Experiments 1-4, which infer or predict QoS from given traces, Experiment 5 closes the loop by actively shaping network behaviour in real time.

Together, these experiments form a layered pipeline: offline regression and forecasting (Experiment 1 and 4), protocol and scheduler identification from traffic signatures (Experiment 2 and 3), and online policy-driven scheduling with RL (Experiment 5).

5.7.3 Unified Performance Comparison across Experiments

This subsection consolidates the key quantitative results from Experiments 1-5 into a unified view, focusing on three common axes: latency prediction, reliability, and classification/QoS-awareness. Only the best-performing model from each experiment is considered, allowing different learning paradigms to be compared on a common set of metrics.

1) Latency and delay prediction

Two experiments deal directly with delay prediction:

a) In Experiment 1, the EGPR model on the PF dataset achieves a validation RMSE of approximately 3.20 ms, MAE around 0.38 ms, and R^2 of about 0.97, providing a strong baseline for one-step-ahead delay prediction.

b) In Experiment 4, a well-tuned Random Forest substantially outperforms the deep sequence models, reaching validation RMSE ≈ 0.62 ms, MAE ≈ 0.15 ms, and $R^2 \approx 0.998$ on unseen devices, clearly surpassing both the hybrid Linear + LSTM + Transformer model (RMSE ≈ 4.33 ms, MAE ≈ 1.07 ms, $R^2 \approx 0.91$) and the pure LSTM (RMSE ≈ 2.36 ms, MAE ≈ 0.59 ms, $R^2 \approx 0.97$).

From a unified perspective, these results indicate that, for one-step-ahead delay prediction in this WLAN setting, classical models (EGPR, Random Forest) currently provide the tightest error bounds, with the Random Forest achieving sub-millisecond average absolute error for new devices.

The RL scheduler in Experiment 5 brings a complementary view: instead of forecasting delay, it actively controls it. The Transformer-based RL agent maintains one-way latency in the approximate range 1.25-4.25 ms in the live network while keeping training loss near zero and maximising cumulative reward over hundreds of epochs. Delay-prediction regressor thus characterises how the WLAN behaves under a given scheduler (PF/RR), whereas the RL agent demonstrates how far latency can be reduced when scheduling decisions are optimised online.

2) Reliability

Reliability appears in different guises across the experiments, but can be summarised on a single axis:

a) In Experiment 1, packet-delivery ratios for the four devices under PF and RR are consistently high (around 99.6-100%), even when average delay varies between stations.

b) In Experiment 5, the RL scheduler achieves a scheduling success ratio of 100%, with no ping timeouts observed during the final runs, while still operating in the low-millisecond latency regime.

Overall, the experiments operate in a high-reliability regime. The RL agent is distinctive in that it explicitly enforces reliability (via success ratio) while simultaneously optimising latency and reward. A unified reliability axis combining delivery ratio (Experiments 1 and 4) and scheduling success ratio (Experiment 5) shows that none of the proposed models achieve latency improvements at the expense of reliability in the tested scenarios.

3) Classification and QoS-awareness

Experiments 2 and 3 contribute to the classification and QoS-awareness perspective:

a) In Experiment 2, protocol classification on PRIO, HTB, and CBQ demonstrates that tree-based and SVM models achieve very high per-class TPRs (often 95-100% for ARP, SSDP, TCP, and TLSv1.3) and AUC values close to 1.0 for several protocols, while some low-volume or more heterogeneous classes (e.g., HTTP, NTP, OCSP, TLSv1) remain harder to distinguish.

b) In Experiment 3, the LSTM on raw feature emerges as the most effective scheduler classifier, achieving around 99.5% validation accuracy with AUC=1.0 and average precision = 1.0 across CBQ, HTB and PRIO, and clearly outperforming the CNN and CNN-LSTM architectures based on image representations (which reach at most 87.5% and 50% validation accuracy, respectively).

Summarised on a unified classification axis, three points stand out:

- Raw-feature models dominate image-based ones once sufficient tabular data is available, as evidenced by the LSTM’s performance on PRIO, HTB, and CBQ.
- Per-class metrics (TPR, AUC, AP) reveal which protocols and scheduler classes are intrinsically easier or harder to separate and provide a natural bridge to the RL scenario, where those traffic types could be prioritised based on predicted class.
- Experiments 2 and 3 effectively provide the QoS-awareness “front end” (detecting protocol/scheduler patterns), while Experiments 1, 4, and 5 focus on predicting or controlling delay given that awareness.

To consolidate these findings, Table 5.27 summarises, for each experiment, the dataset(s) used, the primary task, the best-performing model, and the associated key metrics. This unified view makes it clear that different algorithm families occupy distinct niches in the overall QoS pipeline, with classical regressors excelling at delay prediction, deep sequence models at QoS-aware classification, and the Transformer-based RL agent at online scheduling and QoS optimisation.

Table 5.27: Unified Summary of Best-Performing Models and Key Metrics Across All Experiments.

Experiment	Dataset(s)/Scenario	Primary task	Best-performing model	Key metrics (validation/live)
1	PF (and RR) WLAN traces	Delay prediction (regression, PF focus)	EGPR	RMSE $\approx$ 3.20 ms; MAE $\approx$ 0.38 ms; $R^2 \approx$ 0.96 (PF dataset)
2	PRIO/HTB/CBQ packet captures	Protocol-level QoS classification	Optimisable Tree / SVM (per protocol)	TPR $\approx$ 95-100% for major protocols (ARP, SSDP, TCP, TLSv1.3); AUC up to $\approx$ 1.00 for some classes
3	PRIO/HTB/CBQ (image-based + raw-feature dataset)	Scheduler / QoS-class identification	LSTM on raw traffic features	Accuracy $\approx$ 99.5%; AUC = 1.00; AP = 1.00 across CBQ, HTB, and PRIO classes
4	PF WLAN traces (including unseen devices)	Time-series delay prediction (regression)	Random Forest	RMSE $\approx$ 0.62 ms; MAE $\approx$ 0.15 ms; $R^2 \approx$ 0.998 (unseen devices)

5	Live heterogeneous traffic (ping + background flows)	Online scheduling and QoS optimisation (RL)	Transformer-based RL scheduler (policy)	One-way latency $\approx$ 1.25-4.25 ms; training loss $\rightarrow$ 0; cumulative reward $\approx 5 \times 10^5$; success ratio 100%

Figure 5.58 summarises cross-experiment latency and prediction error. The left panel shows one-way operational latency ranges for PF and RR (Experiment 1), derived from the delay statistics reported in Tables 5.1, 5.2, and 5.7 together with the RL scheduler’s 1.25-4.25 ms range (Experiment 5) from Section 5.6. The right panel compares validation RMSE and MAE for the regression models evaluated in Sections 5.2 and 5.5 (EGPR, hybrid Linear + LSTM + Transformer, LSTM, and Random Forest), highlighting that Random Forest yields the lowest errors, whereas the hybrid model has the highest.

Cross-Experiment Latency Summary

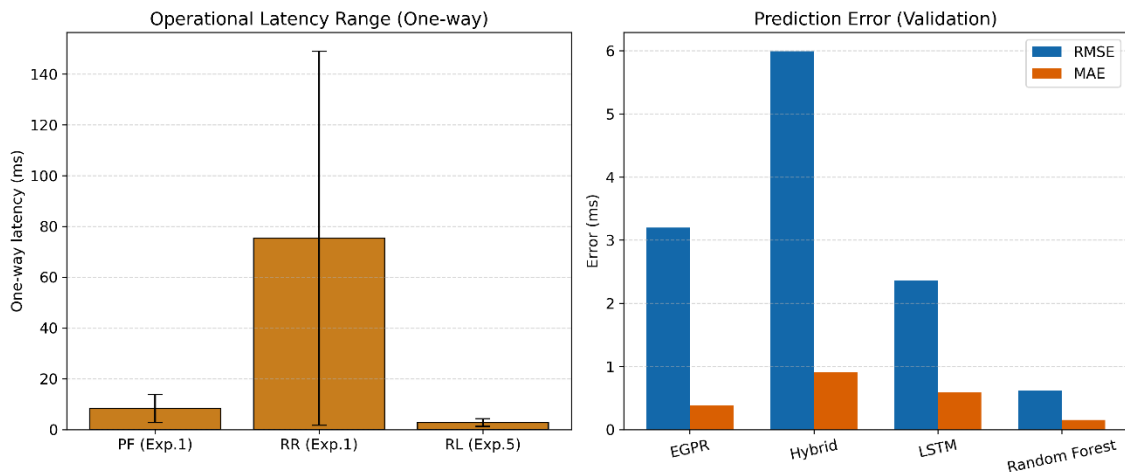


Figure 5.58: Cross-experiment latency and prediction error. (a) One-way operational latency under PF and RR (Experiment 1) and the Transformer-based RL scheduler (Experiment 5); bars indicate the mean latency value and whiskers denote the observed range (min-max). (b) Validation prediction error (RMSE and MAE, ms) for EGPR, hybrid Linear + LSTM + Transformer, LSTM, and Random Forest delay prediction models.

Figure 5.59 summarises the unified performance metrics in Section 5.7.3 by plotting the best (normalised) scores achieved across experiments. Each axis corresponds to a key metric – classification AUC, regression R^2 , minimum one-way latency, reliability, and validation RMSE – scaled $[0, 1]$, with the polygon tracing the strongest model on each axis (LSTM classifier for AUC, Random Forest for R^2 / RMSE, and the Transformer-based RL scheduler for latency and reliability).

In Figure 5.59, all axes are drawn so that a larger value always means better performance. For metrics where “higher is better” (classification AUC, regression R^2 , and reliability),

the best results already point outward – for example, $AUC=1.0$, $R^2 \approx 0.998$, and reliability =100%. For metrics where “lower is better” (latency and RMSE), we reverse the direction before normalising so that the lowest observed values also point outward – here, the minimum one-way latency of 1,25 ms (RL scheduler) and the lowest validation RMSE of 0.617 ms (Random Forest) appear as high scores on their respective axes.

Best Score per Metric and Model (Normalized to 0-1, Real Values in Legend)

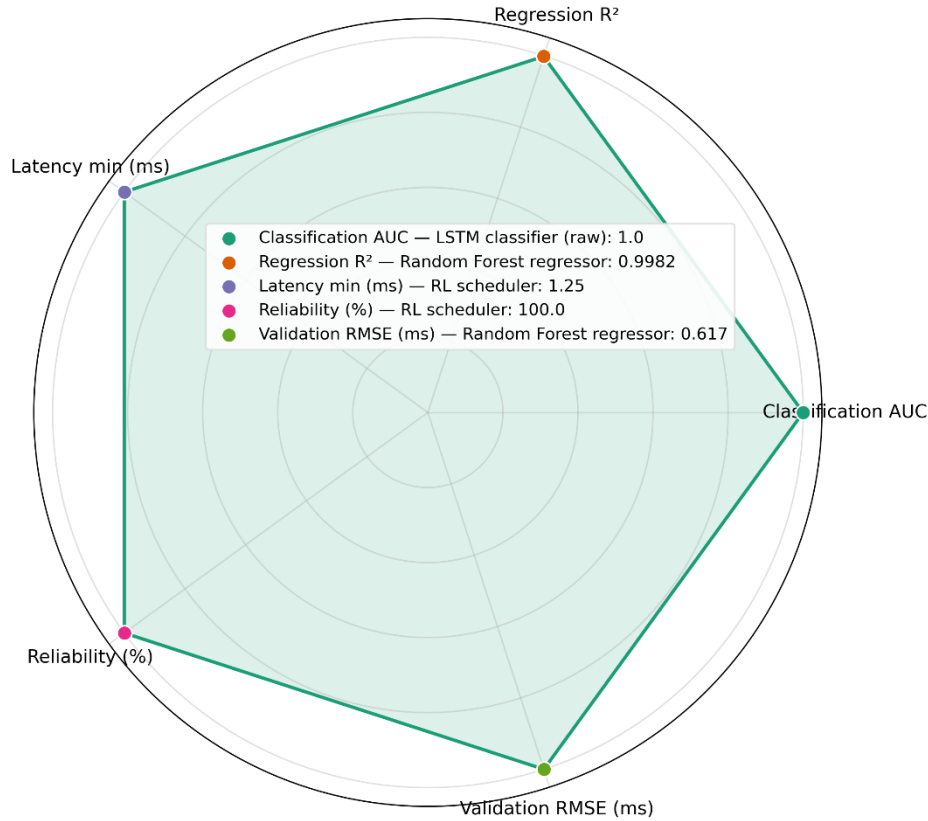


Figure 5.59: Envelope of best performance across experiments. Radar plot of normalised scores (0-1) where larger values indicate better performance, formed by the strongest result on each metric across experiments: classification AUC (LSTM classifier), regression R^2 and validation RMSE (Random Forest regressor), and minimum one-way latency and reliability (Transformer-based RL scheduler). The polygon, therefore, represents an envelope of the best results from different models rather than a single model profile.

Figure 5.60 compares reliability across the scheduling approaches using per-device delivery ratios for PF and RR (Experiment 1) and per-device success ratios for the Transformer-based RL scheduler (Experiment 5). The results show consistently high reliability across all devices, with the RL scheduler achieving a uniform 100% success ratio while PF and RR remain close to 100% with only marginal variations between stations.

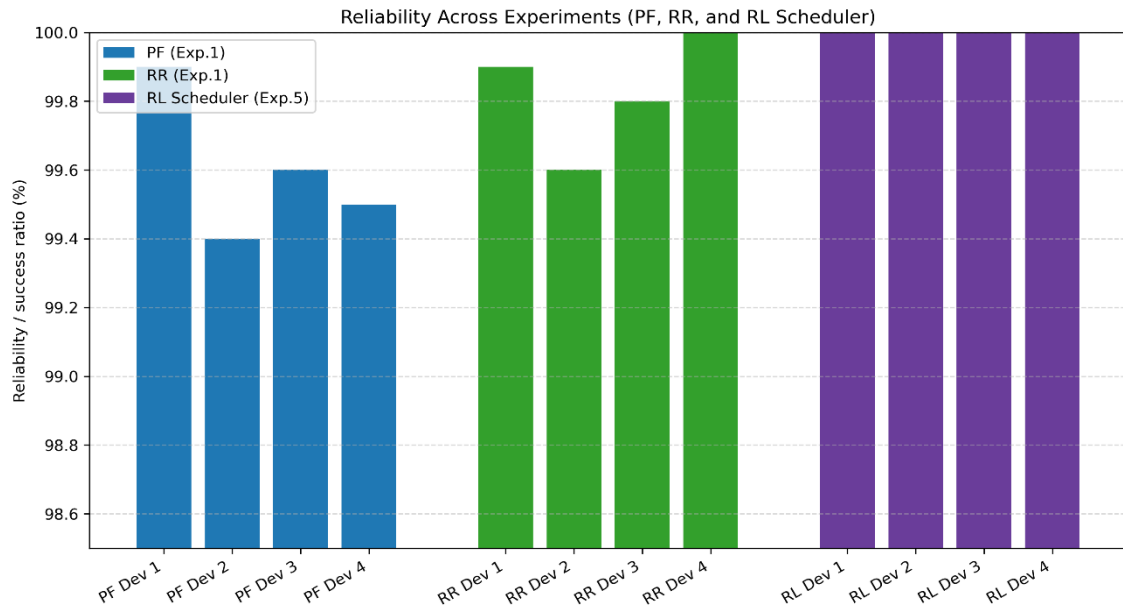


Figure 5.60: Reliability comparison across scheduling approaches. Per-device delivery ratios for PF and RR (Experiment 1) are shown alongside per-device success ratios for the Transformer-based RL scheduler (Experiment 5).

The consistently near-perfect reliability across PF, RR, and the RL scheduler indicates that the observed differences in latency and prediction error across experiments are not driven by packet delivery failures; instead, they reflect how each approach manages delay while maintaining a high success rate.

Chapter 6: Synthesis and Implications of Findings

6.1 Introduction

This chapter presents a synthesis of the findings, key contributions, answers to the research questions, and the broader implications and limitations of all experiments that constitute the body of this research study. The experiments involved a comparative evaluation of scheduling algorithms using ML regression techniques; a comparison of scheduling algorithms using both ML and DL classification models; prediction of network performance through regression-based models; and the implementation of a Transformer-based RL agent for network traffic scheduling. Together, these experiments provide a comprehensive foundation for assessing the effectiveness of intelligent scheduling strategies in real-world network environments.

The analyses and discussions in this Chapter are grounded in the experimental results presented in *Chapter 5*, which evaluated the performance of the models and systems implemented in *Chapter 4*. By synthesising these findings, this chapter highlights key insights, draws broader implications for QoS-aware scheduling and prediction in 5G WSNs, and addresses the core research questions outlined earlier in the study.

6.2 Findings and Implications: Comparative Evaluation of PF and RR Scheduling with EGPR

This section presents the conclusions and future research directions arising from the experimental implementation and comparative analysis of the Proportional Fair (PF) and round robin (RR) scheduling disciplines conducted in a live network environment. Additionally, it presents insights from the EGPR model, which was used to evaluate and predict the performance of both scheduling disciplines using datasets collected during experimentation.

6.2.1 Brief Context

The experimental setup consisted of five PCs: one dedicated to executing the scheduling algorithms, four Raspberry Pi devices serving as traffic endpoints, and a router managing network traffic. The experiment was conducted in a live network environment with additional devices operating concurrently in the background, which could potentially affect traffic conditions. The scheduling schemes were implemented using Python scripts, and network data were continuously collected over two hours. The primary goal was to obtain statistical QoS metrics to compare the performance of the PF and RR scheduling disciplines. This comparison was performed statistically, based on direct network performance measurement, and analytically, through inference using an EGPR model.

6.2.2 Summary of Key Findings (PF vs RR)

1) Performance Highlights: The Round Robin (RR) scheduler exhibited consistently high packet transmission reliability, achieving nearly 100% packet delivery for all devices, with Device 4 demonstrating perfect transmission (100%). While RR generally

achieved higher throughput per device than Proportional Fair (PF), significant latency variability was observed, notably for Device 3, which experienced an average delay of approximately 65.40 ms and peak delays up to 297.94 ms, considerably higher than under PF scheduling. In contrast, the PF scheme showed lower, more balanced delays across devices, suggesting superior overall fairness in latency management. Consequently, while RR achieves better throughput and packet reliability, the PF scheduler delivers a more consistent and equitable QoS, particularly in scenarios sensitive to latency variations.

2) EGPR Model Performance: Table 6.1 is a comparative analysis summarising EGPR predictive modelling results for the Proportional Fair (PF) and Round Robin (RR) scheduling schemes.

Table 6.1: EGPR Model Performance Comparison.

Metric	Proportional Fair (PF)	Round Robin (RR)
RMSE (Validation)	3.2002 ms	5.9915 ms
MSE (Validation)	10.241 ms ²	35.899 ms ²
R-squared (Validation)	0.96457	0.93651
MAE (Validation)	0.3849 ms	0.9073 ms

a) Overall Model Fit: The EGPR model demonstrated strong predictive performance for both scheduling disciplines. However, it performed notably better for the PF scheme, as indicated by lower RMSE (3.2002 ms), lower MAE (0.3849), and a higher R-squared (0.96457). This suggests that delays under PF scheduling were more predictable and less variable.

b) Comparative Predictive Insights from Scatter Plots

- **Instantaneous vs Average Throughput:** In both scheduling disciplines, instantaneous throughput consistently showed a stronger inverse correlation with delay than average throughput. This underscores the greater predictive value of instantaneous throughput for accurately modelling network delays.
- **Jitter Contribution:** In both algorithms, jitter maintained a strong positive correlation with delay, confirming its importance as a critical predictor in modelling network latency.
- **Device-Specific Analysis:** Under both PF and RR, Device 3 revealed significantly higher delay variability. However, this variability was notably greater in RR scheduling, resulting in higher prediction errors. The EGPR model effectively captured the overall trend, though it struggled slightly more to accurately predict delays for Device 3 in the RR scheme.

In summary, the EGPR model predictions indicate that the PF scheduling scheme provides more stable, predictable QoS, particularly in delay management. While RR scheduling achieved higher throughput than PF, it introduced more significant latency fluctuations, reducing predictability. The PF scheme demonstrates superior consistency

and model accuracy in QoS predictive modelling, highlighting its potential advantage for delay-sensitive applications.

3) Significance and context: The outcomes of the experiment directly address the research problem by demonstrating a quantitative methodology for comparing traditional scheduling schemes – PF and RR – using key performance indicators (KPIs) derived from real-time network data, such as delay, throughput, jitter, and packet reliability. The statistical analysis provided clear distinctions in performance, highlighting PF’s superior consistency in delay management, RR’s higher throughput, and greater latency variability. Furthermore, applying the EGPR model successfully addressed the research question by revealing underlying relationships between network performance metrics. The EGPR results demonstrated that instantaneous throughput and jitter are significant predictors of network delay, enabling accurate prediction and a deeper understanding of network behaviour under each scheduling scheme. Consequently, the model identified that the PF scheduling strategy offers better QoS predictability and stability under dynamic, real-time network conditions, effectively addressing the research problem and question.

6.2.3 Contributions to the Field

1) Quantitative Empirical Comparison in a Live Network: Provided a robust quantitative comparison between Proportional Fair (PF) and Round Robin (RR) scheduling algorithms using key QoS metrics (delay, throughput, jitter, and packet reliability) derived directly from a live, dynamic network environment. This addresses practical QoS management challenges and helps bridge the gap between theoretical models and real-world network dynamics.

2) Advanced ML Models for Scheduling Performance Analysis and Predictability Assessment: This study demonstrated the novel application and effectiveness of EGPR not only for accurately predicting network latency but, more significantly, for quantitatively comparing the underlying stability and predictability of different scheduling algorithms. This analysis revealed PF’s superior latency predictability over RR under the tested conditions, as evidenced by significantly better model fit metrics (R^2 , RMSE, and MAE).

3) Establishing QoS Predictability as a Key Differentiator: Our experimental study elevated the concept of QoS predictability (specifically, latency stability) beyond traditional average metrics, demonstrating through EGPR modelling that it serves as a critical and quantifiable differentiator between scheduling algorithms. This highlights the importance of considering predictability when selecting schedulers, especially for latency-sensitive applications.

4) Identification and Validation of Critical Latency Predictors: This experimental study empirically identified and validated specific network performance indicators – notably instantaneous throughput (demonstrating a stronger correlation with delay than average throughput) and jitter – as critical predictors of network latency under both PF and RR scheduling, contributing valuable insights for network behaviour modelling.

5) Hybrid Methodological Framework for QoS Evaluation: This research study established and validated a replicable methodological framework that synergistically combines direct statistical analysis of empirical network measurements with advanced ML modelling (EGPR). This hybrid approach provides a practical framework for future QoS analysis of network scheduling disciplines.

In summary, this experimental study contributes to the body of knowledge by:

- Providing real-world empirical data on PF vs RR performance.
- Highlighting EGPR as an effective tool for analysing and comparing scheduler predictability.
- Establishing predictability itself as a quantifiable and important QoS dimension alongside traditional metrics.
- Identifying strong predictive indicators for latency using this advanced modelling technique.
- Validating a combined statistical and machine-learning approach for evaluating network scheduling.

Collectively, these contributions enhance understanding of network scheduling and performance prediction, support informed decision-making for network management, and open pathways for future interdisciplinary research between networking and applied ML.

6.2.4 Answer to Research Questions

In this subsection, we present the answers to the research problem and the research question for Experiment 1. Table 6.2 provides a framework for comparing scheduling schemes using statistical methods and ML techniques.

1) Research Problem: *How can traditional scheduling algorithms such as Proportional Fair (PF) and Round Robin (RR) be quantitatively compared using key performance indicators extracted from live network data?*

Answer: A rigorous, reproducible comparison can be achieved through the following four-step workflow, which the experiment implemented end-to-end:

Table 6.2: Framework for Comparing Scheduling Algorithms Using Statistical Analysis and ML Models.

Step	What was done	Why it matters
(a) Live Data Capture	Traffic from four Raspberry Pi endpoints and background devices was steered through PF and RR schedulers on a dedicated PC. Continuous packet-level logs were collected for 2 hours.	It guarantees that KPIs reflect real-world cross-traffic, not a synthetic testbed, enabling conclusions to be generalised to operational networks.
(b) KPI Extraction	The experiment collected the delay, jitter, instantaneous and	These KPIs encompass reliability, capacity (throughput), and user

	average throughput, and packet delivery ratio for each device.	experience (delay and jitter), vital dimensions that operators optimise.
(c) Descriptive Statistics	Mean, variance, and percentile profiles were computed for each KPI and device per scheduler.	Yields direct, interpretable evidence: RR delivered 100% packets but revealed a 65 ms average delay (Device 3) versus 12 ms under PF.
(d) Comparative Analytics	The same KPI vectors were fed into an EGPR model, which was trained and validated separately for PF and RR. Model-fit metrics (RMSE, MAE, and R^2) quantified predictability for each scheduler.	It provides an additional, model-based yardstick that captures subtle temporal interactions which simple averages can miss.

2) Scientific Question: *Can regression techniques such as EGPR reveal underlying relationships between network performance metrics and help identify which scheduling strategy offers better QoS under dynamic real-time conditions?*

Answer: Yes – EGPR exposed the latent KPI relationships and singled out PF as the more QoS-suitable scheduler for the tested conditions. Key observations:

a) Model evidence of stability

Table 6.3: EGPR Model Evidence of Stability for the PF Scheduling Discipline.

Metric (validation set)	PF	RR	Interpretation
RMSE	3.20 ms	5.99 ms	Lower error -> PF delays fluctuate less and are easier to learn.
MAE	0.385 ms	0.907 ms	Confirms tighter error bounds with PF
R^2	0.965	0.937	EGPR explains ~96% of PF variance but only 94% of RR

As demonstrated in Table 6.3, the consistently smaller errors and higher R^2 under PF indicate that its latency process is smoother and more regular, a defining trait of **predictable QoS**.

3) KPI relationship uncovered by EGPR

- **Instantaneous throughput vs. delay:** A strong inverse correlation (steeper under both PF and RR than the average throughput link) reveals that short-term capacity spikes, rather than long-term averages, dictate queuing delay.
- **Jitter vs. delay:** A positive, nearly linear relationship indicates that jitter is a first-order predictor of latency variations, reinforcing its diagnostic value.
- **Device heterogeneity:** Device 3 emerged as the stress-test node; EGPR residuals were largest there, especially for RR, proving the model can localise where a scheduler breaks down.

4) Practical Inference: Because EGPR captures the joint distribution of KPIs, operators can feed live throughput and jitter measurements into the PF-trained model and obtain reliable delay forecasts; doing so with the RR-trained model yields wider confidence intervals. PF delivers fairer latencies and enables tighter SLA guarantees thanks to its predictability.

5) Overall Contribution Relative to Research Aims: By coupling direct KPI statistics with EGPR analysis, the experiment:

- Our work validated a repeatable hybrid framework for scheduler benchmarking.
- Elevated predictability (low-variance, model-friendly latency) to a first-class QoS goal beyond the classic “higher is better” throughput comparisons.
- Demonstrated that machine-learning regression models, when fed the right live-network KPIs, become powerful lenses for deciding between legacy scheduling disciplines without expensive over-the-air trials.

The insights obtained from this experimental work address the two posed questions and provide a methodological template for future work on ML-assisted network optimisation.

6.2.5 Limitations and Future Directions

1) Scale and Network Complexity: The experimental setup involved a relatively small number of devices (four Raspberry Pi endpoints, a scheduling PC and a router), which limits generalisation to larger, more complex, and heterogeneous network environments.

2) Duration of Experiment: The two-hour experiment window might not fully capture long-term variations or peak-time network behaviours, limiting insights into the scheduling algorithm’s performance under prolonged or varying traffic conditions.

3) Device and Hardware Constraints: Device-specific hardware limitations or variations (might be the case of Device 3’s consistently poor performance) may have skewed QoS results independently of the scheduling algorithms, complicating clear interpretations.

4) Predictive Model Generalisation: Although the EGPR model demonstrated strong predictive accuracy, its performance was better under stable QoS conditions (PF scheduling). Its predictive effectiveness may decrease under more chaotic or highly variable conditions, as observed with RR scheduling, particularly when predicting outliers or extreme delay values.

5) Limited QoS Metrics: The study primarily focused on delay, throughput, jitter, and packet loss. Other critical QoS parameters, such as bandwidth utilisation, queue length, and congestion levels, as well as energy consumption, power efficiency, CPU and memory usage, were not explored, which limited the depth of the performance evaluation. Furthermore, addressing these limitations in future research could further enhance the reliability, scalability, and applicability of the findings to broader network contexts.

6) Future Work Suggestions based on this Experiment

a) Exploration of Additional ML Models

- Investigation of different regression techniques, such as DNNs or Gradient Boosting algorithms.
- Conduct a comparative analysis of EGPR with emerging regression approaches, such as Deep Gaussian Processes and Transformer-based regression models.

b) Expansion of Experimental Scenarios

- Extend the experiment to evaluate additional scheduling disciplines, such as Weighted Fair Queuing (WFQ), Stochastic Fair Queuing (SFQ), and Earliest Deadline First (EDF).
- Incorporate larger and more diverse datasets from varying network conditions, including mobility scenarios.

c) Refinement of the EGPR Model

- The EGPR model can be refined to enhance its predictive capabilities by integrating additional predictors and QoS metrics, such as CPU and memory usage, power efficiency, and bandwidth utilisation.
- Optimisation of hyperparameters or exploration of kernel functions better suited to handling highly variable network conditions, especially under the RR scheduling scenario.
- Another interesting direction could be the development of adaptive EGPR models that dynamically update or retrain as network conditions evolve, thereby increasing their real-time applicability.

By pursuing these directions, future research could deepen and broaden insights into scheduling schemes and their implications for network performance and QoS management.

6.3 Findings and Implications: Classification-based Comparison of Scheduling Algorithms Using Operational Network Traffic

6.3.1 Brief Context

This investigative study designed and implemented an experimental setup within a live network environment. Various scheduling disciplines, including Priority Queuing (PRIO), Hierarchical Token Bucket (HTB), and Class-Based Queuing (CBQ), were configured on network interfaces. The primary objective was to collect empirical network traffic data to generate comprehensive datasets. These datasets served as inputs for ML

models (Optimizable Tree, Fine Tree, and Fine Gaussian SVM), enabling detailed traffic analysis and facilitating comparisons of scheduling performance through classification and predictive methods.

6.3.2 Summary of Key Findings

The experimental results for the HTB, PRIO, and CBQ datasets show both consistent trends and dataset-specific differences in model performance. Regarding the HTB dataset, the Optimizable Tree consistently outperforms the Fine Gaussian SVM across protocols such as DNS, ICMP, MDNS, NTP, and UDP. Notably, while both models achieve perfect or near-perfect TPRs for ARP, SSDP, and specific TLS protocols, the Fine Gaussian SVM's low detection rates for several classes highlight its sensitivity to the underlying data distribution or parameterisation challenges.

In the PRIO dataset, the comparative analysis further underscores these observations: the Optimizable Tree excels at protocols such as DNS, ICMP, MDNS, QUIC, and UDP, whereas the Fine Gaussian SVM performs better at detecting TCP, TLSv1.2, and TLSv1.3 traffic. This divergence underscores that model efficacy can be protocol-dependent, with Tree-based methods potentially offering more robust detection capabilities for traffic types, while the SVM model may capture complex boundaries more effectively in others.

The CBQ dataset offers an additional perspective by incorporating the performance of a Fine Tree classifier, as evaluated through the AUC metric. Here, both the Optimizable Tree and the Fine Gaussian SVM yield perfect TPRs for protocols such as ARP, SSDP, and TCP, yet the Fine Tree's consistently high AUC values deliver valuable insights into the classifiers' overall discriminative power. In particular, the CBQ results reveal that for protocols within the TLS family and MDNS, the Tree-based methods provide a more reliable representation of the underlying class distributions than the Fine Gaussian SVM.

Collectively, these experiments show that while Tree-based models – especially the Optimizable Tree – tend to achieve higher TPRs across a range of protocols, the optimal classifier choice may depend on the specific protocol characteristics and dataset properties. This nuanced understanding emphasises the importance of adopting a multifaceted evaluation framework when selecting classification models for network traffic analysis.

1) Dataset-Specific Performance for the Optimizable Tree: This study utilised the Optimizable Tree algorithm – the high performer among the evaluated models – as the benchmark to assess how well each dataset (or scheduling algorithm) produced distinguishable traffic patterns that led to improved ML outcomes. This comparative analysis provides insights into the intrinsic quality of the datasets concerning their capacity to delineate protocol-specific behaviours.

a) Consistent Performance for ARP and SSDP: All datasets (PRIO, HTB, and CBQ) achieved a perfect True Positive Rate (100%) for ARP and SSDP, indicating that these protocols exhibit highly distinguishable patterns across all datasets.

b) Divergent Outcomes for DNS and ICMP: DNS reveals a high TPR in the PRIO dataset (99.4%) and a relatively high value in the HTB dataset (92.1%), but a drastic drop in the CBQ dataset (23.1%), indicating that DNS traffic patterns are much less distinguishable under CBQ scheduling. ICMP is nearly perfect in both the PRIO (99.7%) and HTB (100%) datasets, but its performance decreases significantly in the CBQ dataset (51.1%), mirroring the trend observed with DNS.

c) Variability in MDNS and NTP: MDNS detection rates are markedly different: the HTB dataset achieved a perfect score (100%), compared to 40% in PRIO and 85.7% in CBQ. NTP shows a complete lack of detection in PRIO and CBQ (0%), but achieves 85.7% in HTB, suggesting that HTB scheduling captures NTP traffic features more effectively.

d) Performance in TLS Protocols: TLSv1 and TLSv1.2 show performance improvements: TLSv1 rises from 0% in PRIO to 37.5% in HTB and 68.4% in CBQ, while TLSv1.2 consistently performs well across datasets, reaching near-perfect detection in HTB (99.8%) and high values in PRIO (66.9%) and CBQ (77.7%). TLSv1.3 remains robust, with high TPRs across all datasets, indicating that its traffic patterns are reliably distinct regardless of the scheduling scheme used.

e) Inconsistent Results for QUIC and UDP: QUIC is only appreciably detected in the PRIO dataset (83.3%), with no detection in HTB or CBQ. In contrast, UDP exhibits high performance in PRIO (100%), but drops significantly in HTB (55.6%) and is nearly absent in CBQ (0.2%).

The analysis indicates that the HTB scheduling algorithm consistently produced the highest TPRs across various protocols. Notably, HTB achieved near-perfect detection for ARP (100%), ICMP (100%), MDNS (100%), and TLS protocols (with TLSv1.2 at 99.8% and TLSv1.3 at 99.9%) while also delivering robust performance for DNS (92.1%), NTP (85.7%), and TCP (92.4%). Although the PRIO dataset demonstrated high performance for some protocols, such as DNS (99.4%), QUIC (83.3%), and UDP (100%), it struggled with others, notably NTP and TLSv1, where detection was absent. Similarly, the CBQ dataset underperformed in several key areas, with substantially lower TPRs for DNS (23.1%), ICMP (51.1%), and UDP (0.2%). Based on these comparative findings, the HTB scheduling algorithm appears to yield the most distinct traffic patterns, enabling more effective ML across most protocols.

6.3.3 Contributions to the Field

The central innovation of this experimental study lies in its objective to differentiate and classify network scheduling algorithms (PRIO, HTB, and CBQ) solely based on intrinsic patterns observed in passively captured network traffic, without relying on explicit QoS metrics such as latency, jitter, or packet loss.

Why is this innovative and valuable?

1) Implicit Signature Hypothesis: It operates on the compelling hypothesis that different scheduling algorithms, by their very nature of managing queues and prioritising or

shaping packet flows, leave distinct, detectable “signatures” embedded within the resulting traffic patterns (e.g., packet timing, distribution, burst characteristics, protocol interactions over time). The goal is to leverage ML models to learn these implicit signatures directly from packet-level data (headers, timestamps, sizes) captured by tools such as Wireshark.

2) Avoiding Measurement Overhead: Traditional scheduler comparison often involves actively instrumenting the network or endpoints to measure QoS metrics. This approach can be complex, intrusive, and not always feasible, especially on intermediate network nodes or production environments. Our approach offers a potentially less intrusive method that relies solely on traffic observation.

3) Fundamental Pattern Recognition: This approach shifts the focus from the scheduling outcomes (QoS metrics) to the mechanisms reflected in the traffic flow. Successfully classifying schedulers based on these raw patterns demonstrates a deeper understanding and capability to model the subtle ways these algorithms manipulate packet streams.

Furthermore, even with the limitations discussed in the following subsection, the results provide preliminary evidence supporting the hypothesis that scheduler-specific signatures exist within raw traffic patterns and can be detected by ML models. The challenge (and future work) lies in optimising the methods (feature representation derived from raw data, model choice, data quantity/quality) to reliably and accurately capture these potentially subtle signatures without resorting to external QoS measurements.

6.3.4 Answer to Research Questions

In this subsection, we present the answers to the research problem and the research question for Experiment 2. Table 6.4 presents the classification performance for scheduler-specific datasets.

1) Research Problem: *Using raw operational network traffic, how can ML and DL classification models effectively differentiate scheduling disciplines (Prio, HTB, and CBQ)?*

Answer:

Table 6.4: Classification Performance and Observations across QoS Scheduler-Specific Dataset (PRIO, HTB, and CBQ).

Scheduler-specific dataset	Mean (macro-TPR across 12 key protocols)	Stand-out Model	Key Observations
HTB	≈88%	Optimisable Tree	Robust, consistent signatures in packet timing and spacing; easiest to learn.
PRIO	≈72%	Optimisable Tree	Clear imprint for latency-sensitive

			flows (DNS, UDP, QUIC), but weak or absent for others (NTP, TLSv1)
CBQ	$\approx 64\%$	Fine Tree (AUC-weighted)	The scheduler flattens burstiness, so traffic classes overlap and are harder to separate.

Supporting details to answer the research problem:

- **Models tested:** Optimisable Tree, Fine Tree, and Fine Gaussian SVM
- **Best overall discriminator:** the Optimisable Tree – highest TPRs on 8 of 12 protocols in HTB and PRIO, and tied 6 of 12 in CBQ.
- Protocols that are scheduler-agnostic, such as ARP and SSDP, achieved 100% TPR across all schedulers, indicating that not all traffic helps distinguish queueing policies.
- **Protocols that expose a scheduler’s “fingerprint”:**
 - ✓ HTB: ICMP, MDNS, TLSv1.2 and TLSv1.3 (TPR $\geq 99\%$).
 - ✓ PRIO: UDP, QUIC, DNS (TPR $\geq 99\%$ for DNS, 100% for UDP).
 - ✓ CBQ: Only TLSv1.3 remains $\geq 99\%$; DNS and ICMP drop sharply (23% and 51%).
- **Statistical strength:** All models perform far above random.

Conclusion: Yes – ML classifiers, especially tree-based ones, can reliably differentiate schedulers, with HTB being the most distinguishable, PRIO moderately so, and CBQ the least.

2) Scientific Question: *Is it possible to accurately classify and predict scheduling performance based solely on traffic patterns, and what insights do these predictions offer regarding the inherent characteristics of each scheduling discipline?*

Answer:

a) Feasibility of traffic-only prediction

- High protocol-level TPRs (≥ 0.85 for up to 10/12 protocols in HTB) and macro-AUC > 0.9 for CBQ confirm that explicit QoS counters are unnecessary – the performance imprint is already present in packet-level traces.
- Confidence scores correlate with how strongly a scheduler shapes each protocol; low confidence (DNS under CBQ) flags homogenised traffic with minimal QoS differentiation.

b) Insights into scheduler behaviour learned by the model: Table 6.5 presents the insights related to scheduler behaviour learned by the ML models.

Table 6.5: Insights into Scheduler Behaviour Learned by ML Models.

Discipline	Traffic-pattern signature captured by ML	What that reveals
PRIO	Bursty, low-jitter traces for high-priority flows; near-starvation of low-priority ones	It confirms aggressive real-time prioritisation; it is easy to fingerprint but may leak policy intent.
HTB	Regular inter-packet spacing matching token-bucket refill; uniform per-class caps	Scheduler parameters (rate ceiling, burst sizes) are “visible” in traces, enabling passive capacity audits.
CBQ	Smoothed, overlapping inter-arrival histograms; burst suppression	Provides traffic privacy at the cost of lower distinguishability – and, as shown in (latency tests), slightly higher delay for real-time traffic.

3) Conclusion: Accurate, traffic-only classification is feasible and informative. The ML models not only distinguish the queuing scheme but also implicitly rank how each scheduler shapes individual protocols, revealing strengths (HTB’s predictable shaping), weaknesses (PRIO’s starvation of low classes), and privacy characteristics (CBQ’s smoothing).

6.3.5 Limitations and Future Directions

While the findings of our experimental study, which aims to classify or predict network scheduling algorithms (PRIO, HTB, and CBQ) using ML models based on traffic patterns collected via Linux tc and Wireshark, are promising and validate our methodological approach, it is vital to interpret them within the context of the study’s limitations. Thus, understanding these constraints clarifies the path forward for more robust research:

1) Data Volume and Representativeness: The data collection lasted one hour for each scheduling algorithm, representing a limited snapshot of network activity. This constraint likely contributes to the observed performance variability, particularly the poor performance (0% TPR or low AUC) for specific protocols (e.g., HTTP, OCSP, sometimes QUIC, NTP, UDP, ICMPv6). These protocols might have been infrequent in the captured hour, or their distinguishing features may only emerge over extended periods or under different load conditions. Future work should prioritise significantly more extended data collection periods (spanning hours or days).

2) Computational Resources: Performing ML model training and optimisation solely on CPUs limited the practical scope of hyperparameter tuning and potentially the exploration of more complex model architectures. With the Optimizable Tree, tuning occurred, but resource constraints may have limited its extent or similar efforts for the

SVM and Fine Tree models. Access to GPU resources in future work would enable more thorough optimisation, experimentation with different architectures and faster iteration on longer datasets.

3) Feature Engineering and Protocol Complexity: The application of ML models directly to minimally processed network traffic data yielded highly encouraging results, with classification accuracies (specifically, True Positive Rates) exceeding 90% for numerous key protocols such as ARP, SSDP, TCP, and TLS variants across datasets associated with PRIO, HTB, and CBQ schedulers. This strong performance validates the core hypothesis that detectable, protocol-specific patterns exist within raw traffic data and demonstrates the capability of ML models to learn these patterns without extensive feature engineering. This finding supports the innovative approach of seeking scheduler-specific signatures directly within the traffic flow.

However, performance was not uniform across all protocols. Specific classes, such as QUIC, NTP, and UDP, exhibited significantly lower TPRs in some contexts. This suggests that while the raw features are sufficient for many protocols, they may not adequately capture the distinguishing characteristics of others, particularly within the current dataset's constraints (a 1-hour duration and limited scale). Although the high overall accuracy is promising, subsequent research should focus on addressing model robustness and performance under challenging protocols. One potential avenue is the investigation of minimal, pattern-preserving feature representations – for instance, the explicit calculation of inter-arrival times. Such features may better equip the model to discern subtle temporal dynamics arising from varied schedulers, thereby enhancing its ability to differentiate based solely on traffic patterns.

4) Future Directions for this Experimental Work

a) Enhanced Data Collection Strategy: Future work for this experiment should prioritise a more comprehensive data collection strategy to address the limitations of the initial setup. This involves capturing network traffic over significantly longer durations, potentially spanning days or weeks, to encompass diurnal variations and diverse traffic mixes, and ensure sufficient representation of less frequent protocols that are underrepresented in one-hour captures. Each scheduler should systematically collect data under varying network load conditions (low, medium, and high congestion), as algorithm behaviour and resulting traffic signatures can change drastically under stress.

b) Advanced Feature Engineering Focused on Scheduler Signatures: While initial results using minimally processed data were promising, future efforts should explore more targeted feature engineering specifically designed to capture the subtle temporal and structural differences introduced by scheduling algorithms, potentially improving classification, especially for challenging protocols. This requires moving beyond basic packet headers and explicitly calculating features such as inter-arrival time statistics (mean, variance, and percentile) within and between flows, developing metrics for burst analysis (size, duration, inter-burst gaps), and analysing flow duration and packet count distributions. Investigating sequences and distributions of packet sizes could reveal

patterns altered by shaping or policing schemes inherent in schedulers such as HTB and CBQ. Exploring graph-based traffic representations and their associated metrics could uncover high-level structured changes induced by different scheduling policies. A systematic comparison between models trained on raw data and those trained on engineered features will be crucial for determining the value added by explicit feature engineering in revealing scheduler-specific signatures.

c) Advanced Modelling and Optimisation with Enhanced Resources: To fully exploit the rich information embedded in scheduler-shaped traffic, future work should employ GPU-accelerated and distributed computing environments to broaden the depth and breadth of model exploration. With access to GPUs, extensive hyperparameter tuning – via Bayesian optimisation, genetic algorithms or automated ML frameworks (e.g., Auto-Sklearn, Google AutoML) – can be conducted across all classifier families, including tree ensembles, kernel-based SVMs, and emerging deep architectures. In particular, sequence-aware models (such as CNN-LSTM hybrids or Transformer encoders) should be evaluated for their ability to learn temporal dependencies in packet streams, while graph-based neural networks could capture higher-order flow interactions. Distributed training libraries (e.g., Horovod, Ray Tune) will enable scaling to multi-hour or multi-day captures, ensuring robust performance even on highly imbalanced or rare traffic classes. By systematically benchmarking classical and deep methods under varying network loads and scheduling conditions, these computational enhancements will elevate classification accuracy and clarify which modelling technique most effectively harnesses scheduler-induced signatures.

d) Explainable AI (XAI) for Signature Identification: To transform high-performance classifiers into practical diagnostic tools, researchers must peel back the “black box” and expose the specific traffic features that signify each scheduling scheme. Integrating model-agnostic interpretability approaches, such as Shapley Additive Explanations (SHAP) and Local Interpretable Model-Agnostic Explanations (LIME), will quantify the contributions of individual features (e.g., inter-packet intervals, burst lengths, packet-size distributions) to each prediction, both globally and per-sample. Furthermore, building upon these insights, surrogate rule-extraction techniques or decision-path visualisations can distil complex model behaviour into human-readable scheduling “signatures”, enabling network operators to validate that detected patterns correspond to genuine algorithm effects (for example, the smoothing behaviour of HTB versus the starvation threshold of PRIO). Embedding an explainable layer bolsters trust and accountability, guiding targeted feature engineering and enabling lightweight, interpretable monitoring solutions that can run in real time on production networks.

6.4 Findings and Implications: Comparison of Scheduling Algorithms Using DL Models with Data Captured from Operational Network.

6.4.1 Brief Context

The central purpose of these experiments was to compare distinct DL methodologies and data representations for their ability to classify network traffic based on the underlying

QoS scheduling algorithm (PRIO, HTB, CBQ, and a class for traffic captured under no QoS configurations, No_QoS). This investigation serves a dual role: assessing the efficacy of the models themselves (CNN, CNN-LSTM, and LSTM) and, through their performance, gauging the distinguishability of the traffic patterns generated by these different schedulers. Thus, three primary approaches were explored: An LSTM model applied directly to raw network traffic features (3 classes), a CNN, and hybrid CNN-LSTM models utilising image-based representations derived from network data (4 classes). A significant constraint, particularly for image-based approaches, was the limited dataset size, which necessitated careful consideration of overfitting and model robustness. The LSTM model, which processes raw data, established a strong benchmark, demonstrating excellent performance (99.5% accuracy) and stability. The image-based CNN-LSTM showed limited effectiveness. Notably, the image-based CNN model, while exhibiting performance variability across different experimental setups, likely due to data scarcity, demonstrated significant potential by achieving 87.50% validation accuracy in a specific run employing k-fold cross-validation, indicating that the scheduler's signatures might be discernible even within image representations under optimised conditions. However, ongoing work acknowledges the need to further address stability and generalisation.

6.4.2 Summary of Key Findings

The investigation yielded several key findings regarding the application of DL to QoS scheduler classification:

- 1) The LSTM model processing raw network traffic features demonstrated exceptional efficacy, achieving 99.5% validation accuracy with robust generalisation metrics (AUC=1.0, AP=1.0) and stable training dynamics, confirming the distinctiveness of PRIO, HTB, and CBQ scheduler signatures within raw data streams.
- 2) The models utilising image-based data representations faced considerable challenges, primarily attributed to the limited dataset size. While the CNN exhibited potential, achieving a notable 87.50% validation accuracy in a specific configuration employing k-fold cross-validation, suggesting these schedulers might also be distinguishable via image representations under optimal conditions, its performance proved highly variable across different experiments, revealing significant instability and susceptibility to overfitting, thus limiting definitive conclusions from this approach alone.
- 3) The hybrid CNN-LSTM architecture applied to image data proved largely ineffective in the configuration tested, failing to converge reliably. Collectively, these results underscore the critical influence of data representation and dataset sufficiency, highlighting the superior and consistent performance of LSTM on raw features for this task, while indicating that image-based approaches require substantial further development, particularly in terms of data volume or augmentation strategies, to achieve reliable and robust scheduler classification.

6.4.3 Contributions to the Field

This investigation makes several noteworthy contributions to the field of QoS management, despite the identified limitations (see subsection 6.1.3.4), particularly regarding dataset size for the CNN and CNN-LSTM methodologies, which are particularly relevant to the increasingly complex environments of 5G networks and beyond. The comparative analysis of different DL approaches and data representations for identifying active QoS scheduling algorithms yielded valuable insights applicable to automated network monitoring and policy verification. These contributions pave the way for more intelligent and adaptive QoS management systems.

Below are listed some of the potential contributions to the field of QoS management for 5G networks and beyond:

1) Demonstration of Automated QoS Scheduler Identification using LSTM: The outstanding success of the LSTM model operating on raw network traffic features provides strong empirical evidence for the viability of using DL to automatically identify active QoS scheduling mechanisms (such as PRIO, CBQ, and HTB) in near real-time. This capability is highly relevant for 5G and beyond, enabling automated monitoring, SLA verification, and potentially faster troubleshooting of QoS policy enforcement issues in complex, dynamic network environments.

2) Comparative Insight into Data Representation for Network Traffic Analysis: The experiments offer valuable insights by directly comparing the efficacy of raw feature-based analysis (via LSTM) versus image-based representations (via CNN/CNN-LSTM) for this specific QoS classification task. The clear superiority of the LSTM on raw data suggests that, at least for differentiating these schedulers, preserving the temporal or sequential structure inherent in raw traffic features may be more effective than relying on visual patterns derived from plotted representations, especially when image dataset sizes are constrained. This informs data preprocessing and feature engineering choices for future network traffic analysis tasks.

3) Highlighting Challenges and Potential of Image-Based Network Monitoring: While demonstrating significant limitations due to data scarcity, the investigation also revealed the potential of CNNs on image-based network data (achieving 87.50% accuracy in one run). This highlights both the promise and the critical challenges (data volume requirements and representation sensitivity) of using image-based DL techniques, which are often applied successfully in other domains, for nuanced network management tasks such as QoS verification. It underscores the need for substantial datasets or advanced techniques, such as transfer learning or sophisticated data augmentation, if one is to pursue this avenue.

4) Methodological Foundation for Verifying QoS Policy Implementation: By successfully distinguishing between different scheduling algorithms, this work lays a methodological foundation for developing tools to verify whether the intended QoS policies are implemented correctly at the data plane level in 5G network slices or specific service flows. This verification capability is crucial for ensuring that the diverse

performance requirements (latency, throughput, reliability) of different 5G services are met.

6.4.4 Answer to Research Questions

In this subsection, we present the answers to the research problem and the research question for Experiment 3.

1) Research Problem: *To what extent can DL models (CNN, CNN-LSTM) leverage image-based representations of network traffic data to predict scheduling outcomes and QoS performance?*

Answer: Under the constraint of the present study, image-based DL models can sometimes recognise the scheduling algorithm behind a flow; however, they do so far less consistently and with markedly lower ceiling performance than a sequence model that operates on the raw dataset.

a) Performance ceiling and variability

- The best-case image-based CNN run (with 5-fold cross-validation and careful regularisation reached 87.5% validation accuracy.
- Across repetitions, however, the same architecture fluctuated between near-chance and ~80% accuracy, signalling strong sensitivity to data splits and a tendency to overfit the limited image set.
- The hybrid CNN-LSTM did not converge reliably, indicating that the hoped-for “best of both worlds” (spatial filters plus temporal memory) was not realised with the data available.

b) Dataset dependence

- The decisive limitation is the volume and diversity of the image dataset. Even aggressive augmentation could not replicate the breadth of conditions captured by the raw feature stream, so CNN never saw enough exemplars to learn scheduler-specific visual cues robustly.

c) Comparative perspective

- A plain LSTM model at the raw-packet level achieved 99.5% accuracy and an AUC of 1.0, demonstrating that scheduling signatures are readily separable when their original temporal structure is preserved.
- Consequently, while image-based models can leverage spatial images related to network traffic data, their practical utility for real-time scheduler detection remains limited until much larger or better-curated image datasets, or transfer learning pipelines, are introduced.

2) Scientific Question: *Can advanced DL architectures capture complex spatiotemporal features in network data to improve scheduling prediction accuracy in live network scenarios?*

Answer: Yes, provided the architecture aligns with the data representation and sufficient training data are available. In this study, the LSTM – an architecture explicitly designed

for temporal sequences – fully captured the relevant spatiotemporal dynamics in raw flow statistics, whereas CNN-based image models captured them only partially.

a) Evidence for successful capture

- **LSTM on raw features** achieves 99.5% accuracy, a perfect AUC, and average precision, along with smooth learning curves, strongly indicating that the model has learned the scheduler-specific timing and volume patterns.
- **CNN on Images:** Occasional high-accuracy runs show that 2-D convolutional networks can extract spatial patterns that correspond (indirectly) to temporal burstiness and queue build-up, for instance, but not stably with the available data.

b) Evidence for limitations

- **CNN-LSTM on images:** failure to converge suggests that stacking temporal modelling on inadequately learned spatial features compounds, rather than solves, the data sparsity problem.
- **Generalisation gaps:** When unseen traffic conditions were introduced (e.g., validation, cross-validation, or test sets), the image model's performance dropped sharply, whereas the raw-feature LSTM remained steady, highlighting that the current image representation does not yet generalise to the full range of live traffic conditions.

c) Implications for live deployment

- The LSTM (raw feature sequence) model is deployment-ready for near-real-time scheduler identification in 5G slices, needing only lightweight preprocessing and offering millisecond-scale inference on modest hardware.
- Image-based approaches could become viable if:
 - ✓ Substantially larger labelled data is collected (synthetic traffic is generated) to expose CNNs to the breadth of scheduler behaviour.
 - ✓ Another option is to use transfer learning from vision models pre-trained on structurally similar traffic-image tasks, coupled with strong regularisation and domain-specific augmentation.

In summary, advanced DL models can learn complex scheduler-induced patterns, but the match between model type and data representation is critical. For the problem, raw-feature sequence models currently offer the most accurate, stable, and deployment-feasible solution. Thus, image-based CNN or CNN-LSTM pipelines remain promising research directions that require richer data and additional engineering to achieve comparable reliability.

6.4.5 Limitations and Future Directions

While this investigative study provides valuable insights into applying DL models for QoS scheduler classification using different data representations, a critical assessment necessitates acknowledging several inherent limitations. These constraints, stemming primarily from the dataset's characteristics and the scope of the methodologies explored, influence the interpretation and generalisability of the presented findings. Understanding

these limitations is crucial for contextualising the results and guiding future research directions. Below are presented some of the limitations of this investigation:

1) Severe Data Scarcity (Image-Based Models): The most significant limitation is the extremely small dataset size for the image-based approaches (CNN and CNN-LSTM). This might have directly contributed to overfitting, high performance variability across runs and cross-validation folds, and reduced overall confidence in the generalisation capabilities of these models.

2) Limited Devices and Scenario Diversity: Although collected from a live network environment, the involvement of a restricted number of devices limits the diversity of captured traffic patterns, user behaviour and application profiles. Consequently, the models (especially the highly successful LSTM) might have learned patterns specific to this limited setup and may not generalise effectively to broader, more heterogeneous network environments with a wider array of devices and conditions.

3) CNN Model Instability and Sensitivity: Despite showing potential, the CNN model exhibited high sensitivity to specific configurations and data subsets, resulting in performance variability. This instability makes it difficult to confidently ascertain its true potential without further investigation, stabilisation efforts, and likely more diverse data.

4) Limited Exploration of Architectures and Hyperparameters: The study focused on specific instances of CNN, CNN-LSTM, and LSTM architectures. Findings are pertinent to these configurations, and a more exhaustive exploration of model design and hyperparameters might reveal different performance characteristics or identify architectures better suited to this task.

5) Specificity of Image Representation Method: The images used as inputs for the CNN and CNN-LSTM were generated using MATLAB's plotting capability on the captured network data. This specific visualisation method may not optimally encode all discriminative features present in the raw data relevant to scheduler classification compared to other potential image encoding techniques, thus potentially limiting the achievable performance of the image-based models. The findings for these models are inherently tied to this particular representation choice.

6) Future Directions for this Work: Based on the identified limitations, this study identified several avenues for future research to build upon this work and enhance its robustness and applicability:

a) Data Expansion and Diversification: The most critical next step is significantly expanding the dataset, particularly for the image-based approaches. This involves collecting data from a wider variety of devices, diverse network conditions (varying load, congestion levels, and background traffic), different application types, and potentially more complex network topologies to improve model generalisation and mitigate the possible overfitting observed, especially with the CNN model.

b) Model Validation on Heterogeneous Data: Specifically, for the high-performing LSTM model, rigorous testing on external datasets collected from different networks and

under varied conditions is essential to confirm its generalisation capabilities beyond the specific environment of the initial data collection.

c) Advanced Image Representation and Augmentation: Exploration of alternative or potentially more feature-rich methods for encoding network data into images beyond standard plotting (e.g., Gramian Angular Fields and recurrence plots). Furthermore, investigate sophisticated data augmentation techniques designed explicitly for time-series or network data visualisation to artificially increase the size and diversity of the limited image dataset for CNN/CNN-LSTM models.

d) CNN Architecture Optimisation and Stabilisation: Future work might involve systematic hyperparameter optimisation for the CNN model. It might also include exploring different CNN architectures (e.g., ResNet variants or attention mechanisms) specifically chosen for their robustness on smaller datasets or their ability to capture relevant features from the selected image representation.

e) Broader Architectural Exploration: While LSTM performed well, exploring other sequence models (e.g., GRUs, Transformers adapted for time series) on the raw data could yield further improvements or offer different computational trade-offs.

6.5 Findings and Implications: Experimental Design of Delay Prediction Models (Hybrid (Linear+LSTM+Transformer), LSTM, Random Forest)

6.5.1 Brief Context

This experimental study aimed to enhance the prediction of network delay (latency) by comparing advanced DL models with a classical ML baseline. In particular, a novel hybrid sequence model combining a linear layer, LSTM, and Transformer was proposed and evaluated alongside a standard LSTM network and an RF model. The motivation was to determine whether incorporating temporal sequence learning (through LSTM and Transformer components) yields more accurate delay predictions than a well-tuned non-sequential approach. All models were trained and tested in a leakage-safe, device-wise split manner – meaning they were trained on data from one set of network devices and validated on a separate set of unseen devices – to rigorously assess generalisation. By framing network delay prediction as a time-series regression task, the study provided a controlled comparison of model complexity and sequence-awareness in the context of network performance monitoring.

6.5.2 Summary of Key Findings

1) Accuracy of Models: The RF baseline achieved the highest prediction accuracy, with a validation R^2 of ≈ 0.998 and the lowest error (validation RMSE ≈ 0.62 ms). This performance was near-perfect, indicating that the static feature relationships alone captured the network delay variability.

2) LSTM Performance: The pure LSTM model also obtained excellent accuracy, with a validation R^2 of ≈ 0.974 (validation RMSE of ≈ 2.36 ms). It explained the majority of the delay variance and outperformed the more complex hybrid model, despite its simpler architecture.

3) Hybrid Model Outcome: The proposed Linear + LSTM + Transformer hybrid model delivered comparatively lower accuracy, with a validation R^2 of ≈ 0.911 and higher error (validation RMSE ≈ 4.33 ms). It did not outperform the plain LSTM or the Random Forest. This underperformance could indicate that the added Transformer component and complexity did not translate into improved predictive power for this dataset.

4) Generalisation: Notably, all models maintained high training-to-validation performance (e.g., the Random Forest's train vs. validation R^2 were 0.999/0.998), indicating no severe overfitting under the device-level split. The strong validation results confirm that network delay can be reliably predicted for new devices using patterns learned from other devices.

5) Unexpected Efficacy of Baseline: A counterintuitive finding was that the simplest model (Random Forest, with no sequential input) achieved the best results. This implies that instantaneous network metrics (jitter, throughput, packet statistics, etc.) at a given time step contained sufficient information to accurately predict immediate delay, overshadowing any benefits from learning longer temporal dependencies. A similar observation has been noted in other contexts, where a tuned RF yielded performance comparable to that of an LSTM, even when time-sequence factors were taken into account.

6.5.3 Contributions to the Field

This experimental study contributes to the field of network performance modelling and time-series prediction in several important ways:

1) Novel Hybrid Architecture: It introduced and evaluated a new hybrid DL architecture (Linear-LSTM-Transformer) for regression on network time-series data. While the hybrid model did not outperform simpler models in this case, the study provides a blueprint for integrating Transformer layers with recurrent networks for similar forecasting problems, highlighting challenges and considerations (such as tuning complexity and risk of overfitting).

2) Comparative Evaluation: The study provides a rigorous comparative analysis of three modelling paradigms – a deep sequence model with attention mechanisms, a classical recurrent network, and a non-sequential ensemble method – on the same network delay prediction task. Such side-by-side comparisons are valuable to both researchers and practitioners, as they quantify the trade-offs in complexity versus accuracy.

3) Generalisation Methodology: By enforcing a device-wise train/validation split (e.g., grouping data by device), the research design ensured that the models were evaluated on truly unseen entities. This methodology, which prevents data leakage across devices, is a significant contribution to the evaluation practices in network analytics. It demonstrates a reliable way to assess how well models generalise to new network nodes or environments.

4) Empirical Insights: The finding that a non-sequential model (Random Forest) can match or exceed the accuracy of sequence-based DL in predicting network delay is a noteworthy insight. It contributes evidence to the ongoing discourse on when advanced temporal models are warranted. In scenarios where domain features already capture the relevant temporal effects (e.g., current throughput or jitter that encapsulates recent history), a simpler model may suffice. This insight helps refine our understanding of feature importance and temporal dynamics in network QoS data.

6.5.4 Answer to Research Questions

1) Research Question 1: *Which modelling approach yields the most accurate predictions of network delay?*

The experiments revealed that the RF baseline provided the most accurate predictions, achieving the highest R^2 (≈ 0.998) and lowest error on the validation set. In comparison, the DL models were slightly less precise – the LSTM model was the second-best (validation R^2 (≈ 0.974)), and the hybrid model trailed further (validation R^2 (≈ 0.911)). In this study, the classical ensemble method outperformed both deep-learning approaches in predictive accuracy.

2) Research Question 2: *Does incorporating a Transformer component into an LSTM (the hybrid model) improve performance over a standalone LSTM?*

No. Including a Transformer encoder alongside an LSTM did not improve prediction performance in this context. On the contrary, the hybrid model underperformed the pure LSTM model (and even the simpler Random Forest). The LSTM-alone model achieved a lower error and higher R-squared value than the Linear-LSTM-Transformer model. This suggests that the added complexity and attention mechanisms did not yield a performance gain for immediate network delay prediction, possibly because the LSTM did not adequately capture sequential dependencies or because the hybrid's complexity was not fully utilised with the available data.

3) Research Question 3: *How effectively can network delay be predicted from QoS metrics, and what does this imply about the necessity of DL for this task?*

Network delay in this dataset can be predicted with very high accuracy using the available QoS features. All three models explained a large proportion of the variance (even the

lowest performing model exceeded 90% R^2 on validation). This suggests that the chosen feature set (e.g., throughput, loss rate, and jitter) is highly informative for predicting delay. The necessity of DL for this task appears limited, given that an RF achieved the best results, which implies that, at least for short-term delay prediction, complex sequence modelling was not strictly necessary. Furthermore, DL models such as LSTMs may still be advantageous for capturing longer-term temporal patterns or nonlinear feature interactions; however, in this study, their benefits were marginal. The excellent performance of the baseline highlights that simpler models, when carefully tuned, can be highly effective for network delay forecasting, emphasising the importance of benchmark comparisons before opting for more elaborate neural network solutions.

6.5.5 Limitations and Future Directions

Despite the promising results, this research study has several limitations that open avenues for future work. A primary limitation is the dataset's scope: the models were trained and evaluated on a specific wireless QoS dataset (with a fixed set of device types and network conditions). The extraordinary accuracy achieved (especially by the Random Forest) might not generalise to all network environments – for example, in scenarios with more volatile traffic patterns or unseen feature distributions, performance could degrade. Future studies should validate these findings on a more diverse dataset (e.g., different network types, larger-scale deployments, or longer time spans) to ensure the model generalises broadly.

Another limitation is that the evaluation focused on one-step-ahead prediction delay (predicting the immediate next delay value). It remains unclear how the models would perform for longer-horizon forecasting or under concept drift (when network behaviour changes over time). Subsequent research could extend the modelling approach to multi-step predictions, assessing whether sequence-based models then gain an advantage as the prediction horizon increases. The hybrid model's design and training may be further optimised – for instance, by exploring deeper Transformer layers, alternative fusion strategies between the LSTM and Transformer components, or larger sequence lengths – to determine if its performance can surpass that of simpler models in more challenging settings.

Finally, interpretability and real-time deployment considerations were beyond the scope of this experimental study. Although the RF model performs best, it is relatively more straightforward to interpret (feature importance can be extracted) than DL models. Future work may involve analysing feature importance or using SHAP values in the RF to identify which network metrics most strongly influence delay, yielding insights for network engineering. Evaluating the computational efficiency and inference speed of each approach could be valuable, especially if these models are to be deployed in live network monitoring systems. Incorporating these aspects would provide a more holistic understanding of the trade-offs between model complexity, transparency, and practicality in real-world applications.

6.6 Findings and Implications: Transformer-Based RL Agent for Network Traffic Scheduling

6.6.1 Brief Context

This study introduced an advanced framework for optimising traffic scheduling in 5G WSNs, which are increasingly critical in applications that demand URLLC, such as industrial automation, autonomous vehicles, and telemedicine. This framework integrates DRL with Transformer neural networks to effectively address the inherent complexities of network resource allocation and real-time packet prioritisation. By formulating the scheduling problem as a Markov Decision Process, the approach leverages sophisticated state representations, action spaces, and tailored reward functions to adaptively manage and prioritise network traffic. To validate the system experimentally, we deployed it in a live network environment and systematically assessed its performance through real-time ping-request scheduling and response evaluation. The empirical outcomes demonstrate significant promise, validating the Transformer architecture's effectiveness in dynamically optimising network-scheduling tasks and suggesting strong potential for broader applicability and scalability in complex network management scenarios.

6.6.2 Summary of Key Findings

This subsection presents the key findings of this investigative experimental work concisely. Integrating Transformer-based DRL into 5G traffic scheduling delivers robust performance, achieving consistently high scheduling success rates and low latency in real-world tests. The model's rapid convergence and favourable comparison to state-of-the-art methods underscore its effectiveness and scalability in meeting stringent QoS requirements.

1) Robust Scheduling Performance: This study demonstrates that integrating DRL with Transformer neural networks results in robust real-time scheduling in 5G WSNs. The proposed approach consistently achieves a 100% scheduling success ratio and maintains low latency, often 10 ms, even under dynamic network conditions.

2) Effective Convergence: Experimental results across various training durations (250, 350, and 500 epochs) show steady convergence, with the training loss effectively reducing to zero and cumulative rewards increasing linearly. These outcomes confirm the model's capacity to adapt and optimise its decision-making in real-world network scenarios.

3) Competitive Benchmarking: Comparative analysis indicates that the Transformer-based RL scheduler is competitive with and outperforms other simulation-based methods in some respects. Recorded latency values between 2.5 and 8.5 ms underscore its potential to meet stringent 5G QoS benchmarks.

4) Scalability and Efficiency: This investigative work also underscores the scalability of the approach and highlights its efficiency in resource-constrained and heterogeneous environments. This positions the proposed model as a viable foundation for future applications in more complex 5G networks and diverse IoT scenarios.

6.6.3 Contributions to the Field

This experimental work advances the field by integrating DRL with Transformer neural networks to address the complexities of real-time traffic scheduling in 5G networks. The proposed approach achieves consistently high scheduling success, low latency, and robust adaptability to dynamic network conditions by harnessing advanced state representations and sequential processing. Its scalability in heterogeneous and resource-constrained environments further underscores its practical applicability. At the same time, the novel methodological integration lays a solid foundation for future innovations in dynamic resource allocation and adaptive network management across diverse telecommunications and IoT scenarios.

1) Innovative AI-Based Scheduling: This work introduces a groundbreaking framework that integrates DRL with Transformer neural networks to address real-time traffic scheduling challenges in 5G WSNs. This novel architecture utilises advanced state representation and sequential processing capabilities, providing a strategic solution for latency-sensitive applications.

2) Real-Time Adaptability and Robustness: The study presents a model that dynamically adapts to varying network conditions by framing the scheduling task as a Markov Decision Process. The empirical results, which consistently reveal high scheduling success ratios and low latency, underscore its robustness and potential to support critical applications, such as industrial automation and telemedicine.

3) Scalability in Heterogeneous Network: The contributions extend to the scalability of network management, demonstrating that the proposed approach is effective even in resource-constrained environments such as IoT and edge computing scenarios. This work thus provides a viable pathway for deploying advanced scheduling strategies in increasingly dense and complex wireless networks.

4) Foundations for Future Research: Beyond practical applications, integrating Transformer architectures with RL enriches the methodological landscape for both telecommunications and AI research. This advances current state-of-the-art network management techniques and lays the groundwork for further innovations in dynamic resource allocation and adaptive learning in next-generation networks.

6.6.4 Answer to Research Questions

The results effectively validate the proposed MDP framework for network scheduling. The “Training Loss over Epochs” plot provides direct evidence that the RL agent is learning; its decreasing trend indicates that it is successfully refining its policy to better map states to actions that maximise future rewards. The “Average Reward over Epochs” plot confirms the outcome of this learning process, revealing that the agent’s learned policy consistently selects actions (network paths) that yield high rewards, fulfilling the objective of minimising delay. Finally, the “Final Average Latency” and “Final Average Throughput” charts illustrate the practical consequences of this optimised policy: the agent has successfully identified and prioritised the most efficient network routes,

demonstrating that framing the scheduling problem as an MDP allows the agent to discover a strategy that enhances overall network performance autonomously.

6.6.5 Limitations and Future Directions

While the proposed framework yields promising results, a few limitations remain that offer avenues for further research. Below are some of the identified limitations:

1) Limited Scale of Experiments: The evaluations were performed in a controlled, small-scale network environment, which may not capture the full complexity and variability in larger or more heterogeneous deployments. Thus, future studies should extend the experimental setup to validate the approach’s scalability under more diverse and realistic network conditions.

2) Increased Computational Complexity: Incorporating Transformer-based architectures enhances sequential data processing and introduces additional computational overhead. This complexity may challenge implementation in energy-constrained settings or require further optimisation when scaling up to support extensive networks.

3) Need for Broader Experimental Validation: Even though the evaluations were conducted in a real-world live network, they were confined to specific conditions and configurations. Expanding the range of experimental scenarios to include a broader set of live network environments would further validate the proposed approach’s robustness, adaptability, and generalisability.

6.7 Broader Implications for 5G and Emerging Applications Domains

6.7.1 Broader Implications for Experiment 1: Comparative Evaluation of PF and RR Scheduling

1) Transforming How Networks Are Evaluated and Tuned: The broader impact of this study, particularly Experiment 1, highlights the importance of latency predictability. It suggests that predictability can be as crucial as, or even more significant than, traditional metrics such as throughput or packet loss. By introducing “QoS predictability” as a quantifiable target, the work encourages operators and researchers to move beyond chasing ever-higher averages and start optimising for tight, consistent performance envelopes – precisely what delay-sensitive services such as cloud gaming, telesurgery and XR demand

2) Bridging Theory and Production Network: Running the comparison in a busy, un-isolated WLAN demonstrates that advanced analytics can be seamlessly integrated into everyday infrastructure without requiring lab-grade instrumentation. That lowers the barrier for campus and enterprise administrators to replicate the method with their packet captures, bringing academically rigorous evaluation into the operational mainstream.

3) Elevating ML-Aided Network Engineering: The successful use of EGPR signals that probabilistic ML models are mature enough for online performance audits and “what-if”

forecasting. Because GPR outputs both a prediction and its confidence interval, operators gain a risk-aware lens for real-time decision-making, for instance, when and how aggressively to switch schedulers or throttle specific flows.

4) A Replicable Hybrid Methodology: By fusing direct KPI statistics with an ML-based second opinion, our approach delivers a blueprint that other researchers can extend to new schedulers, radio technologies, or KPI sets. The framework’s modularity – particularly its Python-only tooling – makes it easy to incorporate into continuous integration (CI) and continuous delivery/deployment (CD) pipelines for network slicing, or to feed into closed-loop (Self-Optimising Network (SON)) controllers.

5) Enabling Standard and Policy Discussions: Results that quantify latency stability could inform updates to 3GPP QoS classes, Internet Engineering Task Force (IETF) transport drafts, or Multi-access Edge Computing (MEC) SLA templates. Regulators and service-level negotiators can point to “95-percentile delay predictability” metrics rather than coarse averages, helping crystallise enforceable, customer-centric performance guarantees.

6) Catalysing Interdisciplinary Research: Our experimental work operates at the intersection of wireless scheduling, stochastic modelling and applied ML, illustrating how insights in one domain enrich the others. This intersection invites collaboration among networking experts, statisticians, and data scientists to co-design schedulers that are fair, high-throughput, learnable, and controllable in real time.

7) Guiding Future Innovation: Because the study pinpoints instantaneous throughput and jitter as first-order delay predictors, radio-access and MAC-layer researchers can focus optimisation budgets on stabilising exactly those variables. Meanwhile, the demonstrably superior predictability of Proportional Fair under dynamic load encourages chipset vendors to expose fine-grained PF knobs – or hybrid PF/RR schemes – to higher-level controllers.

In essence, this experimental study reframes scheduler evaluation around predictability, delivers a turnkey analytics pipeline that others can adopt, and seeds new conversations – from standard bodies to DevOps war rooms – about how wireless networks should be measured, modelled, and ultimately steered for the next generation of real-time services.

6.7.2 Broader Implications for Experiment 2: Classification-Based Comparison of Scheduling Algorithms

1) Operational Visibility and Troubleshooting: Network operations teams can retrofit “black-box” links – including legacy metro-Ethernet, third-party Customer Premises Equipment (CPEs), and even opaque virtual switches in Virtual Private Clouds (VPCs) – with a software-only monitor that displays the active queueing scheme.

Misconfigurations (HTB slipped to default FIFO after a firmware rollback) could be caught in minutes instead of days.

2) Security and Policy Auditing: Because schedulers imply policy (for example, PRIO may give Voice over Internet Protocol (VoIP) unfair precedence over bulk backups), regulators and auditors can use this passive technique to verify compliance without requiring privileged access. On the other hand, attackers might exploit the same fingerprint to map QoS strategies, raising the stakes for covert, CBQ-like schedulers that mask their patterns.

3) Network Automation and AIOps: Embedding the classifier in an AI for IT Operations (AIOps) loop enables SDN controllers to observe, infer, and adapt in real-time. For instance, if congestion persists despite HTB shaping, the controller could migrate critical flows to an alternative path or dynamically adjust the token-bucket rates.

4) Roots of Explainability for ML-Driven Networking: Tree-based models not only score the highest but are also interpretable. Their splitting rules expose features (for example, the 95th percentile inter-arrival gap) that matter most, offering human-readable insights that might be a welcome antidote to the “black-box” stigma surrounding deep packet inspection with DL.

5) Emerging Application Domains

- **5G/6G network slicing:** Mobile operators can fingerprint tenant slices to ensure that eMBB, URLLC and mMTC slices are enforced by the underlying HTB/CBQ scheduler.
- **Edge/IoT gateways:** Passive verification of PRIO vs. CBQ on constrained gateways helps guarantee that OTA updates do not starve sensor traffic.
- **Cloud-native microservices:** Service meshes, for example, could monitor east-west traffic to confirm that latency-critical Remote Procedure Calls (RPCs) receive the intended shaping.
- **Industrial control and autonomous vehicles:** Safety-critical flows benefit from real-time confirmation that deterministic queueing (e.g., TAS-like schedulers emulated via HTB) is active without halting production lines for intrusive tests.

In summary, passive ML-based fingerprinting of queuing disciplines is feasible and provides valuable insights. It promotes low-overhead observability, strengthens compliance audits and seeds new automation loops – while reminding network architects that even “hidden” scheduler choices leave footprints visible to a keen enough eye (or decision tree). As networks evolve toward fully autonomous, slice-aware and latency-deterministic architectures, such traffic-centric analytics will become a cornerstone capability rather than a research curiosity.

This experimental work demonstrates that AI classifiers – particularly Optimisable Decision Trees – can reliably distinguish between PRIO, HTB, and CBQ scheduling algorithms solely from observed network traffic, opening the way for automated, passive QoS monitoring tools for next-generation networks. Future enhancements in data volume, feature engineering, and computational resources (GPU-accelerated hyperparameter tuning) will further strengthen this approach, enabling fine-grained, explainable insights into scheduler behaviour without intrusive measurement.

6.7.3 Broader Implications for Experiment 3: DL-Based Scheduler Identification

1) Zero-Touch QoS Assurance in 5G/6G Networks: A lightweight LSTM model that recognises the actual scheduler (HTB, PRIO, CBQ, or “none”) with $\approx 99\%$ accuracy provides a drop-in sensor for closed-loop service-management systems. Operators can embed it at edge nodes to verify, within a few milliseconds, that the data plane is enforcing slice-specific QoS policies and then trigger automated remediation when a mismatch is detected.

2) SLA Verification and Regulatory Compliance: Regulators and enterprise customers increasingly demand provable adherence to performance guarantees. A model that can fingerprint the scheduler directly from traffic flows provides carriers with an objective, auditable metric for “Was the promised queuing discipline applied?” – Something the current KPI dashboard infers only indirectly.

3) Data Representation Guidance for ML on Network Telemetry: The stark performance gap between the raw-feature LSTM and image-based CNN/CNN-LSTM offers a concrete lesson: preserve the natural temporal structure first; visual encoding helps only when paired with abundant data or transfer learning. This insight generalises to other telemetry tasks (anomaly detection and congestion-control identification) and can save future researchers months of exploratory effort.

4) Catalyst for Public Traffic Image Datasets: The instability of the CNN results highlights the lack of large, labelled corpora of traffic-image sequences. Publishing this study’s capture scripts, labelling pipeline, and raw traces would seed a benchmark that inspires community-wide progress, similar to how ImageNet accelerated computer vision.

5) Security and Privacy Side Effects: Fingerprinting queuing schemes can be a double-edged sword: defenders gain a diagnostic tool, but adversaries might also reveal a network’s traffic-shaping strategy and adapt their exploits accordingly. Our work informs discussions on protective monitoring and threat modelling in next-generation mobile cores.

6) Towards Explainable Network AI: Because the LSTM’s salient features correspond to interpretable traffic dynamics (burstiness, inter-arrival patterns, queue build-up), the model can be probed with SHAP/attention techniques to expose why a flow is labelled

“CBQ”. This paves the way for **explainable QoS analytics**, increasing operator trust in AI-driven Network Operations Centre (NOC) tooling.

7) Template for other Spatiotemporal Classification Tasks: The methodological comparison – raw sequence data vs. image data vs. hybrid – is a blueprint for evaluating data-to-architecture fit in any domain where temporal signals can be rasterised (for instance, industrial IoT, financial tick data, smart-grid telemetry). Researchers can replicate the pipeline to decide early whether an image pipeline is worth the cost.

This work offers a practical blueprint for observing, verifying, and ultimately automating QoS enforcement in dynamic, sliced 5G/6G networks while crystallising broader best practices for data representation in ML. By demonstrating that an LSTM fed raw, sequential traffic features attains near-perfect scheduler-identification accuracy (99.5%, AUC = 1.0). In contrast, the image-based CNN model peaks at 87.5%, and the CNN-LSTM model struggles to converge under the same data-scarce conditions; the study clarifies that preserving temporal structure is critical when training with limited samples. At the same time, it highlights that CNN performance could improve with larger or augmented image datasets, transfer learning, or more sophisticated regularisation, pointing researchers towards the data volume and representation choice that will govern future advances both inside and beyond telecommunications.

6.7.4 Broader Implications for Experiment 4: Predictive QoS Forecasting

The ability to accurately predict network delay in real time has far-reaching implications for emerging 5G WSN applications. High-fidelity delay prediction using real-time QoS metrics can enhance reliability and efficiency across several critical domains, as outlined below:

1) Autonomous Vehicles: Autonomous and connected vehicles rely on ultra-low-latency V2X communication for safety and coordinated driving. By leveraging high-accuracy delay prediction, 5G networks can provide in-advance warning of impending latency spikes or QoS drops, allowing on-board control systems to proactively adapt – for instance, by slowing down or temporarily suspending non-critical functions so that safety is never compromised under the predicted conditions. Such predictive QoS capabilities enable vehicles to avoid sudden communication disruptions, maintaining the reliability needed for tele-operated driving and cooperative manoeuvres. In turn, this leads to broader societal benefits: fewer accidents due to connectivity loss, smoother traffic flow from well-coordinated vehicles, and increased public trust in autonomous transportation. The economic impact could also be significant, reducing collision-related costs and paving the way for efficient autonomous logistics and mobility services.

2) Remote Healthcare: In remote healthcare applications – from telemedicine consultations to robotic telesurgery – the stakes for network performance are life-and-death. 5G’s ultra-reliably low-latency communication is crucial for transmitting high

definition video, haptic feedback, and vital sensor data in real-time. By applying the study's predictive delay modelling, healthcare networks can anticipate and mitigate latency fluctuations before they disrupt a medical session. For instance, surgeons operating remotely could receive early warning of QoS degradation, allowing them to postpone or adjust a robotic procedure rather than risk a mid-surgery network failure. These AI-driven predictive analytics promise to broaden the reach of telesurgery and enhance the precision and safety of remote procedures. Likewise, for routine telehealth and emergency response, consistently low latency enables patients to consult specialists without travel and ensures that wearable and IoT medical devices relay critical data promptly. This improved reliability translates into societal benefits: timely care in rural or underserved areas, better patient outcomes through continuous monitoring, and greater overall confidence in remote healthcare services. Economically, such capabilities can lower healthcare costs by reducing hospital visits and enabling efficient use of medical expertise across distances.

3) Industrial Automation: Industrial automation in smart factories and critical infrastructure depends on deterministic, real-time communication among sensors, machines, and control systems. Unpredictable network delays can lead to production slowdowns or even safety hazards on the factory floor. By incorporating AI-driven, high-accuracy network delay predictions, 5G-based industrial WSNs can proactively adjust data flows or control commands to prevent performance bottlenecks. For instance, if a latency increase is predicted, control algorithms might reschedule tasks or switch to local edge processing to maintain seamless operation. This foresight helps ensure that assembly robots, autonomous guided vehicles, and process control loops remain synchronised without unexpected downtime. In practical terms, maintaining ultra-low-latency connectivity even in harsh industrial environments enables real-time monitoring and robotic control with unwavering reliability. The technical payoff is higher throughput and productivity. Additionally, avoiding network-related interruptions reduces costly downtime and maintenance delays, resulting in significant economic benefits. Predictive QoS empowers Industry 4.0 applications to run more safely and efficiently, supporting new levels of automation and operational excellence.

6.7.5 Broader Implications for Experiment 5: Transformer-Based RL Scheduler

This research work has a significant impact on emerging telecommunications applications by providing a robust framework for real-time network scheduling. By combining DRL with Transformer neural networks, the proposed approach ensures ultra-reliable, low-latency communication, which is vital for next-generation applications such as autonomous vehicles, telemedicine, and industrial automation. This breakthrough enhances QoS, enabling networks to meet the demanding requirements of modern, dynamic environments.

The model's intelligent, real-time adaptability facilitates dynamic resource allocation in rapidly changing network conditions. This capability supports the seamless integration of emergent services by allowing networks to adjust to variations in traffic load and device

density. Consequently, network operators can effectively manage critical applications, even under peak demand or unexpected disruptions, ensuring consistent performance and reliability.

Furthermore, the scalability demonstrated in this live network environment underscores the framework's potential to extend beyond small-scale deployments. The mode's ability to maintain high scheduling success rates and low latency in real-world settings suggests that future implementations could support vast and heterogeneous networks, a key requirement for smart cities and massive IoT ecosystems. This robust performance lays a solid foundation for future innovations in telecommunications, particularly as the industry moves towards 6G and beyond.

Overall, the broader impact of this research is its contribution to more agile, resilient network infrastructure. By employing advanced AI techniques for adaptive scheduling, the work addresses challenges in 5G networks and establishes a pathway for future developments to support an ever-growing array of emergent and high-demand telecommunication applications.

6.8 Recommendations for Practitioners and Researchers

6.8.1 Recommendations for Experiment 1

1) Network Operators and Campus IT Teams: Prioritise PF on access points and routers that carry voice, XR, cloud gaming, or control-loop traffic, and reserve RR for bulk or best-effort SSIDs. Add lightweight probes that export instantaneous throughput and jitter every few hundred milliseconds, then stream those counters through an EGPR model trained on your traces. The model's prediction error becomes an early warning signal; thus, when it rises, one can pre-emptively throttle offending flows or switch the scheduler before users feel the delay spike.

2) Equipment and Firmware Vendors: Expose run-time scheduler selection and tuning hooks (e.g., REST or gNMI APIs that toggle PF <-> RR or adjust PF fairness weights) so controllers can respond to EGPR alerts. The Software Development Kit (SDK) should ship with reference EGPR inference code or Open Neural Network Exchange (ONNX) blobs and include KPIs, such as instantaneous throughput, in its default telemetry. These additions turn “predictability-aware” scheduling into a differentiating product feature and pave the way for fully closed-loop optimisation.

3) Standard and Policy Bodies (3GPP, IETF, Wi-Fi Alliance): This study recommends extending QoS profiles to include a predictable-latency indicator – for instance, the 95-percentile absolute error of a validated delay-prediction model – alongside mean delay and jitter. Update certification test plans to require devices to collect sub-second throughput samples (not just one-second averages) and demonstrate bounded prediction error under mixed traffic. Such requirements translate the abstract idea of “consistency” into an enforceable metric for service-level agreements.

4) Research Community: The research community is recommended to replicate the PF-versus-RR study using Wi-Fi 6E, 5G NR, and heterogeneous mobility traces to test

external validity. EGPR can be compared with attention-based temporal models that may capture Device 3-type anomalies without losing interpretability. Explore adaptive or hybrid PF/RR schemes guided by live EGPR forecasts and publish the resulting datasets to accelerate cross-disciplinary work between networking and ML.

5) DevOps and CI/CD Engineers: The recommendation of this study for this audience is to insert a predictability gate into the deployment pipeline. Every firmware image or configuration change pushed to staging should replay recorded traffic through the EGPR model; for instance, if the forecast error or predicted 95-percentile delay exceeds a predefined threshold, the pipeline blocks the production deployment. Complement this gate with Canary releases and automated rollback triggers tied to the same prediction-error metric, ensuring that regressions in latency stability never reach users.

In summary, treat latency predictability as a primary design objective, instrument the network with high-granularity throughput and jitter metrics, and let statistically grounded models, such as EGPR, steer day-to-day operations and long-term scheduler evolution.

6.8.2 Recommendations for Experiment 2

1) For Network Operations and Engineering Teams

a) Adopt Passive Fingerprinting Early in the Troubleshooting Chain: Deploy the tree-based classifier as a lightweight sensor on a span port or software taps. Because HTB, PRIO, and CBQ leave readily detectable timing “signatures,” the tool can confirm within minutes whether a queuing change, a firmware rollback, or a mis-push has silently reverted an interface to FIFO. Doing so short-circuits the need for intrusive active probing and accelerates mean-time-to-innocence for the network.

b) Prioritise HTB for Deterministic Slices: In production links that must honour explicit rate ceilings (for instance, 5G slice traffic, broadcast video backhaul), HTB proved the most distinguishable and the most transparently auditable. Thus, use that visibility to create closed-loop policies: if the passive meter shows the token bucket is too permissive or too strict, automation can adjust class rates.

c) Use CBQ Where Privacy Trumps Fine-Grained QoS: If, for instance, the operational goal is to blur application patterns or to resist traffic fingerprinting attacks, CBQ’s smoothing effect is a feature, not a bug; its lower TPRs indicate that it masks protocol-specific bursts more effectively than PRIO or HTB.

2) For Equipment Vendors and SD-WAN Providers

a) Expose Scheduler Telemetry Hooks: Tree-based models deliver high accuracy; however, a one-bit “Which scheme is active now?” counter in firmware would give operators ground truth and enable supervised retraining after software upgrades. Providing such hooks differentiates products without compromising proprietary scheduler logic.

b) Package a “classifier-as-feature”: Embedding the Optimizable Tree model (or its distilled rules) directly in edge CPE and virtual routers enables the device to self-report misconfigurations, feeding AIOps dashboards with rich, context-aware alerts.

3) For Standard Bodies and Regulators

a) Define a Passive-Verification Reference Test: The European Telecommunications Standard Institute (ETSI), the IETF, the IP Performance Metrics (IPPM) working group, or the Metro Ethernet Forum (MEF) working groups could standardise a short capture procedure and a confusion-matrix benchmark, allowing operators to prove, without privileged access, whether they deliver the queuing service levels promised on paper.

b) Mandate Transparency for Priority Schedulers in Regulated Sectors: Because PRIO can starve low-class traffic, utilities, public safety networks, and net-neutrality jurisdictions may require periodic passive audits to demonstrate that critical flows are not unduly harmed.

4) For the Research Community

a) Scale Beyond Three Disciplines: Extend the dataset to include modern variants, such as Deficit Round Robin, FQ-CoDel, and Time-Aware Shaper (TAS), as well as parameter drifts, including changes to the HTB bucket depth. Self-supervised learning could reduce the burden of manual labelling.

b) Probe Explainability: Use feature importance scores and surrogate models to reveal why particular inter-arrival gaps scream “HTB” or “PRIO”. That yields insights for both optimisation and attack-surface reduction.

c) Investigate Adversarial Resilience: How robust are tree-based models if attackers can construct traffic that mimics benign scheduler fingerprints? Studying adversarial traces will harden passive verification pipelines before they reach mission-critical networks.

5) For Security Practitioners

a) Treat Scheduler Fingerprints as Threat-Intel Signals: For example, a sudden, unexpected switch from CBQ to PRIO on an east-west link could indicate a compromised virtual switch whose policy has been rewritten for data exfiltration. Thus, feed the classifier outputs into Security Information and Event Management (SIEM) correlation rules to raise the alarm.

6) Audiences and Their Interests

Table 6.6 summarises the key stakeholder groups that benefit from this study, spanning network operators, vendors, research, regulatory, and security communities by mapping each audience to (i) why passive visibility into live queuing behaviour matters to them and (ii) the most relevant emphasis for adoption, from operator-grade tooling and AIOps integration, to product firmware hooks, publishable datasets and robust models, audit-ready compliance tests, and security oriented IOC/IDS-SIEM integration.

Table 6.6: Stakeholder Perspective and Recommended Focus Areas for QoS Classification and Telemetry.

Audience	Why They Care	Recommended Focus
Network operators and SREs	Need rapid, non-intrusive visibility into live queuing behaviour.	Provide comprehensive tooling and runbooks, and integrate the solution with AIOps to give operators rapid, non-intrusive insight into live queuing behaviour.
Equipment vendors and SD-WAN providers	Can embed classifiers as value-added features and expose telemetry.	Firmware hooks product road maps.
Researchers and ML practitioners	Seek new datasets and modelling challenges in real traffic	Dataset publication, explainability, and robustness.
Regulators and Standard bodies	Compliance with QoS or neutrality rules must be verified without deep access.	Develop a passive-audits test suite certification criterion.
Security and threat-hunting teams	Queueing anomalies can be side-channels or compromise indicators	Fingerprints are as IOC and integrate with IDS/SIEM.

6.8.3 Recommendations for Experiment 3

1) Mobile Network Operators and Service Providers: Begin by deploying lightweight LSTM-based verification probes at edge or gateway nodes; these sequence models infer the active scheduler with ~99.5% accuracy in sub-millisecond time, giving operators a real-time view of whether PRIO, CBQ, HTB – or no QoS at all – is being enforced on each flow. Couple the probes to a closed-loop orchestration stack (e.g., Open Network Automation Platform (ONAP) or an O-RAN Service Management and Orchestration (SMO), so that any mismatch between the intended and observed queuing scheme automatically triggers remediation. To maintain the model's accuracy as the network evolves, continuously capture and label traffic traces – especially during configuration changes or incidents – and feed them into a periodic retraining pipeline.

2) Solution Vendors and OSS/BSS Integrators: Package the raw-feature LSTM as an easily deployable SDK or microservice that exposes Google Remote Procedure Call (gRPC) or Representational State Transfer (REST) endpoints, then embed it into the monitoring and analytics offerings. Further, provide explainable hooks, such as attention maps, SHA values, or equivalent, so that operators can see why a given flow is flagged as PRIO, increasing trust and easing regulatory audits. Finally, implement Continuous Integration (CI) and Continuous Deployment (CD) for model updates, enabling customers to fine-tune their traffic without exposing proprietary data.

3) Research Community (network-ML and Computer vision groups): Address the data-scarcity bottleneck that hampered the CNN and CNN-LSTM experiments by releasing a public, labelled traffic-image dataset and baseline code. Investigate data-efficient vision backbones (ConvNeXt-Tiny, distilled ViTs) and strong regularisation or transfer-learning strategies to determine whether image-based models can close the accuracy gap when more training examples or more powerful architectures are available. In parallel, explore time-series Transformers and temporal convolutional networks as alternatives to LSTMs for raw feature streams.

4) Standard Bodies and Regulators (3GPP, ETSI, FCC/Ofcom): Incorporate ML-based QoS verification metrics into future specifications, such as the 3GPP TS 28 series, so that operators have clear, standardised targets for scheduler identification accuracy and latency. Encourage open-data mandates or shared-capture frameworks that let operators contribute anonymised traces; a larger communal dataset will accelerate research and level the playing field between large and small carriers.

5) Security and Privacy Teams: Treat the scheduler fingerprint as a dual-use capability. While it strengthens internal monitoring, it could also help adversaries infer a network's traffic-shaping strategy. Monitor for abuse – for instance, reconnaissance traffic that systematically probes queuing behaviour – and consider obfuscating or encrypting exported traffic feature logs in less-trusted domains to prevent the leakage of sensitive scheduling information.

In summary, this study recommends starting with raw-feature LSTM deployment to ensure immediate QoS. Parallel efforts should scale datasets, refine image-based models, and embed the technique in commercial and regulatory frameworks, while remaining mindful of its security implications.

6.8.4 Recommendations for Experiment 4

Based on the findings, several recommendations can be made for practitioners and researchers in the field of network performance:

1) Leverage Simple Models First: Given that the RF baseline achieved the highest accuracy with far less complexity, it is recommended to start with simpler models when predicting network delay (or similar metrics). They can serve as a strong baseline and may even suffice for operational use, as was the case here. Only if the baseline accuracy is found inadequate should one consider more complex DL models.

2) Use DL Judiciously: If the problem context suggests significant temporal dependencies (e.g., long-term trends or patterns that simpler models fail to capture), then LSTM or Transformer-based models should be explored. Nonetheless, the added complexity must be justified by rigorous evaluation. This study demonstrated that more complex is not always better – DL models should be employed judiciously, with careful hyperparameter tuning and regular validation against simpler alternatives.

3) Ensure Proper Validation: We recommend adopting robust validation strategies (such as the device-wise split used in this work) for any network prediction modelling.

This approach prevents overly optimistic performance estimates due to data leakage and ensures that the model's success reflects true generalisation to new devices or conditions. Practitioners should mimic this strategy by testing predictive models on network segments or periods not seen during training.

4) Integration into Network Management: The high predictability of network delay using readily available metrics suggests that such models can be integrated into network management systems. For example, network operators could deploy the trained RF model to predict impending latency spikes and proactively adjust configurations or load balancing to mitigate performance issues. Given the model's simplicity and low inference cost, real-time deployment is feasible. Thus, we recommend pilot implementations in controlled environments to evaluate the practical benefits of proactive delay prediction, potentially improving QoS by pre-emptively addressing issues before they impact users.

5) Continued Research on Hybrid Models: Although the hybrid (Linear-LSTM-Transformer) model approach did not outperform in this study, it is a novel attempt worth refining. Researchers are encouraged to further investigate hybrid architectures under various conditions – for instance, with larger datasets or more complex temporal patterns to assess their potential fully. Furthermore, there may be specific contexts (such as multivariate network traffic forecasting or anomaly detection) where combining linear and non-linear sequence components yields improvements. The lessons learned from our results (e.g., the need to manage overfitting and the importance of sufficient training data for complex models) should guide such future explorations.

In summary, this experimental study highlights the importance of conducting thorough empirical comparisons in model selection for network delay prediction. Highlighting the strong performance of a properly tuned traditional model reminds the community that evidence, rather than trends, should drive model choice. Through the above recommendations, we aim to ensure that predictive models deployed in network operations are both effective and grounded in sound experimental insights. The outcomes of this study provide a foundation for developing and evaluating more advanced or specialised predictive modelling techniques in the ongoing quest to enhance network reliability and performance.

6.8.5 Recommendations for Experiment 5

For researchers, these results strongly recommend further investigation into the use of RL for network management, particularly by designing more sophisticated reward functions that incorporate metrics such as jitter or a composite QoS score. Exploring how different RL algorithms (e.g., Actor-Critic methods) and more complex state representations affect convergence and the final policy's effectiveness provides a clear path for future work. For network engineers and operators, the key takeaway is the viability of deploying lightweight, intelligent agents for real-time network monitoring and automated decision-making. Additionally, this approach offers a powerful alternative to static scheduling or routing, as well as manual intervention, enabling the creation of autonomous systems that can dynamically select optimal data paths based on real-time performance data. Adopting

such RL-driven frameworks can lead to more resilient, efficient, and self-optimising networks, ultimately enhancing service reliability and user experience.

Chapter 7: Overall Conclusions

7.1 Introduction

This research study investigated one of the most pressing challenges in deploying 5G and beyond networks: the effective management of QoS for URLLC. The central premise of traditional, static scheduling schemes is fundamentally inadequate for the dynamic, stochastic nature of modern wireless environments, creating a significant barrier to the full realisation of the potential of mission-critical applications. To address this, our research explored a central scientific question: Can a suite of advanced AI techniques – spanning ML, DL, and DRL – be systematically developed and validated to move beyond static rulesets, enabling dynamic, intelligent, and ultimately autonomous network traffic management?

This concluding chapter synthesises the key outcomes of six distinct experimental studies that addressed this question. It distils their collective findings, consolidates their contributions to the field, and reflects on the broader implications for theory and practice. By integrating insights from a hierarchical progression of AI-driven methodologies, this chapter presents a holistic view of the research journey, culminating in a clear set of limitations and promising directions for future inquiry.

7.2 Synthesis of Key Findings: A Narrative from Analysis to Autonomy

The comprehensive body of experimental work charts a deliberate and logical progression, beginning with the characterisation of existing systems, advancing to predictive analysis, and culminating in autonomous control. Several consistent, and at times counterintuitive, patterns emerged across these stages, providing a nuanced understanding of AI's role in network scheduling.

The study began by establishing a quantitative baseline for traditional scheduler trade-offs. The comparative analysis of Proportional Fair (PF) and Round Robin (RR) revealed their complementary strengths. Meanwhile, RR consistently maximised raw throughput and delivery reliability, while PF demonstrated superior performance in maintaining latency fairness and predictability. The application of EGPR modelling was instrumental here, confirming not only the performance characteristics but also highlighting PF's lower variability and, consequently, its higher predictive accuracy. This initial finding underscores that no single traditional scheduler is universally optimal and establishes that “predictability” itself is a crucial, measurable QoS metric.

Building on this, the investigation examined the effectiveness of classifiers in identifying whether scheduling policies leave discernible “fingerprints” on network traffic. The experiments demonstrated resoundingly that they do. Across classification-based comparisons of Priority Queuing (PRIO), Hierarchical Token Bucket (HTB), and Class-Based Queuing (CBQ), tree-based models consistently outperformed SVMs, demonstrating their effectiveness in identifying the active scheduling discipline from operational network data. This confirms that queuing algorithms impart distinctive,

machine-learnable signatures onto traffic flows, opening the door for non-intrusive network monitoring and policy verification.

The investigation into the distinction between DL provided further critical insights into data representation. For classifying schedulers from raw traffic features, LSTM networks achieved near-perfect accuracy, adeptly capturing the sequential nature of network packets. In stark contrast, image-based approaches using CNN and hybrid CNN-LSTM models, while conceptually innovative, proved unstable and less accurate, particularly with limited datasets. This finding delivered a clear verdict on the primacy of sequential feature representations over visual abstractions for this class of network problems.

In the crucial task of predictive QoS forecasting, the findings delivered a surprising and significant insight. While it was hypothesised that complex sequential architectures would excel at delay prediction, a meticulously tuned RF baseline unexpectedly surpassed both LSTM and more advanced hybrid models, including Linear+LSTM+Transformer. This result showed that for next-step latency forecasting in our environment, a rich set of instantaneous network metrics was more informative than the historical temporal dependencies captured by recurrent or attention-based models. This result challenges the common assumption that more complex models are always better and highlights the power of robust feature engineering.

Furthermore, the culmination of this investigative journey was the successful implementation of reinforcement-learning innovation. A Transformer-based DRL agent, trained online in a live 5G WSN environment, demonstrates the feasibility of fully autonomous, adaptive traffic management. The agent consistently achieved 100% scheduling success with ultra-low latency, learning an effective policy to navigate dynamic network conditions without human intervention. This final experiment successfully closed the loop from analysis to control, providing a robust proof of concept for the future of self-driving networks.

7.3 Primary Contributions to Knowledge and Broader Implications

The collective findings from this research study advance the state of the art in network management across several key dimensions. The primary contributions have significant implications for both academic theory and industry practice.

1) A validated Quantitative Evaluation Framework: This study established and validated a comprehensive, multi-stage methodology that combines empirical network measurements with a hierarchy of advanced modelling techniques (ML, DL and DRL). This provides a replicable and robust blueprint for assessing, comparing, and optimising QoS schemes, moving the field beyond isolated, purely theoretical analysis.

2) The Conceptualisation of Predictability as a Core QoS Metric: By demonstrating that the stability and model-based predictability of latency are critical differentiators between schedulers, this study elevates these concepts alongside traditional metrics such as mean delay and throughput. For practitioners, this implies that, for latency-critical

services, selecting a slightly slower but highly predictable scheduler, such as Proportional Fair, may be preferable.

3) The Development of Automated Scheduler Identification Tools: By demonstrating that ML models can accurately detect and classify active QoS policies in real time without requiring intrusive network instrumentation, this work lays the foundation for a new class of automated monitoring, auditing, and verification tools that are fundamental for managing complex, sliced 5G and beyond networks.

4) Actionable Guidance on Feature Engineering and Model Selection: This research study provides clear, empirically backed insights into data representation for network analytics. The demonstrated efficacy of raw time-series features for LSTMs, and the surprising strength of non-sequential metrics for RF in delay prediction, provide pragmatic guidance to researchers and data scientists on where to focus their feature engineering and model selection efforts.

5) A Foundational Proof-of-Concept for Adaptive QoS Management: The successful deployment of the Transformer-based RL agent serves as a landmark demonstration of how RL can dynamically and autonomously optimise scheduling in a live environment. This lays the practical groundwork for the development of fully self-adaptive and self-optimising network control strategies, a cornerstone of 6G vision.

The theoretical implications of this research study lie in enriching queuing theory and network modelling by empirically quantifying the intricate interplay between scheduling schemes and statistical signatures of the traffic they produce. For network operators and engineers, the practical implications are immediate: they can leverage these methodologies to make informed decisions, selecting PF for latency-sensitive applications or deploying intelligent agents for dynamic, automated traffic shaping in complex, heterogeneous 5G service environments.

7.4 Limitations and Future Research Directions

While our research provides a comprehensive investigation, it is vital to acknowledge its limitations, which, in turn, illuminate promising avenues for future research. The primary limitations include the scope of data, as experiments were confined to a specific laboratory hardware configuration and a defined set of traffic loads. The generalisation of these findings to broader network topologies, diverse traffic mixes, and higher-scale 5G slices warrants further validation. The generalisation of models, particularly image-based DL architectures, was found to be sensitive to dataset size, suggesting that their real-world utility depends on access to massive datasets or more advanced training techniques. Finally, the operational constraints of the RL scheduler, which assumed an idealised reward design, must be considered; real-world integration will likely require more complex multi-objective or hierarchical RL formulations to balance competing business and technical goals.

7.5 Cross-cutting limitations

A first limitation is the single-site testbed. All experiments were conducted in a single physical location, using a single WLAN access point, a Linux router, and a modest set of end devices. This constrains the diversity of observed radio conditions, interference patterns, and deployment topologies. The findings cannot be assumed to extrapolate directly to larger multi-cell 5G/6G networks, wired backbones or highly dense IoT deployments.

Second, except for Experiment 1, the study uses a hybrid testbed that combines a controlled WLAN segment with real (live environment) paths over the public Internet (e.g., DNS resolvers and external servers). This is a strength because the models and schedulers experience genuine Internet-scale variability beyond experimental control. At the same time, it is a limitation: only end-to-end RTT and throughput are visible at the measurement point, while the internal causes of delay (congestion, routing changes, peering, anycast effects) along the Internet path remain unobserved. The results reflect one particular WLAN + ISP + set of remote targets, rather than a full heterogeneity of the global Internet.

Third, although several traffic types and protocols are present, the traffic mix remains limited and is primarily driven by our defined experimental usage patterns. Important real-world factors such as user mobility, large numbers of competing flows, or complex industrial workloads are not fully reproduced. This affects how far the results can be generalised to ultra-dense factor floors, vehicular networks or wide-area URLLC deployments.

Fourth, latency is measured as ICMP RTT. Interpreting one-way delay as $RTT/2$ is an approximation, especially over public Internet paths; thus, we can only discuss compatibility with industry latency targets (as in Table 5.8, using Yes*/Partial*), rather than claiming strict URLLC standards compliance, particularly for the 1 ms one-way target.

Finally, DL and RL models were tuned under finite compute and time budgets, using a mixture of manual choices and limited hyperparameter search. The reported configurations are good but not necessarily optimal; different architectures or search strategies could change the relative performance of some models.

1) Experiment-specific limitations

a) Experiment 1 – PF vs RR and EGPR delay prediction: This experiment is based on a purely local WLAN testbed with controlled PF and RR schedulers. Only two scheduling algorithms and one topology are considered, and the EGPR model is trained and tested on data from the same environment. The excellent fit demonstrates that delay is highly predictable in this specific setting; however does not guarantee that the same model would perform equally well under different propagation conditions, multi-cell deployments, or more aggressive congestion and interference.

b) Experiment 2 – PRIO, HTB, CBQ protocol-level classification: The classification models for PRIO, HTB, and CBQ are trained on datasets that include both local WLAN traffic and flows to external Internet servers. While this increases realism, it also introduces uncontrolled variability in path conditions that is not explicitly modelled. Several protocols are underrepresented, leading to class imbalance: the models perform very well on dominant classes (TCP, TLS, ARP) but show poor recall and low AUC for low-volume protocols such as QUIC, NTP, and ICMPv6. The queuing configurations and traffic patterns correspond to a single router and a single ISP; it is unclear how robust these classifiers would be to different shaping policies, link capacities, or backbone paths.

c) Experiment 3 – CNN / CNN-LSTM image-based scheduler identification: The image-based representation of QoS metrics is derived from data collected in the same hybrid WLAN + Internet environment. The mapping from time series to images is precise to the chosen encoding, and the dataset is relatively small by computer vision standards. Some CNN configurations reach 100% validation accuracy and perfect confusion matrices, which, while encouraging, raises the possibility of overfitting or subtle data leakage (e.g., correlations between images from the same capture). There is no evaluation on a completely independent testbed, encoding scheme, or ISP; thus, the generalizability of image-based models remains an open question.

d) Experiment 4 – LSTM and RF QoS-to-delay prediction: The LSTM and RF models achieve extremely high R^2 and very low errors when predicting delay for a held-out device using the PF dataset. This shows that delay is almost deterministic given the selected QoS features in this specific PF-configured WLAN, but it also means that the models are fitted to a single scheduler, topology and local environment without external segments. The feature set omits potentially essential variables such as queue occupancy at intermediate hops, which may matter in more complex radio or multi-hop settings. Both training and validation data come from devices attached to the same AP and router, so generalisation to other schedulers, different WLANs, or scenarios has not been directly tested yet.

e) Experiment 5 – Transformer-based RL scheduling agents: The Experimental setup in Experiment 5 was conducted in a single-site testbed with a limited number of devices, which restricts the generalisability of the findings to larger, more complex 5G network environments. The dataset lacked diversity across geographical locations, traffic patterns, and device heterogeneity, critical factors for assessing the scalability and adaptability of the Transformer-based RL scheduler. Expanding the dataset to include multi-site data and scaling the experimental setup to validate the scheduler's performance in decentralised and large-scale 5G networks would provide a more comprehensive understanding of its capabilities and limitations.

The reward function employed in Experiment 5 was primarily designed to optimise latency reduction and scheduling success ratio, which, while effective for the specific testbed, does not fully address the multi-faceted nature of QoS in 5G networks. Metrics such as jitter, packet loss, energy efficiency, and fairness were not incorporated, potentially limiting the scheduler's ability to balance competing objectives. Future

research should explore more sophisticated reward designs that integrate multiple QoS metrics and adopt multi-objective optimisation frameworks. This would enable the scheduler to dynamically adapt to varying network conditions and ensure long-term stability, fairness, and robustness across diverse traffic scenarios.

7.6 Future Research Directions

Furthermore, building directly upon these foundations and limitations, future research efforts should proceed along several key trajectories. First, the frameworks should be extended to multi-tenant 5G network slices with heterogeneous service-level agreements to test their scalability and robustness. Second, investigating transfer learning approaches and advanced data augmentation techniques for image-based scheduler classification could help overcome the challenge of dataset scarcity. Third, exploring hybrid and multi-agent RL approaches is a vital next step for coordinating scheduling decisions across distributed network nodes in a decentralised manner. Ultimately, integrating explainable AI (XAI) techniques will be crucial for interpreting model decisions, building trust, and ensuring the safe and transparent enforcement of QoS policies in safety- and mission-critical systems.

Furthermore, the findings of this thesis suggest several concrete research directions for extending the work towards prospective 6G, AI-native RANs, and federated learning architectures.

1) AI-native RAN scheduling and control: The PF/RR analysis, EGPR delay model, hybrid Linear+LSTM+Transformer, pure-LSTM, and RF predictors show that, in a controlled PF-configured WLAN, delay can be modelled with very high accuracy from a relatively modest set of QoS features. In an AI-native 6G RAN, such predictors could be embedded as xApps/rApps to provide fast, model-based delay estimates for admission control, slice configuration or scheduler parameter tuning. A natural fit is to re-implement the proposed models in a multi-cell 5G/6G testbed or O-RAN platform, expose richer feature sets (e.g., RSRP/RSRQ, BSR, buffer occupancy, and HARQ statistics), and investigate how model-driven schedulers perform when integrated with standard RRM functions and near-real-time control loops.

2) Multi-objective and cross-layer RL for 6G services: Experiment 5 demonstrated that a Transformer-based RL agent can learn stable low-latency policies in a hybrid WLAN+Internet environment; however, currently, the reward focuses exclusively on RTT. Future work should extend this to multi-objective RL, jointly optimising reliability, throughput, energy and fairness across heterogeneous flows and service classes (URLLC, eMBB, and mMTC). This aligns with 6G vision of service-aware, cross-layer optimisation. Concretely, the reward could be redesigned to incorporate slice-specific KPIs, queue states, and energy budgets, and to be evaluated in scenarios that mimic industrial control, cooperative driving, or XR offloading, ideally on 5G NR/TSN-capable testbeds.

3) Federated and distributed learning across cells and domains: Most models in this research study are trained centrally on data from a single hybrid testbed. In 6G

deployments, however, data will be fragmented across cells, operators, and vertical domains, and privacy or regulatory constraints may prohibit the sharing of raw data. An essential line of extension is to reformulate the QoS-to-delay prediction and scheduler-identification tasks as a federated learning problem, where base stations or edge nodes train local models (e.g., LSTMs, Random Forests, RL agents) and periodically exchange model updates rather than traffic traces. This would allow the proposed architectures to benefit from multi-site diversity (different channels, loads, and applications) while preserving local data ownership and test how robust the models are to non-IID data and partial participation.

4) Data-centric and online adaptation in highly dynamic 6G environments: The experiments highlight that excellent performance is achieved in a single, well-characterised environment; however, also that models may be sensitive to configuration and traffic shifts. Future research should focus on online learning and continual adaptation: for example, continuously updating the delay predictors and RL schedulers as new measurements arrive, detecting distribution shifts (e.g., new applications or routing changes), and triggering controlled retraining. In a 6G context, this could be combined with network analytics to automatically determine when a model is still valid or needs to be re-optimised.

5) From single testbed to large-scale evaluation and standardisation: Finally, our methodological approaches – live hybrid testbed measurements, combined with model-based prediction and RL scheduling – can be extended to multi-site or emulate large-scale scenarios to support standardisation and performance benchmarking. Future work could integrate the proposed models into open 5G/6G platforms and compare them against baseline schedulers and QoS mechanisms under standardised traffic profiles. This would help bridge the gap between proof-of-concept results and deployable AI-native RAN functions for 6G.

7.7 Concluding Remarks

In summary, our research study has systematically demonstrated that the synergetic combination of empirical experimentation with advanced ML, DL, and RL yields a powerful toolkit for understanding, predicting, and ultimately controlling network performance. By framing predictability as a central QoS dimension and leveraging a hierarchy of data-driven models for both passive analysis and active scheduling, this research work paves the way towards more adaptive, intelligent, and efficient networks. The journey from analysis to autonomy presented in this Thesis provides a foundational roadmap for realising the full, transformative potential of 5G and beyond, ensuring that next-generation networks can robustly and reliably meet the demands of our increasingly connected world.

Furthermore, this Thesis argues and demonstrates that a hierarchical suite of AI-driven methodologies – progressing from predictive modelling to autonomous control – can overcome the inherent limitations of traditional network schedulers, enabling the dynamic

and efficient management of QoS required for safety and mission-critical applications in 5G WSNs.

References

- [1] C. Wu, Y. Zhao, S. Xiao, and C. Gao, "Explore deep reinforcement learning to energy-efficient data synchronism in 5G self-powered sensor networks," *IEEE Sensors Journal*, vol. 23, no. 18, pp. 20586–20595, 2023, doi: 10.1109/JSEN.2022.3221983.
- [2] T. Mazhar, M. A. Malik, S. Mohsan., et al., "Quality of service (QoS) performance analysis in traffic engineering model for next-generation wireless sensor networks," *Symmetry*, vol. 15, Art. no. 513, 2023, doi: 10.3390/sym15020513.
- [3] A. A. Nasir, H. D. Tuan, H. Q. Ngo, T. Q. Duong, and H. V. Poor, "Cell-free massive MIMO in the short blocklength regime for URLLC," *IEEE Trans. Wirel. Commun.* vol. 20, no. 9, pp. 5861–5871, 2021.
- [4] R. K. Gupta, S. Kumar, and R. Mira, "Resource allocation for UAV-assisted 5G mMTC slicing networks using deep reinforcement learning," *Telecommunications Systems*, vol. 82, no. 1, pp. 141–159, 2023, doi: 10.1007/s11235-022-00974-3.
- [5] H. Ren, C. Pan, Y. Deng, M. El Kashlan, and A. Nallanathan, "Joint pilot and payload power allocation for massive-MIMO-enabled URLLC IIoT networks," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 5, pp. 816–830, 2020.
- [6] D. Gustin, *et al.*, "Outdoor Field Test of 5G-based V2X Communication for Real-Time Monitoring and Remote Control of a Monorail Vehicle," in *Proc. of IEEE INDIN*, Lemgo, Germany, 2023.
- [7] M. C. Lucas-Estañ, et al., "An Analytical Latency Model and Evaluation of the Capacity of 5G NR to Support V2X Services using V2N2V Communications," *IEEE Trans. Veh. Technol.*, vol. 72, no. 2, pp. 2293–2306, Feb. 2023.
- [8] Y. Zhu, L. Sun, J. Wang, R. Huang, and X. Jia, "Deep reinforcement learning-based joint scheduling of 5G and TSN in industrial networks," *Electronics*, vol. 12, no. 12, p. 2686, 2023, doi: 10.3390/electronics12122686.
- [9] M. Fozi, A. Sharafat, and M. Bennis, "Fast MIMO beamforming via deep reinforcement learning for high mobility mmWave connectivity," *IEEE Journal on Selected Areas in Communications*, vol. 40, pp. 127–142, 2022, doi: 10.1109/JSAC.2021.3126056.
- [10] F. Zhu, B. Wang, Z. Yang, et al., "Robust millimeter beamforming via self-supervised hybrid deep learning," in 2023 31st European Signal Processing Conference (EUSIPCO), pp. 915–919, 2023, doi: 10.23919/EUSIPCO58844.2023.10289989.
- [11] Z. Ding and H. V. Poor, "Joint beam management and power allocation in THz-NOMA networks," *IEEE Trans. Commun.*, vol. 71, no. 4, pp. 2059–2073, Apr. 2023, doi: 10.1109/TCOMM.2023.3240432.
- [12] S. Jing and H. Wang, "Design and Implementation of a 5G Network Architecture Based on Software Defined Network," in 2023 IEEE 3rd International Conference on

Electronic Technology, Communication and Information (ICETCI), 2023, pp. 1444–1449. doi: 10.1109/ICETCI57876.2023.10176812.

[13] M. C. Lucas-Estañ *et al.*, "5G Network Architecture and Configuration Choices to Support Teleoperated Driving at Scale," in 2024 IEEE 100th Vehicular Technology Conference (VTC2024-Fall), 2024, pp. 1-6, doi: 10.1109/VTC2024-Fall63153.2024.10758026.

[14] V. N. Sathi and C. S. R. Murthy, "A novel approach to slice selection in private 5G-and-beyond networks for Industrial IoT," in Proc. IEEE Wireless Communications and Networking Conference (WCNC), Dubai, United Arab Emirates, Apr. 2024, pp. 1–6, doi: 10.1109/WCNC57260.2024.10571072.

[15] Y. Cai, P. Cheng, Z. Chen, M. Ding, B. Vucetic, and Y. Li, "Deep reinforcement learning for online resource allocation in network slicing," IEEE Transactions on Mobile Computing, vol. 23, no. 6, pp. 7099–7116, 2024, doi: 10.1109/TMC.2023.3328950.

[16] H. Wang, Y. Bai, and X. Xie, "Deep reinforcement learning based resource allocation in delay-tolerance-aware 5G industrial IoT systems," IEEE Transactions on Communications, vol. 72, no. 1, pp. 209–221, 2024, doi: 10.1109/TCOMM.2023.3322736.

[17] Y. Qian, G. Liu, and H. Shi, "Deep learning method to optimize the quality of service in UAV-aided 5G/6G networks," "Wireless Networks", 2023, doi: 10.1007/s11276-023-03562-8.

[18] C. Hu *et al.*, "Digital twin-assisted real-time traffic data prediction method for 5G-enabled internet of vehicles," "IEEE Transactions on Industrial Informatics", vol. 18, no. 4, pp. 2811–2819, 2022.

[19] E. Aloupogianni *et al.*, "Network slicing for beyond 5G networks using machine learning," in Proc. 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), pp. 197–200, 2024, doi: 10.1109/CCGridW63211.2024.00031.

[20] K. Vijayakumar, P. Thirumaraiselvan, B. Sivakumar, and P. Seenuvasan, "Revolutionizing Data Transmission - A Comprehensive Review of Deep Learning-Based Routing Mechanism for 5G Wireless Sensor Networks," in *Proc. 2024 10th Int. Conf. Commun. Signal Process. (ICCSP)*, Melmaruvathur, India, 2024, pp. 565–570. doi: [10.1109/ICCSP60870.2024.10544339](https://doi.org/10.1109/ICCSP60870.2024.10544339)

[21] G. Caso, M. Rajiullah, K. Kousias, U. Ali, L. Nardis, A. Brunström, Ö. Alay, M. Neri, and M. Di Benedetto, "An Initial Look into the Performance Evolution of 5G Non-Standalone Networks," in 2023 7th Network Traffic Measurement and Analysis Conference (TMA), 2023, pp. 1-4. doi: 10.23919/TMA58422.2023.10199039.

[22] N. H. Mahmood, G. Berardinelli, E. J. Khatib, R. Hashemi, C. de Lima, and M. Latva-aho, "A functional architecture for 6G special-purpose Industrial IoT

networks,” *IEEE Trans. Ind. Informatics*, vol. 19, no. 3, pp. 2530–2540, Mar. 2023, doi: 10.1109/TII.2022.3182988.

[23] H. Wang, Y. Bai, and X. Xie, “Dynamic resource allocation for 5G-enabled industrial Internet of Things system with delay tolerance,” in *2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall)*, London, United Kingdom, pp. 1–6, 2022, doi: 10.1109/VTC2022-Fall57202.2022.10012838.

[24] T. Do-Duy, D. V. Huynh, O. A. Dobre, B. Canberk, and T. Q. Duong, “Digital twin-aided intelligent offloading with edge selection in mobile edge computing,” *IEEE Wireless Communications Letters*, vol. 11, no. 4, pp. 806–810, 2022.

[25] H. Kim, Y. J. Kim, and W. T. Kim, “Deep reinforcement learning-based adaptive scheduling for wireless time-sensitive networking,” *Sensors*, vol. 24, no. 5281, pp. 1–21, 2024, doi: 10.3390/s24165281.

[26] E. R. Gallup, J. F. Tuttle, J. Immonen, B. Billings, and K. M. Powell, “Transformer neural networks with spatiotemporal attention for predictive control and optimization of industrial processes,” in *2024 American Control Conference (ACC)*, Toronto, ON, Canada, pp. 382–387, 2024, doi: 10.23919/ACC60939.2024.10644816.

[27] F. Salahdine, T. Han, and N. Zhang, “5G, 6G, and Beyond: Recent advances and future challenges,” *Annals of Telecommunications*, vol. 78, no. 9–10, pp. 525–549, Oct. 2023, doi: 10.1007/s12243-022-00938-3.

[28] A. Alhammadi, I. Shayea, A. A. El-Saleh, M. H. Azmi, Z. H. Ismail, L. Kouhalvandi, and S. A. Saad, “Artificial intelligence in 6G wireless networks: opportunities, applications, and challenges,” *Int. J. Intell. Syst.*, vol. 2024, Art. no. 8845070, Mar. 2024, doi: 10.1155/2024/8845070.

[29] J. Deepthy, N. Radhika, and A. Denny, “Regularized hybrid deep learning for DDoS attack prediction in software defined Internet of Things (SD-IoT),” in *Proceedings of the 2024 Third International Conference on Power, Control and Computing Technologies (ICPC2T)*, pp. 83–87, 2024, doi: 10.1109/ICPC2T60072.2024.10474620.

[30] V. Cislighi, C. Quadri, V. Mancuso, and M. A. Marsan, “Simulation of Tele-Operated Driving over 5G Using CARLA and OMNET++,” in *Proc. of IEEE VNC*, Istanbul, Turkiye, 2023, pp. 81–88.

[31] L. Sun, J. Liang, C. Zhang, D. Wu, and Y. Zhang, “Meta-transfer metric learning for time series classification in 6G-supported intelligent transportation systems,” *IEEE Trans. Intell. Transp. Syst.*, vol. 25, no. 3, pp. 2757–2767, 2024, doi: 10.1109/TITS.2023.3250962.

[32] M. Kim, Y. Do, J. Kim, and J. Jeon, “Asynchronous traffic handling in time-sensitive in-vehicle network,” in *2023 17th Int. Conf. Ubiquitous Inf. Mgmt. and Commun. (IMCOM)*, Seoul, Korea, Rep. of, pp. 1–6.

- [33] Y. Fu and X. Wang, "Traffic prediction-enabled energy-efficiency dynamic computing resource allocation in CRAN based on deep learning," *IEEE Open Journal of the Communications Society*, vol. 3, pp. 159–175, 2022, doi: 10.1109/OJCOMS.2022.3146886.
- [34] J. Huang et al., "Reinforcement learning based resource management for 6G-enabled mIoT with hypergraph interference model," *IEEE Transactions on Communications*, vol. 72, no. 7, pp. 4179–4192, 2024, doi: 10.1109/TCOMM.2024.3372892.
- [35] G. O. Boateng, G. Sun, D. A. Mensah, D. M. Doe, R. Ou, and G. Liu, "Consortium blockchain-based spectrum trading for network slicing in 5G RAN: A multi-agent deep reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 22, no. 10, pp. 5801–5815, 2023, doi: 10.1109/TMC.2022.3190449.
- [36] M. Eskandari and A. V. Savkin, "Deep-reinforcement-learning-based joint 3-D navigation and phase-shift control for mobile Internet of Vehicles assisted by RIS-equipped UAVs," *IEEE Internet of Things Journal*, vol. 10, no. 20, pp. 18054–18066, 2023, doi: 10.1109/JIOT.2023.3277598.
- [37] P. Lin, K. Ye, Y. Hu, Y. Lin, and C. Z. Xu, "A novel multimodal deep learning framework for encrypted traffic classification," *IEEE/ACM Transactions on Networking*, vol. 31, no. 3, pp. 1369–1382, 2023, doi: 10.1109/TNET.2022.3215507.
- [38] M. A. Habib et al., "Transformer-based wireless traffic prediction and network optimization in O-RAN," in *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*, Denver, CO, USA, pp. 1–6, 2024, doi: 10.1109/ICCWorkshops59551.2024.10615438.
- [39] S. Kumar, S. Chattopadhyay, and C. Adak, "TPMCF: Temporal QoS prediction using multi-source collaborative features," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3945–3955, 2024, doi: 10.1109/TNSM.2024.3395428.
- [40] L. Magoula, N. Koursioupas, S. Barmounakis, et al., "A deep learning approach for distributed QoS prediction in beyond 5G networks," in *2022 IEEE 33rd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pp. 1412, 2022, doi: 10.1109/PIMRC54779.2022.9977782.
- [41] H. Dutta, A. K. Bhuyan and S. Biswas, "Contextual deep reinforcement learning for flow and energy management in wireless sensor and IoT networks," *IEEE Transactions on Green Communications and Networking*, vol. 8, no. 3, pp. 1233–1244, Sept. 2024, doi: 10.1109/TGCN.2024.3358230.
- [42] L. Chen and B. Li, "Dynamic network scheduling in TSN networks based on improved reinforcement learning," *2023 IEEE 6th International Conference on Electronics and Communication Engineering (ICECE)*, Xi'an, China, 2023, pp. 59–62, doi: 10.1109/ICECE59822.2023.10462309.
- [43] B. Coll-Perales, et al., "End-to-End V2X Latency Modeling and Analysis in 5G Networks," *IEEE Trans. Veh. Technol.*, vol. 72, no. 4, pp. 5094–5109, Apr. 2023.

- [44] R. Prasad and A. Sunny, “QoS-aware scheduling in 5G wireless base stations,” arXiv, vol. abs/2310.11206, Oct. 17, 2023, doi: 10.48550/arXiv.2310.11206.
- [45] J. Wang, D. Feng, S. Zhang, A. Liu, and X.-G. Xia, “Joint computation offloading and resource allocation for MEC-enabled IoT systems with imperfect CSI,” *IEEE Internet of Things Journal*, vol. 8, no. 5, pp. 3462–3475, 2021.
- [46] W. Chen, D. Liu, and Y. Bai, “B5G/6G URLLC latency reduction method for multisensor Industrial Internet of Things,” *IEEE Internet of Things Journal*, vol. 11, pp. 11444–11459, 2024, doi: 10.1109/JIOT.2023.3330814.
- [47] G. Pocovi, B. Soret, K. Pedersen, and P. Mogensen, “MAC layer enhancements for ultra-reliable low-latency communications in cellular networks,” in *Proc. 2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2017, pp. 1005–1010, doi: 10.1109/ICCW.2017.7962790.
- [48] Z. Zhang, Y. Gao, Y. Liu, and Z. Li, “Performance evaluation of shortened transmission time interval in LTE networks,” in *Proc. 2018 IEEE Wireless Communications and Networking Conference (WCNC)*, 2018, pp. 1–5, doi: 10.1109/WCNC.2018.8377098.
- [49] A. Zayat, M. A. Hasabelnaby, M. Obeed, and A. Chaaban, “Transformer masked autoencoders for next-generation wireless communications: Architecture and opportunities,” *IEEE Communications Magazine*, vol. 62, no. 7, pp. 88–94, 2024, doi: 10.1109/MCOM.002.2300257.
- [50] M. Kothari and S. Kurumbanshi, “DQN based distributed cooperative spectrum sensing for multiband multiuser CRN,” in 2024 2nd International Conference on Device Intelligence, Computing and Communication Technologies (DICCT), Dehradun, India, pp. 633–638, 2024, doi: 10.1109/DICCT61038.2024.10532974.
- [51] H. Wang, L. Cui, E. Wang, and J. Liu, “A Differentially Private Approach for Budgeted Combinatorial Multi-Armed Bandits,” *IEEE Transactions on Dependable and Secure Computing*, vol. 22, pp. 424–439, 2025, doi: 10.1109/TDSC.2024.3401836.
- [52] Y. Lan, X. Xu, Q. Fang, and J. Hao, “Sample efficient deep reinforcement learning with online state abstraction and causal transformer model prediction,” *IEEE Transactions on Neural Networks and Learning Systems*, 2023, doi: 10.1109/TNNLS.2023.3296642.
- [53] L. Qin and Z. Xueping, “A traffic flow prediction framework based on deep learning and particle swarm optimization,” in 2023 IEEE International Conference on Sensors, Electronics and Computer Engineering (ICSECE), pp. 1467, 2023, doi: 10.1109/ICSECE58870.2023.10263506.
- [54] X. Wang, H. Yao, T. Mai, S. Guo, and Y. Liu, “Reinforcement learning-based particle swarm optimization for end-to-end traffic scheduling in TSN-5G networks,” *IEEE/ACM Transactions on Networking*, vol. 31, no. 6, pp. 3254–3268, 2023, doi: 10.1109/TNET.2023.3276363.
- [55] S. Lipsa, R. Dash, N. Ivković, and K. Cengiz, “Task scheduling in cloud computing: A priority-based heuristic approach,” *IEEE Access*, vol. 11, pp. 27111–27126, 2023, doi: 10.1109/ACCESS.2023.3255781.

- [56] B. Hazarika, K. Singh, S. K. Singh, C. Pan, and T. Q. Duong, "Asynchronous federated learning-based resource management in URLLC-IoV networks," in Proc. IEEE GLOBECOM, 2023, pp. 6874–6879.
- [57] T. Zhang, K.-Y. Lam, and J. Zhao, "Joint device scheduling and bandwidth allocation for federated learning over wireless networks," IEEE Trans. Wireless Commun., early access, 2023, doi: 10.1109/TWC.2023.3291701.
- [58] J. Xu and H. Wang, "Client selection and bandwidth allocation in wireless federated learning networks: A long-term perspective," IEEE Trans. Wireless Commun., vol. 20, no. 2, pp. 1188–1200, Feb. 2021, doi: 10.1109/TWC.2020.3031503.
- [59] W. Shi, S. Zhou, Z. Niu, M. Jiang, and L. Geng, "Joint device scheduling and resource allocation for latency-constrained wireless federated learning," IEEE Trans. Wireless Commun., vol. 20, no. 1, pp. 453–467, Jan. 2021, doi: 10.1109/TWC.2020.3025446.
- [60] Y. Liu, H. Zhou, Y. Deng, and A. Nallanathan, "Channel access optimization in unlicensed spectrum for downlink URLLC: Centralized and federated DRL approaches," IEEE Journal on Selected Areas in Communications, vol. 41, no. 7, pp. 2208–2222, Jul. 2023, doi: 10.1109/JSAC.2023.3280982.
- [61] J. Zhou, D. Ding, Z. Wu, and Y. Xiu, "Spatial context-aware time-series forecasting for QoS prediction," IEEE Transactions on Network and Service Management, vol. 20, no. 2, pp. 918–931, 2023, doi: 10.1109/TNSM.2023.3250512.
- [62] S. T. Nabi, M. R. Islam, M. G. R. Alam, M. M. Hassan, S. A. Alqahtani, G. Aloï, and G. Fortino, "Deep learning based fusion model for multivariate LTE traffic forecasting and optimized parameter estimation," IEEE Access, vol. 11, pp. 14533–14547, 2023, doi: 10.1109/ACCESS.2023.3242861.
- [63] S. Mahajan, R. Harikrishnan, and K. Kotecha, "Prediction of network traffic in wireless mesh networks using hybrid deep learning model," IEEE Access, vol. 10, pp. 7003–7015, 2022, doi: 10.1109/ACCESS.2022.3140646.
- [64] R. Dangi and P. Lalwani, "A novel hybrid deep learning approach for 5G network traffic control and forecasting," Concurrency and Computation: Practice and Experience, vol. 35, 2022, doi: 10.1002/CPE.7596.
- [65] M. Alsenwi, N. H. Tran, M. Bennis, S. R. Pandey, A. K. Bairagi, and C. S. Hong, "Intelligent resource slicing for eMBB and URLLC coexistence in 5G and beyond: A deep reinforcement learning based approach," IEEE Trans. Wireless Commun., vol. 20, no. 7, pp. 4585–4600, Jul. 2021, doi: 10.1109/TWC.2021.3060514.
- [66] L. Chen et al., "Transformer-based imitative reinforcement learning for multirobot path planning," IEEE Trans. Ind. Informatics, vol. 19, no. 10, pp. 10233–10243, 2023, doi: 10.1109/TII.2023.3240585.

- [67] R. Pamba, R. Bhandari, A. Asha, R. Neware, and A. Bist, "Novel deep learning approach to support optimal resource allocation in 5G environment," *Journal of Mobile Multimedia*, vol. 19, pp. 739–764, 2023, doi: 10.13052/JMM1550-4646.1935.
- [68] K. Zhang, J. Cao, and Y. Zhang, "Adaptive digital twin and multiagent deep reinforcement learning for vehicular edge computing and networks," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 1405–1413, 2022.
- [69] Md. A. Khudhus, "5G Network Optimization using Machine Learning," *Int. J. Res. Appl. Sci. Eng. Technol.*, 2025. doi: [10.22214/ijraset.2025.66782](https://doi.org/10.22214/ijraset.2025.66782)
- [70] B. L. Kundaliya, J. Patel, S. Hadia, and U. Patel, "Machine Learning in Wireless Sensor Networks," *Wireless Personal Communications*, vol. 137, pp. 367–397, 2024. doi: [10.1007/s11277-024-11413-8](https://doi.org/10.1007/s11277-024-11413-8)
- [71] S. O. Olatinwo and T.-H. Joubert, "Deep learning for resource management in Internet of Things networks: A bibliometric analysis and comprehensive review," *IEEE Access*, vol. 10, pp. 94691–94717, 2022, doi: 10.1109/ACCESS.2022.3195898.
- [72] G. Pang, L. You, and L. Fu, "An efficient DRL-based link adaptation for cellular networks with low overhead," in *Proc. 2024 IEEE Wireless Communications and Networking Conference (WCNC)*, Dubai, United Arab Emirates, 2024, pp. 1–6, doi: 10.1109/WCNC57260.2024.10570648.
- [73] X. Ye, Y. Yu, and L. Fu, "Deep reinforcement learning based link adaptation technique for LTE/NR systems," *IEEE Trans. Veh. Technol.*, vol. 72, no. 6, pp. 7364–7379, Jun. 2023, doi: 10.1109/TVT.2023.3236791.
- [74] T. Lee, H. Kim, and J. G. Andrews, "Deep learning-based mmWave beam alignment with only pilot channel measurements," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Denver, CO, USA, 2024, pp. 5305–5310, doi: 10.1109/ICC51166.2024.10622318.
- [75] D. Xie, C. Liu, W. Wang, X. Hu, and M. Peng, "Deep residual learning for channel estimation in UAV mmWave systems: A model-driven approach," in *Proc. 2024 IEEE/CIC Int. Conf. Commun. China (ICCC)*, Hangzhou, China, 2024, pp. 1009–1014, doi: 10.1109/ICCC62479.2024.10681783.
- [76] S. M. M. H. Daneshvar and S. M. Mazinani, "Training a graph neural network to solve URLLC and eMBB coexisting in 5G networks," *Computer Communications*, vol. 225, pp. 171–184, 2024, doi: 10.1016/j.comcom.2024.07.008.
- [77] W. Jing, J. Wang, J. Ren, Z. Lu, H. Shao, and X. Wen, "Radio resource allocation optimization for delay-sensitive services based on graph neural networks and offline dataset," *IEEE Trans. Veh. Technol.*, vol. 74, no. 3, pp. 4510–4525, Mar. 2025, doi: 10.1109/TVT.2024.3497603.
- [78] H. Yang et al., "Knowledge-driven resource allocation for wireless networks: A WMMSE unrolled graph neural network approach," *IEEE Internet of Things Journal*, vol. 11, no. 10, pp. 18902–18916, May 15, 2024, doi: 10.1109/JIOT.2024.3368516.

- [79] Y. Huang, S. Li, C. Li, Y. T. Hou, and W. Lou, "A deep-reinforcement-learning-based approach to dynamic eMBB/URLLC multiplexing in 5G NR," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6439–6456, Jul. 2020, doi: 10.1109/JIOT.2020.2978692.
- [80] F. Lotfi and F. Afghah, "Meta reinforcement learning approach for adaptive resource optimization in O-RAN," in *2025 IEEE Wireless Communications and Networking Conference (WCNC)*, 2025, pp. 1–6, doi: 10.1109/WCNC61545.2025.10978365.
- [81] Y. Lin, Y. Hsu, Y. Chen, Y. Chang, J. Fang, and C. Huang, "Meta-learning traffic pattern adaptation for DRL-based radio resource management," in *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2024, pp. 227–232, doi: 10.1109/ICCWorkshops59551.2024.10615690.
- [82] R. Sohaib, O. Onireti, Y. Sambo, R. Swash, and M. Imran, "Energy efficient resource allocation framework based on dynamic meta-transfer learning for V2X communications," *IEEE Trans. Netw. Serv. Manag.*, vol. 21, pp. 4343–4356, 2024, doi: 10.1109/TNSM.2024.3400605.
- [83] S. Aravindan and W. Lee, "State-aware variational Thompson sampling for deep Q-networks," *arXiv:2102.03719*, Feb. 2021, doi: 10.5555/3463952.3463973.
- [84] H. Mukhtar, A. Afzal, S. Alahmari, and S. Yonbawi, "CCGN: Centralized collaborative graphical transformer multi-agent reinforcement learning for multi-intersection signal free-corridor," *Neural Networks*, vol. 166, pp. 396–409, 2023, doi: 10.1016/j.neunet.2023.07.027.
- [85] M. Kothari and S. Kurumbanshi, "DQN based distributed cooperative spectrum sensing for multiband multiuser CRN," in *2024 2nd International Conference on Device Intelligence, Computing and Communication Technologies (DICCT)*, Dehradun, India, 2024, pp. 633–638, doi: 10.1109/DICCT61038.2024.10532974.
- [86] Z. Yang, J. Zheng, and G. Chen, "Reinfier and Reintrainer: Verification and interpretation-driven safe deep reinforcement learning frameworks," *arXiv:2410.15127*, Oct. 2024, doi: 10.48550/arXiv.2410.15127.
- [87] D. Gross and H. Spieker, "Co-activation graph analysis of safety-verified and explainable deep reinforcement learning policies," *arXiv:2501.03142*, Jan. 2025, pp. 611–621, doi: 10.48550/arXiv.2501.03142.
- [88] A. Sanyoto, D. Perdana, and Y. Bisono, "Performance evaluation of Round Robin and Proportional Fair scheduling algorithms on 5G millimeter wave for node density scenarios," *Proc. (unpublished)*, 2019.
- [89] A. El-Shinnawy, A. Badawi, and A. Nassar, "Two-step scheduling algorithm for IEEE 802.16 wireless networks," in *2010 IEEE 12th International Conference on Communication Technology (ICCT)*, 2010, pp. 620–623, doi: 10.1109/ICCT.2010.5688935.

- [90] B. Li, A. Eryilmaz, and R. Srikant, “Emulating Round-Robin in wireless networks,” in Proc. 18th ACM Int. Symp. Mobile Ad Hoc Networking and Computing (MobiHoc), 2017, pp. –, doi: 10.1145/3084041.3084052.
- [91] H. Ferng, C. Lee, J. Huang, and G. Chiu, “Designing a fair scheduling mechanism for IEEE 802.11 wireless LANs,” IEEE Commun. Lett., vol. 9, no. 4, pp. 301–303, Apr. 2005, doi: 10.1109/LCOMM.2005.04003.
- [92] MathWorks, “Deep Learning in MATLAB,” [Online]. Available: <https://www.mathworks.com/help/deeplearning/ug/deep-learning-in-matlab.html>. [Accessed: May 21, 2024].
- [93] MathWorks, “Reinforcement Learning Toolbox” [Online]. Available: <https://www.mathworks.com/products/reinforcement-learning.html>. [Accessed: May 21, 2024].
- [94] MathWorks, “Machine Learning,” [Online]. Available: <https://www.mathworks.com/solutions/machine-learning.html>. [Accessed: May 21, 2024].
- [95] N. Zhang, J. Xiong, J. Zhong, and K. Leatham, “Gaussian process regression method for classification for high-dimensional data with limited samples,” in 2018 Eighth International Conference on Information Science and Technology (ICIST), 2018, pp. 358–363, doi: 10.1109/ICIST.2018.8426077.
- [96] Y. Lee and J. Park, “Model selection algorithm in Gaussian process regression for computer experiments,” Commun. Stat. Appl. Methods, vol. 24, no. 4, pp. 383–396, 2017, doi: 10.5351/CSAM.2017.24.4.383.
- [97] MathWorks, “Gaussian Process Regression,” [Online]. Available: <https://www.mathworks.com/help/stats/gaussian-process-regression.html>. [Accessed: May 21, 2024].
- [98] B. Sun and B. Park, “Route choice modeling with Support Vector Machine,” Transportation Research Procedia, vol. 25, pp. 1806–1814, 2017, doi: 10.1016/J.TRPRO.2017.05.151.
- [99] J. Zou, C. Yuan, X. Zhang, G. Zou, and A. Wan, “Model averaging for Support Vector Machine by cross-validation,” 2021.
- [100] MathWorks, “Support Vector Machine,” [Online]. Available: <https://www.mathworks.com/help/stats/support-vector-machine.html>. [Accessed: May 21, 2024].
- [101] MathWorks, “TreeBagger (Random Forest) Ensemble Learning,” [Online]. Available: <https://www.mathworks.com/help/stats/treebagger.html>. [Accessed: May 21, 2024].
- [102] MathWorks, “Feature Selection and Extraction,” [Online]. Available: <https://www.mathworks.com/help/stats/feature-selection.html>. [Accessed: May 21, 2024].

- [103] MathWorks, “Feature Extraction and Generation,” [Online]. Available: <https://www.mathworks.com/help/stats/feature-extraction.html>. [Accessed: May 21, 2024].
- [104] MathWorks, “Classification Learner App,” [Online]. Available: <https://www.mathworks.com/help/stats/classification-learner-app.html>. [Accessed: May 21, 2024].
- [105] MathWorks, “fitctree (Classification Tree),” [Online]. Available: <https://www.mathworks.com/help/stats/fitctree.html>. [Accessed: May 21, 2024].
- [106] MathWorks, “fitcsvm (Support Vector Machine for Classification),” [Online]. Available: <https://www.mathworks.com/help/stats/fitcsvm.html>. [Accessed: May 21, 2024].
- [107] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [108] F. Chollet, *Deep Learning with Python*, Manning Publications, 2017.
- [109] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems*, vol. 32, pp. 8024–8035, 2019.
- [110] B. Adhikari, M. Jaseemuddin, and A. Anpalagan, “Resource Allocation for Co-Existence of eMBB and URLLC Services in 6G Wireless Networks: A Survey,” *IEEE Access*, vol. 12, pp. 552–581, 2024, doi: 10.1109/ACCESS.2023.3343250.

Appendices

Appendix A: Implementation of ML Algorithms for Classification in MATLAB

This study utilised MATLAB's Classification Learner app to implement machine-learning models for classification tasks. The app offers an interactive environment for training, validating, and tuning classification models using algorithms such as Decision Trees, SVMs and Ensembles. The implementation process involved key steps to develop models capable of classifying and predicting scheduling algorithms and network performance metrics. For instance, Decision Tree models were designed to classify scheduling algorithms, such as Proportional Fair (PF) and Round Robin, based on selected input features [104], [105], and [106].

Below are described the key steps involved in implementing classification models using this application:

1) Launching the classification learner app:

- Open the MATLAB software package and navigate to the “APPS” tab.
- Under “ML and DL,” select “Classification Learner”

2) Loading data into the App:

- Click “New Session” and import data from the workspace or a file.
- Select the dataset, specify the response variable (the target classification), and choose predictor variables (features).
- Ensure that the data is appropriately pre-processed and formatted for classification tasks.

3) Selecting and training classification models:

- In the “Model Type” section, select from various classifiers, including decision trees, support vector machines, logistic regression, and ensemble methods.
- For a quick start, use the option “All Quick-To-Train” to train multiple models simultaneously and compare their performance.
- Click “Train” to initiate the training process.

4) Evaluating model performance:

- After training, assess the model accuracy and other metrics displayed in the app.
- Use visualisation tools such as confusion matrices, ROC curves, and scatter plots to understand model performance and identify potential areas for improvement.

5) Exporting and utilising the trained model:

- When satisfied with a model, export it to the MATLAB workspace by clicking “Export Model.”

- The exported model can be used to predict new data or integrated into larger applications.
- The app also generates MATLAB code for the trained model, automating the classification process programmatically.

The methodological approach for designing and implementing machine-learning models for classification or prediction remains broadly consistent, whether the input data is synthetic or derived from a live network. The core processes – data preprocessing, feature engineering, model selection, training, and evaluation – are applicable in both scenarios. The crucial distinction lies mainly in the nature and origin of the datasets themselves, not in how one handles them algorithmically. Regardless of the source, one first needs to explore their datasets, handle missing data, select which attributes can help train a model, and use that information to decide which classification or prediction model is best. After those data manipulation phases, the core steps for training a model are the same for synthetic and real-world data [104], [105], and [106].

However, algorithmically generated synthetic data typically exhibits characteristics that differ from those of real-world data. Synthetic data may be intentionally structured and balanced, potentially lacking the noise, variability, and inherent biases found in live network data. On the other hand, data captured from a live network is often incomplete and may contain outliers or imbalanced class distributions reflecting real-world conditions. This disparity means that models trained on synthetic data might not generalise effectively to the complexities of live network environments, potentially underperforming on data containing unknown or unexpected variations, requiring more evaluation after training. The emphasis shifts from the algorithm process to carefully considering data characteristics, necessitating robust preprocessing, feature engineering strategies, and, in some cases, an iterative retraining approach with data augmentation for real-world data [104], [105], and [106].

Appendix B: Implementation of ML algorithms for Regression in MATLAB

This study used MATLAB's Regression Learner App to implement regression models for predicting key network performance indicators and QoS mechanisms (scheduling disciplines) [105], [106]. This application, integrated into MATLAB, provides an interactive environment for training, validating, and tuning regression models. Below are the steps used by this study to implement regression algorithms using this App effectively:

1) Launch the Regression Learner App:

- In the MATLAB software package, navigate to the Apps tab.
- Under the ML and DL section, click on Regression Learner.

2) Import Data:

- Click "New Session," then select "From Workspace" or "File" to load the dataset.

- In the dialogue box, choose the dataset, and the App will automatically identify predictors and response variables; however, one can adjust these selections as needed.
- Specify a validation scheme to assess the model's performance, such as cross-validation or holdout validation.

3) Explore and Pre-process Data:

- Utilise the Plots section to visualise data distributions and relationships between predictors and response variables.
- If necessary, perform feature selection or transformation:
 - ✓ Click Feature Selection to include or exclude specific predictors based on their importance.
 - ✓ Enable Principal Component Analysis (PCA) in the Options section to reduce dimensionality.

4) Select and Train Models:

- In the Models section, choose the type of regression model to train. Options include:
 - ✓ Linear Regression
 - ✓ Regression Trees
 - ✓ Support Vector Machines (SVMs)
 - ✓ Ensemble of Regression Trees
 - ✓ Neural Network Regression Models
- For a comprehensive comparison, multiple models can be trained simultaneously.
- Click Train to initiate the training process. MATLAB has a parallel computing toolbox, and the app can train models in parallel to expedite this process.

5) Evaluate Model Performance:

- After training, assess model performance using various plots and metrics:
 - ✓ Response Plot: Visualise predicted vs. actual responses
 - ✓ Residual Plots: Examine the residuals to identify patterns that may indicate model inadequacies.
 - ✓ Predicted vs. Actual Plot: Assess how well the model's predictions align with actual values.
- Review performance metrics such as RMSE, MAE, and R-squared to determine model accuracy.

6) Tune Hyperparameters:

Improving model performance in the Regression Learner App entails tuning specific hyperparameters for each regression model type. This study experimented with different hyperparameters for Regression Trees, SVMs, Gaussian Process Regression (GPR) models, and Random Forests (a type of ensemble method):

a) Regression Trees

- **Minimum leaf size:** Determines the minimum number of observations per leaf. Smaller sizes can capture more detail; however, it may lead to overfitting, while larger sizes promote a simpler model.
- **Maximum splits:** Sets the maximum number of decision splits (or branching points) in the tree. Increasing this value allows the tree to model more complex relationships, but can also increase the risk of overfitting.
- **Surrogate decision splits:** Enable surrogate splits to handle missing data by filling in alternative predictors that can mimic the decisions of the primary split predictor.

b) Support Vector Machines

- **Kernel function:** Specify the function to transform the data into a higher-dimensional space. Options include linear, Gaussian (RBF), and polynomials. Additionally, the choice of kernel can significantly impact model performance.
- **Box Constraint (C):** Controls the trade-off between achieving a low error on the training data and minimising the model complexity. A higher value emphasises that the model fits the training data more precisely, leading to a more complex model that could overfit the training data and perform so well on new, unseen data.
- **Epsilon (ϵ):** Defines the margin of tolerance, where no penalty is given to errors. Adjusting this hyperparameter affects the number of support vectors and the model's sensitivity to errors.
- **Kernel scale:** Scales the input data for the kernel function, which is particularly vital for Gaussian and polynomial kernels. Adequate scaling can improve model accuracy [100].

c) Gaussian Process Regression (GPR) Models

- **Kernel function:** Determines the covariance function that defines the smoothness and generalisation properties of the model. Typical choices include squared exponential, Matérn, and rational quadratic kernels [97].
- **Basis function:** This function specifies the mean function of the Gaussian process, which can be constant, linear, or custom-defined. Selecting an appropriate basis function can significantly enhance the model's fit.
- **Sigma (σ):** This represents the noise level in the data. Accurately estimating this parameter helps the model distinguish between signal and noise.

d) RF (Ensemble of Regression Trees)

- **Number of learners (Trees):** Specifies the number of trees in the ensemble. Increasing this number can improve performance; however, it can also increase computational cost [101].
- **Minimum leaf size:** As with individual regression trees, this parameter sets the minimum number of observations per leaf. Adjusting this parameter can help balance bias and variance.

- **Sample size:** The sample size determines the fraction of the training data used to train each tree. Additionally, smaller sample sizes can introduce more diversity among trees, potentially enhancing ensemble performance.
- **Learning rate:** Applicable primarily to boosted ensembles, it controls the contribution of each tree to the final model. Lower learning rates require more trees; however, it can also lead to better generalisation.

7) To adjust these hyperparameters in the Regression Learner App:

- **Select the Model Type:** Choose the desired model (e.g., Regression Trees, SVM, GPR, or Ensemble) from the model gallery.
- **Access Hyperparameter Setting:** In the model “Summary” tab, locate the Model Hyperparameters section.
- **Enable Optimisation:** Select the “Optimise” checkboxes to tune the desired hyperparameters.
- **Train the Model:** Click “Train” to start training with the desired hyperparameters. The app will perform optimisation to identify the best hyperparameter values.

For a more automated approach, the app supports “Optimisable” model options, which streamline the hyperparameter tuning process. Additionally, carefully adjusting these parameters can enhance the model’s performance and yield more accurate regression results.

8) Compare Models:

- Train multiple models and compare their performance:
 - ✓ Use the Compare feature to Juxtapose metrics across different models.
 - ✓ Identify the model with the best performance based on validation metrics.

9) Validate with Test Data:

- To assess model performance on unseen data:
 - ✓ Import a separate test dataset by clicking Test Data in the Test tab
 - ✓ Evaluate the model’s performance metrics on this test set to ensure it generalises well.

10) Export and Deploy the Model:

- Once the model satisfies the objectives established for the experiment:
 - ✓ Click Export Model to save the trained model to the MATLAB workspace.
 - ✓ Generate MATLAB code for the trained model by selecting Generate Function, facilitating automation and integration into other projects.
- For deployment, the model can be exported for use in Simulink or MATLAB Production Server, depending on the application's needs.

By following these steps, this study effectively utilised the Regression Learner App to develop and evaluate regression models tailored to its data analysis requirements.

The development of ML models for comparing scheduling algorithms in 5G WSNs should require a careful balance between technical rigour and practical insights. By systematically developing and applying these models, researchers in this field can uncover valuable insights that drive improvements in network QoS.

Appendix C: Implementation of DL Models in MATLAB

This subsection explores the practical implementation of DL models in MATLAB. This study examines how to leverage MATLAB’s built-in functions, toolboxes, and intuitive interface to streamline the process of building, training, and deploying various DL architectures. This subsection provides a hands-on guide to harnessing DL capabilities within the MATLAB ecosystem, from constructing neural networks and configuring training parameters to visualising results and exporting models for real-world applications [25], [26], [31], and [92].

1) CNN: Deep neural networks specialised in image recognition, image classification, and object detection. CNNs are specifically designed to process pixel data and excel at identifying patterns in images, such as edges, textures, and shapes, which can then be used to classify images or their parts [92], [104]. The architecture of a CNN model is typically formed of several layers, including convolutional layers, pooling layers, and fully connected layers, each of which performs different operations on the input data:

- **Convolutional Layers:** These layers perform convolutional operations that filter input images to extract features such as edges and corners. Thus, the models use filters or kernels that slide over the input image, creating a feature map that summarises the presence of detected features [92], [104].
- **Pooling Layers:** After the convolutional layers, pooling layers reduce the dimensions of the feature maps to decrease the computational load, control overfitting, and retain the essential information. For instance, max pooling (taking the maximum value from a set of values) is one of these layers' most commonly used techniques.
- **Fully Connected Layers:** Towards the end, CNN models typically include one or more fully connected layers, in which all neurons in a given layer are connected to all neurons in the subsequent layer. The role of these layers is to classify the images based on the features extracted and combined in the previous layers.
- Regarding the use of CNNs for classifying and predicting scheduling algorithms using image-based representations of network traffic data, the table below presents a few compelling reasons:

Table C.1: Rationale for Using CNN for Scheduling Algorithm Classification.

Aspect	Explanation
Pattern Recognition	CNN models identify patterns within images, making them adept at recognising network traffic patterns (such as congestion and bandwidth utilisation) from image-based data.

Feature Extraction	The CNN models automatically extract and learn features from images, eliminating the need for manual feature engineering and highlighting essential aspects of network traffic for analysis.
Adaptability	The CNN model's learning capabilities enable it to adapt to changing network traffic conditions, ensuring accurate predictions even as network dynamics evolve.
High Performance	CNN models have demonstrated high accuracy in image classification tasks, suggesting they can reliably classify and predict network traffic from images for scheduling algorithms.

Table C.1 displays the suitability and advantages of using CNNs for analysing network traffic represented as images for scheduling algorithm classification and prediction.

This investigative study used the DL Designer App in MATLAB to implement a CNN [92], [104]. This process involved a series of structured steps. Below are the detailed steps that guided this study through this process:

1) Launch the Deep Network Designer App:

- In MATLAB, navigate to the App tab.
- Under the ML and DL section, click on Deep Network Designer.

2) Create a New CNN Architecture:

- In the Start Page, select New and choose Blank Network to begin constructing the CNN model from scratch.
- Alternatively, the App offers a pre-trained network as a starting point for transfer learning.

3) Design the CNN Layers:

- Drag and drop layers from the layer Library into the Designer pane to build a CNN architecture.
- The CNN architecture includes:
 - ✓ Image Input Layer: This layer defines the image input size.
 - ✓ Convolutional Layers: These layers apply filters to the input image to extract features.
 - ✓ Batch Normalisation Layers: These layers normalise the activations and gradients, improving training stability.
 - ✓ ReLU Layers: Introduce non-linearity into the model.
 - ✓ Pooling Layers: These layers are responsible for downsampling the spatial dimensions to reduce computation and control overfitting.
 - ✓ Fully Connected Layers: These layers combine features learned by the convolutional layers for classification.
 - ✓ Softmax Layer: This layer converts the output probability distributions into a more interpretable form.
 - ✓ Classification Layer: This layer specifies the loss function for training.

4) Configure Layer Properties:

- Select each layer to modify its properties in the Properties pane. The key hyperparameters to consider for a CNN model include:
 - ✓ Filter size (convolutional layer): Determines the dimensions of the convolutional filters used in the layer.
 - ✓ Number of filters (convolutional layer): This parameter specifies the number of filters applied, which in turn affects the output depth.
 - ✓ Stride and padding (convolutional layer): Control the movement of filters across the input and handling of border pixels.
 - ✓ Pooling size (pooling layer): Defines the region over which pooling is performed
 - ✓ Number of neurons (fully connected layer): Sets the number of neurons in the layer, which in turn influences the model's capacity.

5) Analyse the Network:

- Click to analyse to verify architectural issues, such as mismatched input sizes or missing connections.
- Resolve any errors or warnings before proceeding to training.

6) Import and Prepare Data:

- Navigate to the Data tab and select "Import Data" to load the dataset.
- Ensure the data is organised appropriately, typically with images stored in folders named after their respective classes.
- The App supports importing data from MATLAB workspace variables or directly from folders.

7) Set Training Options:

- In the Training tab, configure training parameters by clicking "Training Options."
- The key parameters to tune in CNN models include:
 - ✓ Initial learning rate: Controls the step size during optimisation.
 - ✓ Max Epochs: Determines the number of times the entire dataset is passed through the network.
 - ✓ Mini-batch size: Specifies the number of samples per gradient update.
 - ✓ Optimiser: Choose between algorithms such as Stochastic Gradient Descent with Momentum (SGDM) or Adam.
 - ✓ Validation frequency: Sets how often the validation data is evaluated during training.

8) Train the Network:

- Click Train to initiate the training process.
- Monitor training progress through live plots of key metrics, including accuracy and loss.
- The App can accelerate training using the Parallel Computing Toolbox and a compatible Graphics Processing Unit (GPU).

9) Evaluate Model Performance:

- After training, evaluate the model's performance using metrics such as accuracy, precision, recall, and confusion matrices.
- Use the Training Results pane to visualise these metrics and identify potential areas for improvement.

10) Fine-Tune Hyperparameters:

Based on the evaluation, adjust the hyperparameters to improve the model's performance:

- Learning rate schedule: Experiment with different learning rates and implement learning rate decay to fine-tune the training process.
- Regularisation parameters: Adjust L2 regularisation to prevent overfitting.
- Data augmentation: Apply rotation, scaling, and flipping techniques to increase dataset variability.

11) Export and Deploy the Model:

- To export the model, click Export to save the trained network to the MATLAB workspace.
- Generate the MATLAB code for the network architecture by selecting Generate Code, facilitating reproducibility and further customisation.
- For deployment, export the model for integration into Simulink or generate standalone code for embedded systems.

By implementing these steps and carefully tuning the specified hyperparameters, this study effectively utilised a CNN model, developed using MATLAB's DL Designer App, to classify scheduling algorithms based on network traffic data captured from a live network environment [92], [104].

2) LSTM Networks: LSTMs are a particular type of RNN specifically designed to address the problem of learning long-term dependencies [92]. Traditional RNNs struggle with this due to the vanishing gradient problem, in which gradients during training become increasingly small, leading the network to forget earlier information. The LSTM overcomes this issue by incorporating mechanisms called gates that regulate the flow of information. Thus, an LSTM model is typically composed of three gates:

- **Forget Gate:** This gate determines what information is discarded from the cell state.
- **Input Gate:** This gate updates the cell state with new information.
- **Output Gate:** This gate determines what information from the current cell state is used to compute the output [92], [104].

Table C.2: Rationale for Using LSTM with image-based representation of network data.

Aspect	Explanation
Temporal Features	LSTM models are adept at capturing temporal dependencies, making them ideal for analysing network traffic data that inherently exhibits temporal characteristics.
Sequence Prediction	LSTMs excel at predicting future states based on past and current data, which is crucial for forecasting network conditions to inform improved scheduling decisions.
Handling Variable-Length Sequences	Network traffic data can vary in length over time. LSTMs effectively handle variable-length sequences, enabling flexible data analysis without requiring data manipulation, such as padding or truncation.
Feature Learning	LSTM models can automatically learn and identify important time-dependent features in network traffic data, enabling more nuanced and accurate classification and prediction.
Adaptability to New Patterns	As network conditions change, LSTMs can learn new patterns in the data when updated with recent information, ensuring they remain effective over time in predicting scheduling needs based on network traffic.

Table C.2 highlights the advantages and applicability of LSTMs in handling the complexities of network traffic data for the classification and prediction of scheduling algorithms [92], [104].

The implementation of an LSTM network for scheduling algorithm classification using image datasets using MATLAB's Deep Network Designer involved the following steps:

1) Launch the Deep Network Designer App:

- In MATLAB, navigate to the Apps tab.
- Under the ML and DL section, click on Deep Network Designer.

2) Prepare the image Dataset

- Ensure the image datasets are organised appropriately.
- Consistent Image dimensions
- Data augmentation

3) Design the LSTM Network Architecture:

- In the Deep Network Designer, start with a Blank Network to construct the LSTM model.
- A typical architecture for processing image datasets with LSTM layers includes:
 - ✓ Image input layer: The entry point for image data into the neural network. It specifies the expected dimensions of the input images and normalises

the data, ensuring it is appropriately scaled for training. This layer is crucial for determining the input size and preprocessing requirements, thereby facilitating the seamless integration of image data into the network architecture.

- ✓ Convolutional layers: Extract spatial features from individual frames.
- ✓ ReLU layers: Introduce non-linearity into the model.
- ✓ Pooling layers: Downsampling the spatial dimensions to reduce computation and control overfitting.
- ✓ LSTM layer: Processes the sequence of extracted features to capture temporal dependencies.
- ✓ Fully connected layer: Maps the LSTM outputs to the desired output size.
- ✓ Softmax layer (for classification): Converts the output to probability distributions
- ✓ Classification layer: Specifies the loss function for training.

4) Configure Layer Properties:

- Select each layer to modify its properties in the “Properties” pane. The key hyperparameters considered for the experiments included:
 - ✓ Filter size (convolutional layer): Determines the dimensions of the convolutional filters used in the layer.
 - ✓ Number of filters (convolutional layer): Specifies the number of filters applied, which affects the output depth.
 - ✓ Number of hidden units (LSTM layer): Sets the number of LSTM units, influencing the model’s capacity to learn temporal features.
 - ✓ Output mode (LSTM layer): Defines whether the layer outputs the last element or the entire sequence.
 - ✓ Number of neurons (fully connected layer): Sets the number of neurons in the layer, which in turn influences the model’s capacity.

5) Analyse the Network:

- Click “Analyse” to check for architectural issues, such as mismatched input sizes or missing connections.
- Resolve any issues or warnings before proceeding to training.

6) Import and Prepare Data:

- Ensure the image dataset is loaded into the MATLAB workspace.
- In the Deep Network Designer, import the dataset by selecting “Import Data”.
- The App supports importing data from MATLAB workspace variables or directly from folders.

7) Set Training Options:

- Configure training parameters in the “Training” tab by clicking “Training Options”.
- The key hyperparameters tuned for the experiments included:

- ✓ Initial learning rate: Controls the step size during optimisation.
- ✓ Max epochs: Determines the number of times the entire dataset is passed through the network.
- ✓ Mini-batch size: Specifies the number of samples per gradient update.
- ✓ Optimiser: Choose between algorithms such as “sgdm” and “Adam”.
- ✓ Validation frequency: Sets how often the validation data is evaluated during training.

8) Train the Network:

- Click “Train” to initiate the training process.
- Monitor training progress with live plots of metrics like accuracy and loss.
- Use the Parallel Computing Toolbox and a compatible GPU (if available) to accelerate training.

9) Evaluate LSTM Model Performance:

- After training the LSTM model, assess the model’s performance using metrics such as accuracy, precision, recall, and confusion matrices.
- Use the “Training Results” pane to visualise the metrics and identify potential areas for improvement.

10) Fine-Tune Hyperparameters:

Based on the evaluation results, experiment with different hyperparameters to improve performance:

- Learning rate schedule: Implement learning rate decay to fine-tune the training process.
- Regularisation parameters: Adjust L2 regularisation to prevent overfitting.
- Data augmentation: Apply scaling, flipping, and rotation schemes to increase dataset variability.

11) Export and Deploy the Model:

- When satisfied with the model, click “Export” to save the trained network to the MATLAB workspace.
- Generate MATLAB code for the network architecture by selecting “Generate Code”, facilitating reproducibility and more customisation.
- For deployment, generate standalone code for embedded systems or export the model for integration into Simulink.

Following these structured steps and carefully tuning hyperparameters, this study designed and implemented an LSTM model for image representation of network datasets using MATLAB’s Deep Network Designer App [92], [104].

Appendix D: Implementation of ML and DL Models Using Python

1) Implementation of Decision Tree Models Using Python: The methodology for developing and evaluating a decision tree classifier using a Python software package involves several steps: data preparation, model building, model evaluation,

hyperparameter tuning, and final model evaluation. Each of these stages is critical to ensuring the robustness and accuracy of the model [107], [108], and [109].

a) Data Preparation: Data preparation encompasses collecting, cleaning, and splitting the dataset to ensure it is suitable for model training and evaluation. Table D.1 presents the steps involved in the data preparation stage.

Table D.1: Data Preparation Workflow for Decision Tree Models.

Step	Description	Example/Tools
1.1 Data Collection	Gather the dataset relevant to our problem and ensure it contains both input features and target variables	CSV files, databases, APIs
1.2 Data Cleaning	Handle missing values, outliers, and inconsistencies. If necessary, convert categorical variables to numerical representations (e.g., one-hot encoding).	Pandas, NumPy (Python)
1.3 Data Splitting	Divide the data into training, validation, and test sets. The training set builds the model, the validation set tunes hyperparameters, and the test set evaluates the final performance.	train_test_split (scikit-learn)

b) Model Building: This stage encompasses initialising the decision tree classifier and growing the tree based on the training data [107], [108], [109]. Table D.2 presents the steps required to build a decision tree model.

Table D.2: Steps in Decision Tree Model Building.

Step	Description	Example/Tools
2.1 Initialisation	Create a decision tree classifier object, specifying initial hyperparameters such as criterion and maximum depth.	DecisionTreeClassifier (scikit-learn)
2.2 Tree Growing	Recursively split the training data based on features that best separate the target variable until the stopping criteria (e.g., maximum depth) are met.	Handled internally by the algorithm (e.g., Gini impurity, entropy)

c) Model Evaluation: Evaluate the trained model's performance using appropriate metrics, as outlined in Table D.3 detailing the key steps for assessing Decision Tree models.

Table D.3: Decision Tree Model Evaluation Steps.

Step	Description	Example/Tools
3.1 Prediction	Use the trained decision tree to predict the performance of the validation or test set.	Predict method (scikit-learn)
3.2 Performance Metrics	Assess the model performance using metrics suitable for the problem type, such as accuracy, precision, recall, and F1-score for classification tasks	accuracy_score, precision_score, recall_score, f1_score (sci-kit-learn)

d) Hyperparameter Tuning (Iterative): This stage involves optimising model performance by adjusting hyperparameters through iterative training and evaluation to determine the optimal set of parameters for the Decision Tree model [107]-[109]. Table D.4 presents the steps involved.

Table D.4: Steps in Decision Tree Model Hyperparameter Tuning.

Step	Description	Example/Tools
4.1 Parameter Selection	Hyperparameters such as max_depth, min_sample_split, and criterion are identified for adjustment.	Grid Search, Random Search, Bayesian Optimisation
4.2 Model Training and Evaluation	Train the model with selected hyperparameters and evaluate its performance on the validation set.	Repeat steps 2 and 3
4.3 Iteration	Continue adjusting hyperparameters and retraining the model until optimal performance is achieved.	

e) Final Model Evaluation: Select the best-performing model and evaluate its performance on the test set to ensure generalisation. Table D.5 presents the steps involved.

Table D.5: Final Steps in Evaluating the Decision Tree Model.

Step	Description
5.1 Model Selection	Choose the model with the best validation performance based on the selected metrics.
5.2 Final Evaluation	Assess the chosen model's performance on the test set to obtain an unbiased estimate of its generalisation capability.

2) Implementation of EGPR Models Using Python: Implementing an EGPR model using Python involves several systematic steps. These steps encompass data preparation, model building, model evaluation, hyperparameter tuning, and final model evaluation. This comprehensive approach ensures the design and implementation of a robust and accurate predictive model [107]–[109].

a) Data preparation: The data preparation process is the foundational step that ensures the dataset is suitable for modelling. This step involves collecting, cleaning, and splitting the data accordingly. Table D.6 presents the steps involved.

Table D.6: Data Preparation Workflow for EGPR Models.

Step	Description	Example/Tools
1.1 Data Collection	Collect data for the dataset containing features (input and target variables).	CSV files, databases, APIs
1.2 Data Cleaning	Handle missing data values, outliers, and inconsistencies. Normalise and standardise features if needed, as it can improve the performance and stability of Gaussian processes.	Pandas, Numpy (Python)
1.3 Data Splitting	Divide the dataset into training, validation, and test sets. The training set is used to train the EGPR model, the validation set for hyperparameter tuning, and the test set for evaluating the final model.	train_test_split (sci-kit learn)

b) Model Building: Building the EGPR model encompasses selecting appropriate kernel and likelihood functions, initialising the model, and training it with the prepared data. The steps required for this process are presented in Table D.7.

Table D.7: Steps in EGPR Model Building.

Step	Description	Example/Tools
2.1 Kernel Selection	Choose a kernel function (covariance function) that defines the similarity between data points. Common choices include RBF, Matern, and Polynomial.	RBF, Matern, RationalQuadratic, ExpSinesSquared, DotProduct (scikit-learn, GPy)
2.2. Likelihood Function Selection	Choose a likelihood function that models the distribution of the target variable given the latent function. For EGPR, the extended likelihood is used.	Use the RobustMax likelihood function, e.g., likelihoods.RobustMax
2.3 Model Initialisation	Create an EGPR model object. Provide the kernel function, likelihood function, and initial hyperparameter values.	GPR (scikit-learn), GPRegressor (GPy) with RobustMax likelihood function
2.4 Model Training	Train the model using the training data. The training entails optimising the model's hyperparameters (kernel parameters and noise variance) by maximising the marginal likelihood or using other optimisation methods such as MCMC.	fit method (scikit-learn), optimise or optimize_restarts (GPy)

c) Model Evaluation: Evaluating the model's performance is vital to ensure its predictive accuracy and reliability. The model's evaluation steps are listed in Table D.8.

Table D.8: EGPR Model Evaluation Steps.

Step	Description	Example/Tools
3.1 Prediction	Use the trained model to make predictions on the validation or test set. EGPR provides both the predictive mean and the predictive variance (uncertainty).	Predict method (scikit-learn, GPy)

3.2 Performance Metrics	Evaluate the model's performance using appropriate metrics for regression, such as RMSE, MSE, MAE, and R-squared.	mean_absolute_error, mean_squared_error, r2_score (scikit-learn)

d) Hyperparameter Tuning (Iterative): Optimising the hyperparameters is crucial for enhancing the EGPR model's performance and ensuring it generalises well to unseen data. Table D.9 presents the steps for this tuning process.

Table D.9: Steps in EGPR Model Hyperparameter Tuning.

Step	Description	Example/Tools
4.1 Parameter Selection	Choose a set of hyperparameters to adjust (e.g., kernel parameters such as length scale, variance, and noise variance of the likelihood).	Grid Search, Random Search, Bayesian Optimisation
4.2 Model Training and Evaluation	Train a new model with the selected hyperparameters using the training set and evaluate its performance on the validation set.	Repeat steps 2 and 3
4.3 Iteration	Repeat steps 4.1 and 4.2 with different hyperparameter combinations until the desired performance is achieved on the validation set.	

e) Final Model Evaluation: This stage ensures that the selected model performs well on unseen data, providing an unbiased estimate of its generalisation ability. Table D.10 lists the steps involved in the final model evaluation.

Table D.10: Final EGPR Model Evaluation Steps.

Step	Description
5.1 Model Selection	Choose the model with the best performance on the validation set, based on the chosen metric (such as minimising RMSE or maximising R-squared).

5.2 Final Evaluation	Evaluate the selected model's performance on the held-out test set to get an unbiased estimate of its generalisation ability.

3) Implementation of SVMs Models Using Python: The methodological approach for implementing SVM models entails several vital steps. These steps have been addressed in the discussion of the implementation of the models previously discussed (Decision Trees and EGPR) [107]-[109].

a) Data Preparation: This process encompasses collecting, cleaning, and splitting the dataset to ensure it is suitable for SVM model training and evaluation. Table D.11 outlines the procedure.

Table D.11: Data Preparation Workflow for SVMs Models.

Step	Description	Example/Tools
1.1 Data Collection	Create the dataset containing features (input variables) and a target variable (categorical for classification or continuous for regression).	CSV files, databases, APIs
1.2 Data Cleaning	This process involves handling missing values, outliers, and inconsistencies. Normalising or standardising features, especially when using kernel-based SVMs, can improve performance and convergence speed.	Pandas, NumPy (Python)
1.3 Data Splitting	The data should be divided into training, validation and test sets. The training set is used to train the SVM model, the validation set to tune hyperparameters, and the test set to evaluate the final model.	train_test_split (scikit-learn)

b) Model Building: This stage entails selecting the appropriate kernel, initialising the SVM model, and training it with preprocessed data. Table D.12 outlines the steps required to build an SVM model.

Table D.12: Steps in SVMs Model Building.

Step	Description	Example/Tools
2.1 Kernel Selection	Choose a kernel function if using a kernel-based SVM. Common choices include linear, polynomial, RBF (Radial Basis Function), and sigmoid. The kernel function defines how data is transformed into a high-dimensional space.	Linear, poly, RBF, sigmoid (scikit-learn)
2.2 Model Initialisation	Create an SVM classifier or regressor object. Specify the chosen kernel and any initial hyperparameter values (e.g., C, gamma, degree for polynomial kernel)	SVC (for classification), SVR (for regression) (scikit-learn)
2.3 Model Training	Train the SVM model using the training data. The training entails finding the optimal hyperplane that separates the classes (for classification) or approximates the target variable (for regression) while maximising the margin.	fit method (scikit-learn)

c) Model Evaluation: This stage involves evaluating the trained SVM model's performance using appropriate metrics to ensure its robustness and effectiveness. Table D.13 presents the steps for this process.

Table D.13: SVM Model Evaluation Steps.

Step	Description	Example/Tools
3.1 Prediction	Use the trained SVM model to predict the validation and test sets.	Predict method (scikit-learn)
3.2 Performance Metrics	Evaluate the model's performance using suitable metrics. For classification, accuracy, precision, recall, and F1-score are used. For	accuracy_score, precision_score, recall_score, f1_score, mean_absolute_error, mean_squared_error, r2_score

	regression, use RMSE, MAE, MSE, and R-squared.	(scikit-learn)

d) Hyperparameter Tuning (Iterative): This stage optimises the model's hyperparameters to enhance performance and generalisation. Table D.14 outlines the steps involved in this process.

Table D.14: Steps in SVM Model Hyperparameter Tuning.

Step	Description	Example/Tools
4.1 Parameter Selection	Choose a set of hyperparameters to adjust (e.g., C, gamma, kernel-specific parameters).	Grid Search, Random Search, Bayesian Optimisation
4.2 Model Training and Evaluation	Train a new SVM model with the selected hyperparameters using the training set and evaluate its performance on the validation set.	Repeat steps 2 and 3
4.3 Iteration	Repeat steps 4.1 and 4.2 with different hyperparameter combinations until the desired performance is achieved on the validation set.	

e) Final Model Evaluation: This process involves selecting the best-performing model and assessing its performance on the test set to ensure its reliability. Table D.15 presents the steps required for this final evaluation.

Table D.15: Final SVM Model Evaluation Steps.

Step	Description
5.1 Model Selection	Choose the SVM model that achieved the best performance on the validation set (based on the chosen metric).
5.2 Final Evaluation	Evaluate the selected model's performance on the held-out test set to get an unbiased estimate of its generalisation ability.

4) Implementation of Ensemble Models Using Python: The methodology for implementing Ensemble Models involves the following structured approach: base learner selection, ensemble building, model evaluation, hyperparameter tuning, and final model

evaluation. Each stage is crucial to ensuring the robustness and accuracy of the ensemble model [107]–[109].

a) Data Preparation

Table D.16 details the necessary steps for data preparation for ensemble models.

Table D.16: Data Preparation Workflow for Ensemble Models.

Step	Description	Example/Tools
1.1 Data Collection	Gather the dataset with features and variables	CSV files, databases, APIs
1.2 Data Cleaning	Handle dataset issues (missing values, outliers, and inconsistencies). Feature scaling might be necessary, depending on the base learners	Pandas, NumPy (Python)
1.3 Data Splitting	Divide the data into training, validation and test sets.	train_test_split (scikit-learn)

b) Base Learner Selection: This stage involves selecting the appropriate base learner, which is fundamental for building a strong ensemble model. Table D.17 outlines the steps required for this process.

Table D.17: Steps Involved in Base Learner Selection for Ensemble Models.

Step	Description	Example/Tools
2.1 Model Types	Choose the types of base learners to include in the ensemble. This could involve using multiple instances of the same model type (e.g., decision trees in Random Forest) or a mix of different model types (e.g., decision trees, SVMs, logistic regression)	DecisionTreeClassifier, SVC, LogisticRegression (scikit-learn)
2.2 Base Learner Initialisation	Create instances of the chosen base learners. Specify initial hyperparameter values for these base learners if necessary to ensure they are appropriately configured for the ensemble method.	DecisionTreeClassifier (max_depth=5), SVC (kernel= “RBF”) (scikit-learn)

c) Ensemble Building: This stage involves training the base learners using a strategy tailored to the chosen ensemble method. Table D.18 outlines the steps required for this process.

Table D.18: Steps in Ensemble Model Building.

Step	Description
3.1 Training Strategy	Train the base learners using a strategy tailored to the ensemble method: ▪ Bagging (e.g., Random Forest): Train each base learner on a different bootstrap sample of the training data (with replacement). ▪ Boosting (e.g., AdaBoost, Gradient Boosting): Train base learners sequentially, where each subsequent learner focuses on correcting the errors of the previous learners. ▪ Stacking: Train base learners independently, then train a meta-learner on their predictions to make the final prediction.

d) Model Evaluation: This stage assesses the ensemble model's performance to verify its predictive accuracy and reliability. Table D.19 details the evaluation steps.

Table D.19: Ensemble Model Evaluation Steps.

Step	Description	Example/Tools
4.1 Prediction	Use the trained ensemble model to predict the validation and test sets. Predictions are typically aggregated from the base learners (e.g., averaging for regression and majority voting for classification).	predict method (scikit-learn)
4.2 Performance Metrics	Evaluate the ensemble's performance using suitable metrics based on the problem type	For classification: accuracy_score, precision_score, recall_score, and f1_score. For regression: mean_squared_error, mean_absolute_error, r2_score, f1_score,

		recall_score, precision_score, and accuracy_score.

e) Hyperparameter Tuning (Iterative): This stage optimises the model’s configuration to improve performance and generalisation. Table D.20 outlines the specific tuning steps.

Table D.20: Steps in Ensemble Model Hyperparameter Tuning.

Step	Description	Example/Tools
5.1 Parameter Selection	Choose the hyperparameters of the ensemble method and the base learners to adjust (e.g., number of base learners, learning rate, kernel parameters).	Grid Search, Random Search, Bayesian Optimisation
5.2 Model Training and Evaluation	Train a new ensemble with the selected hyperparameters using the training set and evaluate its performance on the validation and test sets.	Repeat steps 2 to 4
5.3 Iteration	Repeat steps 5.1 and 5.2 with different hyperparameter combinations until the desired performance is achieved on the validation or test set.	

f) Final Model Evaluation: This stage selects and evaluates the final ensemble model to ensure its robustness and effectiveness on new (unseen) data. Table D.21 outlines the steps involved.

Table D.21: Final Ensemble Model Evaluation Steps.

Step	Description
6.1 Model Selection	Choose the ensemble model with the best performance on the validation and test sets based on the chosen metric (e.g., maximising F1-score for classification and minimising RMSE for regression).
6.2 Final Evaluation	

5) Implementation of LSTM Models Using Python: Implementing the LSTM model with PyTorch (a popular, open-source DL framework known for its flexibility and ease of use) involves several systematic steps: data preparation, model architecture design,

model training, model evaluation, hyperparameter tuning, and final model evaluation. This structured approach ensures the development of a robust, accurate LSTM-based predictive model for network traffic prediction [107]–[109].

a) Data Preparation: This stage is a critical step that ensures the dataset is suitable for modelling. It involves collecting, preprocessing, and appropriately splitting the data. Table D.22 details the steps involved.

Table D.22: Data Preparation Workflow for LSTM Models.

Step	Description	Example/Tool
1.1 Data Collection	Gather a sequential dataset. This study gathered network traffic data	CSV files, audio files, databases, APIs
1.2 Data Preprocessing	Clean the data by handling missing values and outliers. Normalise or standardise numerical data if required. Create input sequences of a fixed length suitable for LSTM processing.	Pandas, NumPy
1.3 Data Splitting	Divide the data into training, validation, and test sets. For time-series data, maintain temporal order by avoiding shuffling during splitting.	Custom splitting scripts, torch.utils.data.random_split (with shuffling disabled for time series data)

b) Model Architecture Design: This stage entails defining the LSTM layers and configuring the model’s components using PyTorch’s nn module. Table D.23 outlines the steps required to build the model.

Table D.23: Steps in LSTM Model Building.

Step	Description	Example/Tools
2.1 Layer Definition	Define the layers of the LSTM Model: - Embedding layer - LSTM layers - Dropout layers - Dense layers	PyTorch
2.2 Model Compilation	Create a model object. Define the loss function (e.g., categorical cross-entropy for multi-class classification, binary cross-entropy for binary classification, MSE for regression), the optimiser (e.g., Adam,	PyTorch

	RMSprop), and metrics (e.g., accuracy).	

The definition of the layers of the LSTM model involved:

- Embedding layer: Converts categorical data (e.g., words) into dense vectors
- LSTM layer: Core layers that process the sequence data, maintaining hidden and cell states.
- Dropout layers: Regularisation layer to prevent overfitting

c) Model Training: This stage involves implementing a training loop that processes input sequences, computes the loss, and updates the model's weights. Table D.24 outlines the steps involved.

Table D.24: Steps in LSTM Model Training.

Step	Description	Example/Tools
3.1 Training Process	Train the LSTM model using the training data. The model processes input sequences, calculates the loss, and updates weights and biases using the optimiser and backpropagation through time (BPTT).	PyTorch
3.2 Epochs and Batches	Train for a certain number of epochs with a specific batch size.	

d) Model Evaluation: This stage assesses the trained LSTM model's performance using appropriate metrics to verify its effectiveness. Table D.25 details the steps involved.

Table D.25: LSTM Model Evaluation Steps.

Step	Description	Example/Tools
4.1 Prediction	Use the trained model to make predictions on the validation and test sets.	model.eval(), torch.no_grad(), model(input)
4.2 Performance Metrics	Evaluate the model using suitable metrics (e.g., accuracy, precision, recall, F1-score for classification and MAE, MSE, RMSE for regression).	accuracy_score, precision_score, recall_score, f1_score (scikit-learn)

e) Hyperparameter Tuning (Iterative): This stage involves **optimising** the LSTM model’s hyperparameters to improve performance, enhance robustness, and ensure effective generalisation to unseen data. Table D.26 details the hyperparameter tuning for the LSTM models.

Table D.26: Steps in LSTM Model Hyperparameter Tuning.

Step	Description	Example/Tools
5.1 Parameter Selection	Choose hyperparameters to adjust. For instance, the number of LSTM layers, the number of units in each LSTM layer, the dropout rate, the learning rate, the optimiser, the batch size, and the sequence length.	Grid Search, Random Search, Bayesian Optimisation (using libraries such as Optuna)
5.2 Model Training and Evaluation	Train a new LSTM model with the selected hyperparameters and evaluate its performance on the validation set. This may involve modifying the model architecture and retraining.	Repeat steps 2 to 4
5.3 Iteration	Adjust the hyperparameters and retrain the model until the desired performance metrics are achieved on the validation set. Techniques such as early stopping can also be employed to prevent overfitting.	

f) Final Model Evaluation: This step involves selecting the optimal LSTM model and evaluating its performance on a held-out test set to ensure its robustness and generalisation. Table D.27 outlines the specific procedure.

Table D.27: Final LSTM Model Evaluation Steps.

Step	Description
6.1 Model Selection	Choose the LSTM model with the best performance on the validation set based on the chosen metrics (e.g., maximising F1-score for classification and minimising RMSE for regression).
6.2 Final Evaluation	Evaluate the performance of the selected LSTM model on the held-out test set to

	obtain an unbiased estimate of its generalisation abilities.

6) Implementation of CNN Models Using Python: The methodological approach for implementing CNN models encompasses the following structured steps: data preparation, model architecture design, model training, model evaluation, hyperparameter tuning, and final model evaluation. Each step is crucial to ensuring the robustness and accuracy of the CNN-based predictive model [107]–[109].

a) Data Preparation: Similar to the previously discussed model, this phase involves collecting, preprocessing, and splitting the dataset to ensure it is suitable for training and evaluating the CNN model. Table D.28 details the specific steps.

Table D.28: Data Preparation Workflow for CNN Models.

Step	Description	Example/Tools
1.1 Data Collection	Gather the dataset. This dataset must contain images with corresponding labels for image classification.	Image files and image databases.
1.2 Data Preprocessing	Resize images to a uniform size (e.g., 227 x 227). Normalise pixel values (e.g., scale them to 0-1). Data augmentation, if necessary (e.g., applying transformations such as rotations or flips to increase the size of the training dataset artificially).	OpenCV, PIL, scikit-image (Python image processing libraries)
1.3 Data Splitting	Divide the data into training, validation, and test sets.	Random_split (PyTorch) train_test_split (scikit-learn)

b) Model Architecture Design: This phase involves defining a sequence of CNN layers to extract and process the features from the input image. This process is detailed in Table D.29 [107]–[109].

Table D.29: Steps in CNN Model Building.

Step	Description	Example/Tools
2.1 Layer Definition	Definition of the CNN layers. This involves a sequence of: - Convolutional layers - Activation functions - Pooling layers - Dropout layers - BatchNormalisation layers - Flatten layers - Dense (fully connected) layers	nn.Conv2d, nn.ReLU, nn.MaxPool2d, nn.Dropout, nn.BatchNorm2d, nn.Flatten, nn. Linear
2.2 Model Compilation	Create a model object by connecting the layers. Specify the loss function (e.g., categorical cross-entropy for multi-class classification, binary cross-entropy for binary classification, MSE for regression), the optimiser (e.g., Adam, SGD, RMSprop), and metrics (e.g., accuracy).	Torch.nn.Module Torch.nn.Sequential (PyTorch)

c) Model Training: This stage entails iteratively updating the CNN's weights based on the loss computed from the model's predictions and the actual labels. The training process iterates over multiple epochs. Table D.30 outlines the procedural steps for this phase.

Table D.30: Steps in CNN Model Training.

Step	Description	Example/Tools
3.1 Training Process	Train the CNN model using the training data. The training involves feeding the data to the model, calculating the loss, and using backpropagation and the optimiser to update the model's weights and biases.	model.train(), Forward pass: outputs = model(images), Loss calculation: loss = criterion(outputs, labels), Backwards pass: loss.backward(),

		Weight update: optimizer.step(), optimizer.zero_grad()
3.2 Epochs and Batches	Train for a certain number of epochs (passes through the entire training dataset) with a specific batch size (the number of samples processed before the model's internal parameters are updated).	Dataloader, batch_size, shuffle (PyTorch)

d) Model Evaluation: This phase entails assessing the CNN's performance by generating predictions on the validation or test set and computing appropriate metrics to determine the model's effectiveness. Table D.31 details the specific steps.

Table D.31: CNN Model Evaluation Steps.

Step	Description	Example/Tools
4.1 Prediction	Use the trained CNN model to predict the validation and test sets.	model.eval(), torch.no_grad(), model(input)
4.2 Performance Metrics	Evaluate the model's performance using suitable metrics (e.g., accuracy, recall, precision, F1-score, AUC for classification; RMSE, MSE, MAE for regression).	Accuracy_score, recall_score, precision_score, f1_score, and roc_auc_score

e) Hyperparameter Tuning (Iterative): Optimising the hyperparameters is vital for improving the CNN model's performance and ensuring it generalises well to unseen data. This process involves defining a search space, training models with various configurations, and assessing their effectiveness. Table D.32 outlines the specific steps.

Table D.32: Steps in CNN Model Hyperparameter Tuning.

Step	Description	Example/Tools
5.1 Parameter Selection	Choose hyperparameters to adjust (e.g., number of layers, filter sizes, number of filters, learning rate, optimiser, activation functions, dropout rate, and batch size).	Random Search, Grid Search, and Bayesian Optimisation using libraries such as Optuna or Ray Tune.

5.2 Model Training and Evaluation	Train a new model with the selected hyperparameters and evaluate its performance on the validation set. This may involve modifying the model architecture and retraining.	Repeat steps 2 to 4
5.3 Iteration	Repeat steps 5.1 and 5.2 with different hyperparameter settings until the desired performance is achieved.	

f) Final Model Evaluation: Assess the optimised CNN model to confirm its generalisation capabilities. Table D.33 outlines the steps taken for this evaluation.

Table D.33: Final CNN Model Evaluation Steps.

Step	Description	Example/Tools
6.1 Model Selection	Choose the CNN model with the best performance on the validation set, based on the chosen metric (maximising accuracy or minimising loss).	
6.2 Final Evaluation	Evaluate the performance of the selected CNN model on the held-out test set to get an unbiased estimate of its generalisation abilities.	Use the same evaluation metrics as in validation (e.g., accuracy)

7) Implementation of Transformer Neural Network Models Using Python

Implementing Transformer neural network models involves a systematic approach, starting with data collection to create the dataset, followed by preprocessing and splitting it into subsets [107]–[109].

a) Data Collection: Table D.34 details the steps involved in the data collection for Transformer neural network models.

Table D.34: Data Preparation Workflow for Transformer Neural Network Models.

Step	Description	Example/Tools
1.1 Data Collection	Create the dataset (network traffic data)	Network monitoring tools such as Wireshark or custom scripts
1.2 Data Preprocessing	Process raw network traffic data, normalise features, and handle missing values.	Pandas, NumPy

1.3 Data Splitting	Divide the data into training, validation, and test sets.	PyTorch tools such as <code>torch.utils.data.DataLoader</code>

b) Model Architectural Design: This stage entails selecting the optimal architecture regarding encoder and decoder utilisation, depending on the problem domain. While translation tasks typically necessitate an encoder-decoder structure, text classification is generally best served by an encoder-only architecture. Table D.35 details the procedure for constructing the model.

Table D.35: Steps in Transformer Neural Network Model Building.

Step	Description	Example/Tools
2.1 Encoder-Decoder or only Encoder/Decoder	The Transformer consists of an encoder and a decoder. Some tasks only use one of these components. Deciding which architecture is most suitable for solving the problem is vital.	PyTorch <code>nn.Transformer</code> , TensorFlow Keras Transformer APIs
2.2 Layer Definition	Define the components of the Transformer: - Input embeddings - Positional Encoding - Encoder layers - Multi-Head Self-Attention - Feed Forward Neural Network - Decoder layer - Output layer	PyTorch <code>nn.Embedding</code> , <code>nn.MultiheadAttention</code> , TensorFlow Layer APIs
2.3 Model Compilation	Create the model object. Define the loss function (e.g., categorical cross-entropy for classification), the optimiser (e.g., Adam, SGD), and metrics (e.g., accuracy).	<code>Torch.optim.Adam</code> (PyTorch), <code>tf.keras.losses.CategoricalCrossentropy</code> , <code>Tf.keras.metrics.Accuracy</code>

c) Model Training: This stage involves processing input sequences through the network layers (encoder, decoder, or both) to generate predictions. The model learns by minimising a loss function via backpropagation and gradient-based optimisation algorithms. Table D.36 details the steps involved in this process.

Table D.36: Steps in Transformer Neural Network Model Training.

Step	Description	Example/Tools
3.1 Training Process	Train the model using the training dataset. The model processes input sequences, calculates the loss, and updates its weights and biases using backpropagation and the optimiser.	torch.utils.data.DataLoader, torch.nn.Transformer, torch.nn.CrossEntropyLoss, torch.nn.MSELoss, torch.optim.Adam, torch.optim.SGD, optimizer.zero_grad(), loss.backward()
3.2 Epochs and Batches	Train for a certain number of epochs with a specified batch size.	

d) Model Evaluation: This stage assesses the Transformer model's effectiveness in learning and generalising underlying data patterns. The procedure typically follows the steps described in Table D.37 [107]–[109].

Table D.37: Transformer Neural Networks Model Evaluation Steps.

Step	Description	Example/Tools
4.1 Prediction	Use the trained model to make predictions on the validation and test sets.	nn.Model, Tensor, device, no_grad(), eval(), DataLoader
4.2 Performance Metrics	Evaluate the model using suitable metrics.	torchmetrics, torch.Tensor

e) Hyperparameter Tuning (Iterative): This stage employs an iterative process to determine the optimal settings for the hyperparameters defined in Table D.38. During each iteration, the model is trained and evaluated on validation data, using performance metrics to identify the most effective configuration.

Table D.38: Steps in Transformer Neural Network Model Hyperparameter Tuning.

Step	Description	Example/Tools
5.1 Hyperparameter Selection	Define the hyperparameters to tune (e.g., number of layers, number of attention heads, hidden dimension, dropout rate, learning rate, optimiser, and batch size).	Bayesian Optimisation (can be used with libraries such as Optuna), Grid Search, and Random Search.
5.2 Model Training and Evaluation	Train a new model with the selected hyperparameters and evaluate it on the validation set.	Repeat steps 2 to 4
5.3 Iteration		

f) Final Model Evaluation: This critical phase moves beyond the training and validation datasets to assess the model’s ability to generalise to new, unseen data. It is essential for determining the model’s suitability for the intended task and provides a realistic estimate of its performance in real-world scenarios. Table D.39 outlines the steps involved in the final evaluation of a Transformer neural network model.

Table D.39: Final Transformer Neural Network Model Evaluation Steps.

Step	Description
6.1 Model Selection	Choose the model with the best performance on the validation set
6.2 Final Evaluation	Evaluate the selected model on the held-out test set to obtain an unbiased estimate of its generalisation ability.

8) Implementation of the RL Model

Table D.40 presents an overview of the key steps involved in implementing an RL technique to prioritise network traffic. Each row outlines a distinct phase of the process, covering environment setup, algorithm selection, and final deployment [107]–[109].

Table D.40: Fundamental Steps in RL Model Implementation.

Step	Description	Example/Tools
2.1 Environment Definition	Definition of the network simulation or real-world setup in which the RL agent will operate. This process includes specifying how traffic flows are generated,	- Custom Python scripts - Network simulators such as NS-3 or Mininet - Docker-based testbeds in a controlled environment

	how congestion is modelled, and how varied network conditions (bandwidth, latency, packet loss) are during training.	
2.2 State and Action Spaces	Determination of which observations (states) the RL agent will have access to (e.g., traffic load, queue lengths, latency, packet arrival rates) and what actions it can take (e.g., assigning traffic priorities, throttling rates, packet scheduling).	- NumPy arrays for representing states - Discrete or continuous action spaces in OpenAI Gym-like environments
2.3 Reward Definition	Specify how the agent's performance will be measured at each time step. In network traffic prioritisation, reward signals could be based on latency, throughput, packet loss, rate, or a weighted combination to encourage balanced network performance.	- Custom reward functions in Python - Standard metrics are throughput, delay, and packet drops.
2.4 RL Algorithm Selection	Select a suitable RL algorithm (e.g., DQN, PPO, DDPG, A2C) based on the nature of the state and action spaces, as well as the desired performance. Consider whether value-based or policy-based methods are appropriate or if actor-critic architectures would be beneficial.	- PyTorch or TensorFlow for model building - Libraries such as Stable Baselines, Ray RLlib, or custom RL implementations
2.5 Model Training	Train the RL agent in the defined environment. This involves running multiple episodes of network simulation, updating the agent's policy or value function based on the reward signals, and tuning hyperparameters (learning	- PyTorch optimisers (Adam, RMSProp) - Hyperparameter tuning with Optuna or Ray Tune.

	rate, exploration/exploitation)	
2.6 Model Evaluation and Testing	Evaluate the trained model's performance under traffic patterns, load scenarios, and network conditions. Compare RL-driven results with the baseline methods (e.g., FIFO scheduling, priority-based queues) to quantify improvement.	- Automated test scripts in Python - Visualisation tools (Matplotlib, Grafana dashboards) for monitoring performance
2.7 Deployment	Integrate the trained RL model into the actual (or production-like) network environment, ensuring it can process incoming traffic data in real-time and make scheduling or prioritisation decisions. Monitor performance and continue iterating as needed.	- Model export formats (ONNX, TorchScripts) - Real-time inference servers (TensorFlow Servicing, TorchServe)

Appendix E: Source Codes for Experiment 1 (Comparative Evaluation of Scheduling Algorithms with ML Regression Models)

This appendix presents the source code used in Experiment 1, which evaluated baseline-scheduling algorithms in a live wireless network. The scripts implement the PF and RR schedulers, which serve as benchmarks for assessing network performance, and compare these schedulers using statistical and ML methods. Each code block corresponds to the core logic used during execution and data collection.

The implementation details provided in this appendix support the experimental procedures described in *Section 4.2*, the performance evaluations presented in *Section 5.2*, and the broader findings and implications discussed in *Section 6.2* regarding baseline-scheduling algorithms for 5G WSNs.

Appendix E.1: Source Code for Proportional Fair Scheduler

```
#!/usr/bin/env python3

"""

This script implements a network monitoring and scheduling simulation
tool. It uses
```

the Proportional Fair (PF) scheduling algorithm to select a target device from a predefined list within a Wireless Local Area Network (WLAN).

For each scheduling interval, the script pings the selected device using the icmplib library to measure Quality of Service (QoS) metrics. The collected metrics include

delay (round-trip time), jitter (delay variation), packet loss, and throughput.

Throughput is calculated based on a fixed ICMP payload size of 64 bytes.

The PF metric, a key component of the scheduler, is computed as the ratio of a

device's instantaneous throughput to its long-term average throughput. This ensures

a balance between maximising overall network throughput and maintaining fairness

among all devices.

All collected data is logged to a uniquely timestamped CSV file with a structure

optimised for subsequent machine learning analysis. The script also provides

real-time logging to the console and a corresponding log file.

"""

Import standard libraries for time, data handling, and system operations.

import time

import csv

import random

import logging

import statistics

from datetime import datetime

import os

Import the 'ping' function from the icmplib library for ICMP communication.

from icmplib import ping

```

# Configuration and Logging Setup

# -----

# Defines the time in seconds to wait between each scheduling decision
and ping.

TRANSMISSION_INTERVAL = 0.5

# Sets the total duration of the experiment in seconds. A value of 0
runs indefinitely.

EXPERIMENT_DURATION = 7200

# Specifies the maximum time in seconds to wait for a ping response.

PING_TIMEOUT = 2.0

# Defines the size of the ICMP packet payload in bytes.

PING_SIZE = 64

# A small constant added to the denominator in the PF metric
calculation to prevent division by zero.

EPSILON = 1e-3

# A list of dictionaries, each defining a target device with its IP
address and a unique name.

TARGET_DEVICES = [

    {"ip": "192.168.1.140", "name": "device_1"},

    {"ip": "192.168.1.141", "name": "device_2"},

    {"ip": "192.168.1.142", "name": "device_3"},

    {"ip": "192.168.1.143", "name": "device_4"}

]

# Generates a unique filename for the CSV output using the current
timestamp.

TIMESTAMP = datetime.now().strftime("%Y%m%d_%H%M%S")

CSV_FILE = f"pf_wlan_qos_metrics_{TIMESTAMP}.csv"

# Configures the logging system to output messages to both the console
and a file.

# The log format includes a timestamp, log level, and the message
itself.

logging.basicConfig(

    level=logging.INFO,

```

```

format="[% (asctime)s] %(levelname)s: %(message)s",

datefmt="%H:%M:%S",

handlers=[

    logging.StreamHandler(),

    logging.FileHandler(f"pf_wlan_log_{TIMESTAMP}.log")

]

)

# Creates a logger instance for use throughout the script.

logger = logging.getLogger(__name__)

# -----

# Device Class Definition

# -----

class Device:

    """

    Represents a single network device being monitored. This class
    encapsulates

    all state information and performance metrics associated with the
    device.

    """

    def __init__(self, ip: str, name: str):

        """

        Initialises a Device instance with its network details and
        metric trackers.

        Args:

            ip (str): The IP address of the device.

            name (str): A unique identifier for the device.

        """

        self.ip = ip

        self.name = name

        # Counters for sent, received, and lost packets.

        self.packets_sent = 0

```

```

        self.packets_received = 0

        self.packets_lost = 0

        # Timestamps to track activity and calculate inter-arrival
times.

        self.first_ping_time: float | None = None

        self.last_ping_time: float | None = None

        self.last_selection_time: float | None = None

        # Performance metrics for delay (round-trip time).

        self.delays: list[float] = []

        self.last_delay: float | None = None

        self.avg_delay: float | None = None

        self.min_delay: float | None = None

        self.max_delay: float | None = None

        # Throughput metrics calculated in bits per second (bps).

        self.instantaneous_throughput_bps = 0.0

        self.average_throughput_bps = 0.0

        # The weight for the exponential moving average (EMA)
calculation of throughput.

        self.alpha = 0.2

        # The Proportional Fair (PF) metric. Initialized to a high
value to

        # prioritize scheduling for new or previously unscheduled
devices.

        self.pf_metric = 1000.0

        # A dictionary to store the history of key metrics for
analysis.

        self.history = {

```

```

        "timestamps": [],

        "delays": [],

        "throughputs_bps": [],

        "pf_metrics": [],

        "selected": []

    }

    def update_metrics(self, success: bool, delay: float | None,
current_time: float, was_selected: bool) -> dict | None:

        """

        Updates the device's performance metrics based on the result
of a ping attempt.

        This method is called for every device in each round, but
calculations are

            only performed if the device was the one selected for the ping.

        Args:

            success (bool): True if the ping was successful, False
otherwise.

            delay (float | None): The measured round-trip time in
milliseconds.

            current_time (float): The timestamp of the current
scheduling round.

            was_selected (bool): True if this device was selected for
the ping.

        Returns:

            dict | None: A dictionary of the latest metrics if the
device was

                selected, otherwise None.

        """

        # Record the selection status and timestamp for this round.

        self.history["selected"].append(was_selected)

        self.history["timestamps"].append(current_time)

        # Only update detailed metrics if this device was the one
pinged.

```



```

        if was_selected:

            self.packets_sent += 1

            self.last_selection_time = current_time

            # Record the time of the first ping attempt.
            if self.first_ping_time is None:

                self.first_ping_time = current_time

            if success and delay is not None:

                self.packets_received += 1

                # Update delay statistics.
                self.delays.append(delay)

                # Maintain a moving window of the last 10 delay values.
                if len(self.delays) > 10:

                    self.delays.pop(0)

                self.last_delay = delay

                self.avg_delay = statistics.mean(self.delays)

                self.min_delay = min(self.delays)

                self.max_delay = max(self.delays)

                # Calculate jitter as the standard deviation of recent
delays.

                jitter = statistics.stdev(self.delays) if
len(self.delays) >= 2 else 0.0

                # Calculate instantaneous throughput in bits per
second.

                # Formula: (Payload Size in Bytes * 8 bits/byte * 1000
ms/s) / Delay in ms

                instantaneous_rate = (PING_SIZE * 8 * 1000) / delay if
delay > 0 else 0.0

```

```

else:

    self.packets_lost += 1

    jitter = None

    instantaneous_rate = 0.0

    # Update the average throughput using an exponential
moving average (EMA).

    if self.average_throughput_bps == 0:

        self.average_throughput_bps = instantaneous_rate

    else:

        self.average_throughput_bps = (self.alpha *
instantaneous_rate +
(1 - self.alpha) *
self.average_throughput_bps)

        self.instantaneous_throughput_bps = instantaneous_rate

    # Update the Proportional Fair metric.

    if self.average_throughput_bps > 0:

        self.pf_metric = instantaneous_rate /
(self.average_throughput_bps + EPSILON)

    else:

        # Assign a high PF metric if the average throughput is
zero to prioritize selection.

        self.pf_metric = 1000.0

    # Store historical data for this round.

self.history["throughputs_bps"].append(self.instantaneous_throughput
_bps)

    self.history["pf_metrics"].append(self.pf_metric)

    self.history["delays"].append(delay if success else None)

```

```

        # Calculate the current packet loss rate.

        loss_rate = self.packets_lost / self.packets_sent if
self.packets_sent > 0 else 0.0

        # Calculate the time elapsed since the last ping to this
device.

        inter_arrival = current_time - self.last_ping_time if
self.last_ping_time is not None else None

        self.last_ping_time = current_time

        # Return a dictionary of the newly calculated metrics for
logging.

        return {

            "delay_ms": delay if success else None,

            "jitter_ms": jitter,

            "inter_arrival_s": inter_arrival,

            "loss_rate": loss_rate,

            "inst_throughput_bps":
self.instantaneous_throughput_bps,

            "avg_throughput_bps": self.average_throughput_bps,

            "pf_metric": self.pf_metric

        }

    else:

        # For non-selected devices, carry over old metric values
in the history.

        self.history["delays"].append(None)

self.history["throughputs_bps"].append(self.instantaneous_throughput
_bps)

        self.history["pf_metrics"].append(self.pf_metric)

        return None

# -----
# Proportional Fair Scheduler

```

```

# -----

class ProportionalFairScheduler:

    """

    Implements the Proportional Fair (PF) scheduling algorithm.

    The scheduler selects the device with the highest PF metric in
    each round,

    balancing resource allocation between high-throughput and low-
    throughput devices.

    """

    def __init__(self, devices: list[Device]):

        """

        Initializes the scheduler with a list of devices to manage.

        Args:

            devices (list[Device]): The list of Device objects to
            schedule.

        """

        self.devices = devices

        self.schedule_history: list[str] = []

        self.round_number = 0

    def select_device(self) -> Device | None:

        """

        Selects a device based on the highest Proportional Fair (PF)
        metric.

        If multiple devices share the highest metric, one is chosen
        randomly

        from the candidates to ensure fairness and prevent starvation.

        Returns:

            Device | None: The selected device instance, or None if no
            devices are available.

        """

        self.round_number += 1

```

```

        # Collect the current PF metric from each device.

        device_metrics = [(device, device.pf_metric) for device in
self.devices]

        # Sort devices in descending order based on their PF metric.

        device_metrics.sort(key=lambda x: x[1], reverse=True)

        # Identify all devices that have the maximum PF metric.

        max_metric = device_metrics[0][1] if device_metrics else 0

        candidates = [device for device, metric in device_metrics if
metric == max_metric]

        # Randomly choose one device from the candidates to break ties.

        selected = random.choice(candidates) if candidates else None

        if selected:

            logger.info(f"Round      {self.round_number}:      Selected
{selected.name}      ({selected.ip})      with      PF      metric
{selected.pf_metric:.2f}")

            self.schedule_history.append(selected.name)

        return selected

# -----
# Main QoS Collection Loop
# -----

def main() -> None:

    """

    The main function that orchestrates the experiment.

    It initializes devices and the scheduler, then enters a loop to
perform

    scheduling, pinging, metric collection, and data logging.

```

```

"""

# Define the headers for the CSV file. These are chosen to be clear
and

# suitable for analysis with tools like pandas or scikit-learn.

fieldnames = [

    "timestamp",

    "device_name",

    "device_ip",

    "round_id",

    "delay_ms",

    "jitter_ms",

    "inter_arrival_s",

    "packets_sent",

    "packets_received",

    "packets_lost",

    "loss_rate",

    "inst_throughput_bps",

    "avg_throughput_bps",

    "pf_metric",

    "is_selected"

]


# Create a data directory if it doesn't exist and prepare the CSV
file path.

os.makedirs("data", exist_ok=True)

csv_path = os.path.join("data", CSV_FILE)


# Open the CSV file and write the header row.

with open(csv_path, mode='w', newline='') as file:

    writer = csv.DictWriter(file, fieldnames=fieldnames)

```

```

        writer.writeheader()

        # Create a list of Device objects from the configuration.

        devices = [Device(dev["ip"], dev["name"]) for dev in
TARGET_DEVICES]

        # Initialize the Proportional Fair scheduler with the list of
devices.

        scheduler = ProportionalFairScheduler(devices)

        logger.info(f"Starting QoS metrics collection with PF scheduling
for {len(devices)} devices")

        logger.info(f"Data will be saved to {csv_path}")

        start_time = time.time()

        round_id = 0

    try:

        # The main experiment loop.

        while True:

            round_id += 1

            current_time = time.time()

            # Check if the configured experiment duration has been
exceeded.

            if EXPERIMENT_DURATION > 0 and (current_time - start_time)
> EXPERIMENT_DURATION:

                logger.info(f"Experiment                                duration
({EXPERIMENT_DURATION}s) reached. Stopping.")

                break

            # Use the scheduler to select the next device to ping.

            selected_device = scheduler.select_device()

            if not selected_device:

```

```

        logger.warning("No device selected by the scheduler.")

        time.sleep(TRANSMISSION_INTERVAL)

        continue

    # Perform the ICMP ping on the selected device.
    try:
        result = ping(
            selected_device.ip,
            count=1,
            timeout=PING_TIMEOUT,
            interval=0.2,
            payload=b"X" * PING_SIZE, # Use a bytes payload
            privileged=False
        )
        success = result.is_alive
        delay = result.avg_rtt if success else None # Extract
        delay in ms.

    except Exception as e:
        logger.error(f"An exception occurred during ping to
        {selected_device.ip}: {e}")
        success = False
        delay = None

    # Update metrics for all devices and prepare data for CSV
    logging.

    timestamp_str =
    datetime.fromtimestamp(current_time).isoformat()

    rows = []

    for device in devices:
        is_selected = (device == selected_device)

```



```

        updated_metrics = device.update_metrics(

            success if is_selected else False,

            delay if is_selected else None,

            current_time,

            is_selected

        )

    # Construct a dictionary representing a row in the CSV
file.

    row = {

        "timestamp": timestamp_str,

        "device_name": device.name,

        "device_ip": device.ip,

        "round_id": round_id,

        "packets_sent": device.packets_sent,

        "packets_received": device.packets_received,

        "packets_lost": device.packets_lost,

        "is_selected": 1 if is_selected else 0

    }

    # If the device was selected and pinged, add its
detailed metrics.

    if is_selected and updated_metrics:

        row.update({

            "delay_ms":
round(updated_metrics["delay_ms"], 2) if updated_metrics["delay_ms"]
is not None else None,

            "jitter_ms":
round(updated_metrics["jitter_ms"], 2) if
updated_metrics["jitter_ms"] is not None else None,

            "inter_arrival_s":
round(updated_metrics["inter_arrival_s"], 2) if
updated_metrics["inter_arrival_s"] is not None else None,

```

```

        "loss_rate":
round(updated_metrics["loss_rate"], 4),

        "inst_throughput_bps":
round(updated_metrics["inst_throughput_bps"], 4),

        "avg_throughput_bps":
round(updated_metrics["avg_throughput_bps"], 4),

        "pf_metric":
round(updated_metrics["pf_metric"], 4)

    })

    else:

        # For non-selected devices, log their last-known
state and clear ping-related fields.

        row.update({

            "delay_ms": None,

            "jitter_ms": None,

            "inter_arrival_s": None,

            "loss_rate": None,

            "inst_throughput_bps":
round(device.instantaneous_throughput_bps, 4),

            "avg_throughput_bps":
round(device.average_throughput_bps, 4),

            "pf_metric": round(device.pf_metric, 4)

        })

    rows.append(row)

# Append the collected data for this round to the CSV file.
with open(csv_path, mode='a', newline='') as file:

    writer = csv.DictWriter(file, fieldnames=fieldnames)

    for row in rows:

        writer.writerow(row)

# Log a summary of device status every 10 rounds.

```

```

        if round_id % 10 == 0:

            summary = ", ".join([f"{d.name}: {d.packets_received}/{d.packets_sent} pkts, PF={d.pf_metric:.2f}" for d in devices])

            logger.info(f"Completed {round_id} rounds. Devices: {summary}")

            # Pause before starting the next scheduling round.

            time.sleep(TRANSMISSION_INTERVAL)

    except KeyboardInterrupt:

        logger.info("Experiment stopped by user (KeyboardInterrupt).")

    # After the loop ends, print a final summary of the experiment.
    total_duration = time.time() - start_time

    logger.info("\n--- Experiment Summary ---")

    logger.info(f"Total duration: {total_duration:.1f} seconds")

    logger.info(f"Total scheduling rounds: {round_id}")

    for device in devices:

        success_rate = (device.packets_received / device.packets_sent * 100) if device.packets_sent > 0 else 0.0

        logger.info(f"Device: {device.name} ({device.ip})")

        logger.info(f"Packet Stats: {device.packets_received} received / {device.packets_sent} sent ({success_rate:.1f}% success)")

        logger.info(f"Final Avg Throughput: {device.average_throughput_bps:.2f} bps")

        logger.info(f"Final PF Metric: {device.pf_metric:.4f}")

        if device.delays:

            logger.info(f"Final Delay (ms): avg={statistics.mean(device.delays):.2f}, min={min(device.delays):.2f}, max={max(device.delays):.2f}")

```

```

        logger.info(f"All data has been saved to {csv_path}")

# This standard entry point ensures that the main() function is called
only when

# the script is executed directly.

if __name__ == "__main__":

    main()

```

Appendix E.2: Source Code for Round Robin Scheduler

```

#!/usr/bin/env python3

"""

Enhanced QoS Metrics Collection with Round Robin Scheduling for WLAN

This script demonstrates Round Robin scheduling by pinging multiple
target devices

in a WLAN environment using icmpLib. It collects comprehensive QoS
metrics including

delay, jitter, packet loss, and throughput, storing them in a machine
learning-friendly

CSV format for later analysis.

The scheduler cycles through devices in order, ensuring each device is
served in turn.

"""

import time

import csv

import random

import logging

import statistics

from datetime import datetime

from icmpLib import ping

import os

# -----

# Configuration and Logging Setup

```

```

# -----

# Defines the time in seconds to wait between each scheduling decision.

TRANSMISSION_INTERVAL = 0.5

# Sets the total duration of the experiment in seconds. A value of 0
runs indefinitely.

EXPERIMENT_DURATION = 7200

# Specifies the maximum time in seconds to wait for a ping response.

PING_TIMEOUT = 2.0

# Defines the payload size in bytes for each ICMP ping packet.

PING_SIZE = 64

# A list of dictionaries, where each dictionary represents a target
device

# with its IP address and a unique name for identification.

TARGET_DEVICES = [

    {"ip": "192.168.1.140", "name": "device_1"},

    {"ip": "192.168.1.141", "name": "device_2"},

    {"ip": "192.168.1.142", "name": "device_3"},

    {"ip": "192.168.1.143", "name": "device_4"}

]

# Generates a unique timestamp string for file naming to prevent
overwriting results.

TIMESTAMP = datetime.now().strftime("%Y%m%d_%H%M%S")

# Defines the CSV filename, incorporating the timestamp for uniqueness.

CSV_FILE = f"rr_wlan_qos_metrics_{TIMESTAMP}.csv"

# Configures the logging system to output messages to both the console
and a log file.

# The log file is also named with a unique timestamp.

logging.basicConfig(

    level=logging.INFO,

    format="[% (asctime)s] %(levelname)s: %(message)s",

    datefmt="%H:%M:%S",

```

```

handlers=[

    logging.StreamHandler(),

    logging.FileHandler(f"rr_wlan_log_{TIMESTAMP}.log")

]

)

# Creates a logger instance for use throughout the script.
logger = logging.getLogger(__name__)

# -----

# Device Class Definition

# -----

class Device:

    """

    Represents a target network device and manages its associated
    Quality of Service (QoS)

    metrics, history, and state over the course of the experiment.

    """

    def __init__(self, ip: str, name: str):

        # --- Basic Device Information ---

        self.ip = ip      # IP address of the device.

        self.name = name  # A unique, human-readable name for the
device.

        # --- Packet Counters ---

        self.packets_sent = 0      # Total number of pings sent to
this device.

        self.packets_received = 0  # Total number of successful ping
responses.

        self.packets_lost = 0      # Total number of lost or failed
pings.

        # --- Timestamps for Tracking ---

```

```

        self.first_ping_time = None          # Timestamp of the first ping
attempt.

        self.last_ping_time = None          # Timestamp of the most recent
ping attempt.

        self.last_selection_time = None     # Timestamp of when this
device was last selected.


    # --- Real-time Performance Metrics ---

        self.delays = []                    # A list to store recent delay
values (in ms) for jitter calculation.

        self.last_delay = None              # The delay of the most recent
successful ping.

        self.avg_delay = None               # The average delay calculated
from the 'delays' list.

        self.min_delay = None               # The minimum delay in the
'delays' list.

        self.max_delay = None               # The maximum delay in the
'delays' list.


    # --- Throughput Metrics (in bits per second) ---

        self.instantaneous_throughput_bps = 0.0    # Throughput
calculated from the last successful ping.

        self.average_throughput_bps = 0.0          # Exponential moving
average of throughput.

        self.alpha = 0.2                          # Smoothing factor
for the exponential moving average.


    # --- Historical Data for Analysis ---

        self.history = {

            "timestamps": [],                    # Record of all timestamps when
this device was considered.

            "delays": [],                       # Record of delay values (None if
not selected or ping failed).

            "throughputs_bps": [],              # Record of instantaneous
throughput values.

            "selected": []                      # Boolean record of whether the
device was selected in a given round.

```

```

    }

    def update_metrics(self, success: bool, delay: float | None,
current_time: float, was_selected: bool) -> dict | None:

        """

        Updates the device's performance metrics based on the result
of a ping attempt.

        This method is called for every device in every round, but it
only updates detailed

        metrics if the device was the one selected for pinging
(`was_selected` is True).

        Args:

            success: A boolean indicating if the ping was successful.

            delay: The round-trip time in milliseconds if the ping was
successful.

            current_time: The timestamp of the current processing
round.

            was_selected: A boolean indicating if this device was
selected for the ping.

        Returns:

            A dictionary of the latest calculated metrics if the device
was selected,

            otherwise None.

        """

        # Record the selection status and timestamp for historical
analysis.

        self.history["selected"].append(was_selected)

        self.history["timestamps"].append(current_time)

        # Only perform detailed metric updates for the device that was
actually pinged.

        if was_selected:

            self.packets_sent += 1

            self.last_selection_time = current_time

```



```

        # Set the timestamp for the very first ping attempt.

        if self.first_ping_time is None:

            self.first_ping_time = current_time

        if success:

            self.packets_received += 1

        # Update delay-related metrics if a valid delay was
provided.

        if delay is not None:

            self.delays.append(delay)

            # Maintain a rolling window of the last 10 delays.

            if len(self.delays) > 10:

                self.delays.pop(0)

            self.last_delay = delay

            self.avg_delay = statistics.mean(self.delays)

            self.min_delay = min(self.delays)

            self.max_delay = max(self.delays)

        # Jitter is calculated as the standard deviation of
recent delays.

        # Requires at least two data points to be meaningful.

        jitter = statistics.stdev(self.delays) if
len(self.delays) >= 2 else 0.0

        # Instantaneous throughput is calculated based on
payload size and RTT.

        # Formula: (PayloadSize * 8 bits/byte * 1000 ms/s) /
RTT_ms

        instantaneous_rate = (PING_SIZE * 8 * 1000) / delay if
delay and delay > 0 else 0.0

        else:

```

```

        # If the ping failed, update the lost packet count and
        reset relevant metrics.

        self.packets_lost += 1

        jitter = None

        instantaneous_rate = 0.0

    # Update the average throughput using an exponential
    moving average (EMA)

    # to smooth out fluctuations.

    if self.average_throughput_bps == 0:

        self.average_throughput_bps = instantaneous_rate

    else:

        self.average_throughput_bps = (self.alpha *
        instantaneous_rate +
                                         (1 - self.alpha) *
        self.average_throughput_bps)

        self.instantaneous_throughput_bps = instantaneous_rate

    # Store the results of this round in the device's history.

    self.history["throughputs_bps"].append(self.instantaneous_throughput
    _bps)

    self.history["delays"].append(delay if success else None)

    # Calculate the current packet loss rate.

    loss_rate = self.packets_lost / self.packets_sent if
    self.packets_sent > 0 else 0.0

    # Calculate the time elapsed since the last ping to this
    specific device.

    inter_arrival = current_time - self.last_ping_time if
    self.last_ping_time is not None else None

    self.last_ping_time = current_time

```

```

        # Return a dictionary of the newly calculated metrics for
logging.

        return {

            "delay_ms": delay if success else None,

            "jitter_ms": jitter,

            "inter_arrival_s": inter_arrival,

            "loss_rate": loss_rate,

            "inst_throughput_bps":
self.instantaneous_throughput_bps,

            "avg_throughput_bps": self.average_throughput_bps

        }

    else:

        # For non-selected devices, append placeholders to history
to maintain data alignment.

        self.history["delays"].append(None)

self.history["throughputs_bps"].append(self.instantaneous_throughput
_bps)

        return None

# -----

# Round Robin Scheduler

# -----

class RoundRobinScheduler:

    """

    Implements a simple Round Robin scheduling algorithm.

    This scheduler cycles through a list of devices sequentially,
ensuring that each

    device gets a turn in a predictable, circular order.

    """

    def __init__(self, devices: list[Device]):

        self.devices = devices

```

```

        self.index = 0          # A pointer to the next device to be
scheduled.

        self.round_number = 0    # A counter for the number of
selections made.

    def select_device(self) -> Device:

        """

        Selects the next device from the list in a round-robin fashion.

        Returns:

            The device object that has been selected for the current
round.

        """

        self.round_number += 1

        # Get the device at the current index.

        device = self.devices[self.index]

        logger.info(f"Round      {self.round_number}:      Selected
{device.name} ({device.ip})")

        # Increment the index for the next round, wrapping around to
the start if necessary.

        self.index = (self.index + 1) % len(self.devices)

        return device

# -----

# Main QoS Collection Loop

# -----

def main() -> None:

    """

    The main execution function of the script.

    It initializes the devices and scheduler, sets up the CSV output
file,

    and runs the main loop to collect QoS metrics until the experiment
duration is met or it is manually stopped.

    """

    # Define the headers for the output CSV file. These are structured
for easy

```

```

    # import into data analysis tools like pandas or for machine
    learning.

    fieldnames = [

        "timestamp",    "device_name",    "device_ip",    "round_id",
        "delay_ms",

        "jitter_ms",        "inter_arrival_s",        "packets_sent",
        "packets_received",

        "packets_lost",    "loss_rate",    "inst_throughput_bps",
        "avg_throughput_bps",

        "is_selected"

    ]

    # Create a 'data' directory if it doesn't exist and prepare the
    CSV file path.

    os.makedirs("data", exist_ok=True)

    # Construct the full path for the CSV file.

    csv_path = os.path.join("data", CSV_FILE)

    # Open the CSV file in write mode and write the header row.

    with open(csv_path, mode='w', newline='') as file:

        writer = csv.DictWriter(file, fieldnames=fieldnames)

        writer.writeheader()

    # Create a list of Device objects from the configuration.

    devices = [Device(dev["ip"], dev["name"]) for dev in
TARGET_DEVICES]

    # Instantiate the Round Robin scheduler with the list of devices.

    scheduler = RoundRobinScheduler(devices)

    logger.info(f"Starting QoS metrics collection with Round Robin
scheduling for {len(devices)} devices")

    logger.info(f"Data will be saved to {csv_path}")

    start_time = time.time()

```

```

round_id = 0

try:
    # The main loop continues until the duration is reached or
    interrupted.

    while True:

        round_id += 1

        current_time = time.time()

        # Check if the configured experiment duration has been
        exceeded.

        if EXPERIMENT_DURATION > 0 and (current_time - start_time)
        > EXPERIMENT_DURATION:

            logger.info(f"Experiment                                duration
            ({EXPERIMENT_DURATION}s) reached. Stopping.")

            break

        # Use the scheduler to select the next device to ping.

        selected_device = scheduler.select_device()

        # Attempt to ping the selected device.

        try:

            # Use icmpLib to send a single ping with a specified
            timeout and payload.

            result = ping(

                selected_device.ip,

                count=1,

                timeout=PING_TIMEOUT,

                interval=0.2,

                payload=b"X" * PING_SIZE,

                privileged=False

            )

```

```

        success = result.is_alive

        delay = result.avg_rtt if success else None # Extract
RTT in milliseconds.

    except Exception as e:

        # Log any errors that occur during the ping process.

        logger.error(f"Ping          exception          for
{selected_device.ip}: {e}")

        success = False

        delay = None

    # Prepare to update metrics and write data for all devices
for this round.

    timestamp_str          =
datetime.fromtimestamp(current_time).isoformat()

    rows = []

    # Iterate through all devices to update their state and
create a CSV row.

    for device in devices:

        is_selected = (device == selected_device)

        # Update metrics. Only the selected device will have
new ping results.

        updated_metrics = device.update_metrics(

            success if is_selected else False,

            delay if is_selected else None,

            current_time,

            is_selected

        )

        # Construct a dictionary representing a row in the CSV
file.

        row = {

```

```

        "timestamp": timestamp_str,

        "device_name": device.name,

        "device_ip": device.ip,

        "round_id": round_id,

        "packets_sent": device.packets_sent,

        "packets_received": device.packets_received,

        "packets_lost": device.packets_lost,

        "is_selected": 1 if is_selected else 0

    }

    # If this device was selected and pinged, add its
    specific metrics to the row.

    if is_selected and updated_metrics:

        row.update({

            "delay_ms":
round(updated_metrics["delay_ms"], 2) if updated_metrics["delay_ms"]
is not None else None,

            "jitter_ms":
round(updated_metrics["jitter_ms"],          2)                if
updated_metrics["jitter_ms"] is not None else None,

            "inter_arrival_s":
round(updated_metrics["inter_arrival_s"],          2)                if
updated_metrics["inter_arrival_s"] is not None else None,

            "loss_rate":
round(updated_metrics["loss_rate"], 4),

            "inst_throughput_bps":
round(updated_metrics["inst_throughput_bps"], 4),

            "avg_throughput_bps":
round(updated_metrics["avg_throughput_bps"], 4)

        })

    else:

        # For non-selected devices, fill metric fields
        with None or last known values.

        row.update({

            "delay_ms": None,

```



```

        "jitter_ms": None,

        "inter_arrival_s": None,

        "loss_rate": None,

        "inst_throughput_bps":
round(device.instantaneous_throughput_bps, 4),

        "avg_throughput_bps":
round(device.average_throughput_bps, 4)

    })

    rows.append(row)

# Append the data for all devices for this round to the
CSV file.

    with open(csv_path, mode='a', newline='') as file:

        writer = csv.DictWriter(file, fieldnames=fieldnames)

        for row in rows:

            writer.writerow(row)

# Log a summary message every 10 rounds to provide progress
feedback.

    if round_id % 10 == 0:

        summary = ", ".join([f"{d.name}: {d.packets_received}/{d.packets_sent} pkts" for d in devices])

        logger.info(f"Completed {round_id} rounds. Devices: {summary}")

# Pause execution for the configured interval before
starting the next round.

    time.sleep(TRANSMISSION_INTERVAL)

except KeyboardInterrupt:

    # Handle manual interruption (Ctrl+C) gracefully.

    logger.info("Experiment stopped by user.")

```

```

# Calculate and log a final summary of the experiment results.

total_duration = time.time() - start_time

logger.info("\n--- Experiment Summary ---")

logger.info(f"Duration: {total_duration:.1f} seconds")

logger.info(f"Total rounds: {round_id}")


for device in devices:

    success_rate = (device.packets_received / device.packets_sent
* 100) if device.packets_sent > 0 else 0.0

    logger.info(f"{device.name} ({device.ip}):")

    logger.info(f"  Packets: {device.packets_received} received /
{device.packets_sent} sent ({success_rate:.1f}%)")

    logger.info(f"                                Avg                throughput:
{device.average_throughput_bps:.2f} bps")

    if device.delays:

        logger.info(f"                                Delay:
avg={statistics.mean(device.delays):.2f}ms,
min={min(device.delays):.2f}ms, max={max(device.delays):.2f}ms")

    logger.info(f"Data saved to {csv_path}")

# Standard entry point for a Python script.

if __name__ == "__main__":

    main()

```

Appendix F: Source Code for Experiment 3 (Comparison of Scheduling Algorithms using DL Models with Data Captured from Operational Networks)

The implementation presented in this appendix corresponds to the LSTM model developed for scheduling classification tasks, as introduced in *Section 4.7*, evaluated in *Section 5.4.3*, and discussed in the context of broader implications in *Section 6.4*. The source code reflects the model architecture and training procedures used to classify network traffic based on underlying scheduling disciplines.

Appendix F.1: Source Code for the LSTM Model for Scheduling Classification

```
# =====  
  
# 1. Import Necessary Libraries  
  
# =====  
  
import numpy as np  
  
import pandas as pd  
  
import torch  
  
import torch.nn as nn  
  
import torch.optim as optim  
  
from sklearn.model_selection import train_test_split  
  
from sklearn.metrics import (  
    confusion_matrix, classification_report, accuracy_score,  
    precision_score, recall_score, f1_score, roc_curve, auc,  
    precision_recall_curve, average_precision_score  
)  
  
from sklearn.preprocessing import StandardScaler, OneHotEncoder,  
LabelEncoder  
  
from sklearn.compose import ColumnTransformer  
  
from sklearn.pipeline import Pipeline  
  
from sklearn.impute import SimpleImputer  
  
from sklearn.utils.class_weight import compute_class_weight  
  
from sklearn.base import BaseEstimator, TransformerMixin  
  
import matplotlib.pyplot as plt  
  
import seaborn as sns  
  
import os
```

```

# =====

# 2. Device Configuration

# =====

# Check if CUDA is available and set device accordingly

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

print(f'Using device: {device}')

# =====

# 3. Custom Transformer: TopNCategoriesTransformer

# =====

class TopNCategoriesTransformer(BaseEstimator, TransformerMixin):

    """

    Transformer that limits categorical features to the top N
    categories.

    Less frequent categories are grouped into an 'Other' category.

    """

    def __init__(self, top_n=10):

        self.top_n = top_n

        self.top_categories = {}

    def fit(self, X, y=None):

        for col in X.columns:

            top_categories = X[col].value_counts().nlargest(self.top_n).index

            self.top_categories[col] = set(top_categories)

        return self

```

```

def transform(self, X):

    X_transformed = X.copy()

    for col in X.columns:

        X_transformed[col] = X_transformed[col].apply(lambda x: x
if x in self.top_categories[col] else 'Other')

    return X_transformed

# =====

# 4. Load and Prepare the Datasets

# =====

# Define file paths

file_prio = 'PRIO.csv'

file_htb = 'HTB.csv'

file_cbq = 'CBQ.csv'

# Verify that all files exist

for file in [file_prio, file_htb, file_cbq]:

    if not os.path.exists(file):

        raise FileNotFoundError(f"The file '{file}' does not exist in
the current directory.")

# Load datasets

print("Loading datasets...")

data_prio = pd.read_csv(file_prio)

data_htb = pd.read_csv(file_htb)

data_cbq = pd.read_csv(file_cbq)

print("Datasets loaded successfully.")

# Assign labels

data_prio['Source'] = 'PRIO'

```

```

data_htb['Source'] = 'HTB'

data_cbq['Source'] = 'CBQ'

# Combine datasets

combined_data = pd.concat([data_prio, data_htb, data_cbq],
                           ignore_index=True)

print(f"Combined dataset shape: {combined_data.shape}")

# =====

# 5. Identify Feature Types

# =====

# Identify numerical and categorical features

numerical_features = combined_data.select_dtypes(include=['int64',
                                                         'float64']).columns.tolist()

categorical_features = combined_data.select_dtypes(include=['object',
                                                            'category']).columns.tolist()

# Remove the target column from features if present

if 'Source' in numerical_features:

    numerical_features.remove('Source')

if 'Source' in categorical_features:

    categorical_features.remove('Source')

print(f"Numerical features ({len(numerical_features)}):
{numerical_features}")

print(f"Categorical features ({len(categorical_features)}):
{categorical_features}")

# =====

# 6. Preprocessing Pipeline

# =====

# Define the number of top categories to keep

TOP_N = 10

```

```

# Create pipelines for numerical and categorical features

numeric_transformer = Pipeline(steps=[

    ('imputer', SimpleImputer(strategy='mean')),

    ('scaler', StandardScaler())

])

categorical_transformer = Pipeline(steps=[

    ('topn', TopNCategoriesTransformer(top_n=TOP_N)),

    ('imputer', SimpleImputer(strategy='most_frequent')),

    ('onehot', OneHotEncoder(handle_unknown='ignore'))

])

# Combine preprocessing steps

preprocessor = ColumnTransformer(

    transformers=[

        ('num', numeric_transformer, numerical_features),

        ('cat', categorical_transformer, categorical_features)

    ]

)

# =====

# 7. Encode Target Labels

# =====

# Initialize LabelEncoder

le = LabelEncoder()

combined_data['Source_encoded'] =
le.fit_transform(combined_data['Source'])

# Display label encoding mapping

label_mapping = dict(zip(le.classes_, le.transform(le.classes_)))

```

```

print(f"Label encoding: {label_mapping}")

# =====

# 8. Split the Data into Training, Validation, and Testing Sets

# =====

# Features and target

X = combined_data.drop(['Source', 'Source_encoded'], axis=1)

y = combined_data['Source_encoded']

# Split into train (60%) and temp (40%)

X_train, X_temp, y_train, y_temp = train_test_split(

    X, y, test_size=0.4, random_state=42, stratify=y

)

# Split temp into validation (20%) and test (20%)

X_val, X_test, y_val, y_test = train_test_split(

    X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp

)

print(f"Training set shape: {X_train.shape}")

print(f"Validation set shape: {X_val.shape}")

print(f"Test set shape: {X_test.shape}")

# =====

# 9. Apply Preprocessing

# =====

print("Applying preprocessing...")

X_train_processed = preprocessor.fit_transform(X_train)

X_val_processed = preprocessor.transform(X_val)

X_test_processed = preprocessor.transform(X_test)

```



```

print("Preprocessing completed.")

# Convert to dense arrays if necessary
if hasattr(X_train_processed, "toarray"):

    X_train_processed = X_train_processed.toarray()

if hasattr(X_val_processed, "toarray"):

    X_val_processed = X_val_processed.toarray()

if hasattr(X_test_processed, "toarray"):

    X_test_processed = X_test_processed.toarray()

print(f"Processed Training set shape: {X_train_processed.shape}")
print(f"Processed Validation set shape: {X_val_processed.shape}")
print(f"Processed Test set shape: {X_test_processed.shape}")

# =====

# 10. Compute Class Weights

# =====

print("Computing class weights...")

class_weights = compute_class_weight(

    class_weight='balanced',

    classes=np.unique(y_train),

    y=y_train

)

class_weights = torch.tensor(class_weights,
dtype=torch.float).to(device)

print(f"Class weights: {class_weights}")

# =====

# 11. Convert Data to PyTorch Tensors

# =====

```

```

print("Converting data to PyTorch tensors...")

# Convert to numpy arrays

X_train_np = X_train_processed.astype(np.float32)

X_val_np = X_val_processed.astype(np.float32)

X_test_np = X_test_processed.astype(np.float32)

# Corrected the following lines by replacing np.long with int

y_train_np = y_train.values.astype(int)

y_val_np = y_val.values.astype(int)

y_test_np = y_test.values.astype(int)

# Convert to PyTorch tensors

X_train_tensor = torch.tensor(X_train_np).to(device)

y_train_tensor = torch.tensor(y_train_np).to(device)

X_val_tensor = torch.tensor(X_val_np).to(device)

y_val_tensor = torch.tensor(y_val_np).to(device)

X_test_tensor = torch.tensor(X_test_np).to(device)

y_test_tensor = torch.tensor(y_test_np).to(device)

# Create TensorDatasets

train_dataset = torch.utils.data.TensorDataset(X_train_tensor,
y_train_tensor)

val_dataset = torch.utils.data.TensorDataset(X_val_tensor,
y_val_tensor)

test_dataset = torch.utils.data.TensorDataset(X_test_tensor,
y_test_tensor)

# Create DataLoaders

batch_size = 64

train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)

```

```

val_loader      =      torch.utils.data.DataLoader(val_dataset,
batch_size=batch_size, shuffle=False)

test_loader      =      torch.utils.data.DataLoader(test_dataset,
batch_size=batch_size, shuffle=False)

print("Data conversion completed.")

# =====

# 12. Define the LSTM Model

# =====

class LSTMModel(nn.Module):

    def __init__(self,      input_dim,      hidden_dim,      num_layers,
output_dim=3, dropout=0.5):

        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_dim

        self.num_layers = num_layers

        # LSTM Layer

        self.lstm = nn.LSTM(input_dim,      hidden_dim,      num_layers,
batch_first=True, dropout=dropout)

        # Fully Connected Layer

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):

        # Initialize hidden state and cell state

        h0      =      torch.zeros(self.num_layers,      x.size(0),
self.hidden_dim).to(x.device)

        c0      =      torch.zeros(self.num_layers,      x.size(0),
self.hidden_dim).to(x.device)

```

```

        # Forward propagate LSTM

        out, _ = self.lstm(x, (h0, c0)) # out: tensor of shape
(batch_size, seq_length, hidden_dim)


        # Decode the last time step

        out = self.fc(out[:, -1, :])

        return out

# Instantiate the Model

input_dim = X_train_processed.shape[1] # Number of features after
preprocessing

hidden_dim = 128

num_layers = 2

output_dim = len(le.classes_) # Number of classes: PRIO, HTB, CBQ

model = LSTMModel(input_dim, hidden_dim, num_layers,
output_dim).to(device)

print("LSTM model instantiated.")

# =====

# 13. Define Loss Function and Optimizer

# =====

# Define loss function with class weights

criterion = nn.CrossEntropyLoss(weight=class_weights)

# Define optimizer

optimizer = optim.Adam(model.parameters(), lr=0.001)

print("Loss function and optimizer defined.")

# =====

# 14. Training the Model with Early Stopping

```

```

# =====

num_epochs = 30

patience = 5

best_val_loss = float('inf')

epochs_no_improve = 0

# Lists to store losses and accuracies

train_losses = []

val_losses = []

train_accuracies = []

val_accuracies = []

print("Starting training...")

for epoch in range(num_epochs):

    # Training Phase

    model.train()

    running_loss = 0.0

    correct = 0

    total = 0

    for inputs, labels in train_loader:

        # Reshape inputs to add sequence dimension (batch_size,
        seq_length=1, input_dim)

        inputs = inputs.unsqueeze(1)

        # Forward pass

        outputs = model(inputs)

        loss = criterion(outputs, labels)

```

```

        # Backward and optimize

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()

    # Accumulate loss

    running_loss += loss.item() * inputs.size(0)

    # Calculate accuracy

    _, predicted = torch.max(outputs.data, 1)

    total += labels.size(0)

    correct += (predicted == labels).sum().item()

epoch_train_loss = running_loss / len(train_loader.dataset)

epoch_train_acc = correct / total

train_losses.append(epoch_train_loss)

train_accuracies.append(epoch_train_acc)

# Validation Phase

model.eval()

val_running_loss = 0.0

val_correct = 0

val_total = 0

```

```

with torch.no_grad():

    for inputs, labels in val_loader:

        # Reshape inputs to add sequence dimension

        inputs = inputs.unsqueeze(1)

        outputs = model(inputs)

        loss = criterion(outputs, labels)

        val_running_loss += loss.item() * inputs.size(0)

        # Calculate accuracy

        _, predicted = torch.max(outputs.data, 1)

        val_total += labels.size(0)

        val_correct += (predicted == labels).sum().item()

    epoch_val_loss = val_running_loss / len(val_loader.dataset)

    epoch_val_acc = val_correct / val_total

    val_losses.append(epoch_val_loss)

    val_accuracies.append(epoch_val_acc)

    print(f'Epoch [{epoch+1}/{num_epochs}], '

          f'Train    Loss:    {epoch_train_loss:.4f},    Train    Acc:
{epoch_train_acc:.4f}, '

          f'Val      Loss:    {epoch_val_loss:.4f},    Val      Acc:
{epoch_val_acc:.4f}')

    # Early Stopping Check

    if epoch_val_loss < best_val_loss:

```

```

        best_val_loss = epoch_val_loss

        epochs_no_improve = 0

        # Save the best model

        torch.save(model.state_dict(), 'best_lstm_model.pth')

    else:

        epochs_no_improve += 1

        if epochs_no_improve >= patience:

            print('Early stopping!')

            break

# Load the Best Model

model.load_state_dict(torch.load('best_lstm_model.pth'))

print("Best model loaded.")

# =====

# 15. Plot Training and Validation Losses

# =====

plt.figure(figsize=(10, 5))

plt.plot(train_losses, label='Training Loss', marker='o')

plt.plot(val_losses, label='Validation Loss', marker='o')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Training and Validation Loss')

plt.legend()

plt.grid(True)

plt.show()

# =====

```



```

# 16. Evaluate the Model on the Test Set

# =====

def get_predictions(model, loader):

    """

    Function to get true labels and predictions from a DataLoader.

    """

    model.eval()

    all_preds = []

    all_labels = []

    with torch.no_grad():

        for inputs, labels in loader:

            # Reshape inputs to add sequence dimension

            inputs = inputs.unsqueeze(1)

            outputs = model(inputs)

            _, predicted = torch.max(outputs.data, 1)

            all_preds.extend(predicted.cpu().numpy())

            all_labels.extend(labels.cpu().numpy())

    return all_labels, all_preds

print("Evaluating on test set...")

targets, predictions = get_predictions(model, test_loader)

# Classification Report

print("Classification Report:\n", classification_report(targets,
predictions, target_names=le.classes_))

# Calculate Additional Metrics

accuracy = accuracy_score(targets, predictions)

precision = precision_score(targets, predictions, average='weighted')

```

```

recall = recall_score(targets, predictions, average='weighted')

f1 = f1_score(targets, predictions, average='weighted')

print(f"Accuracy: {accuracy:.4f}")

print(f"Precision (weighted): {precision:.4f}")

print(f"Recall (weighted): {recall:.4f}")

print(f"F1 Score (weighted): {f1:.4f}")

# =====

# 17. Plot Confusion Matrix

# =====

cm = confusion_matrix(targets, predictions)

plt.figure(figsize=(6, 4))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

            xticklabels=le.classes_,

            yticklabels=le.classes_)

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

# =====

# 18. Plot ROC Curves for Each Class

# =====

from sklearn.preprocessing import label_binarize

# Binarize the output for ROC computation

y_test_binarized = label_binarize(targets, classes=np.unique(y))

n_classes = y_test_binarized.shape[1]

```

```

# Compute scores using softmax outputs

def get_softmax_scores(model, loader):

    model.eval()

    all_scores = []

    with torch.no_grad():

        for inputs, _ in loader:

            # Reshape inputs to add sequence dimension

            inputs = inputs.unsqueeze(1)

            outputs = model(inputs)

            scores = torch.softmax(outputs, dim=1)

            all_scores.append(scores.cpu().numpy())

    return np.vstack(all_scores)

scores_roc = get_softmax_scores(model, test_loader)

# Compute ROC curve and ROC area for each class

fpr = dict()

tpr = dict()

roc_auc = dict()

for i in range(n_classes):

    fpr[i], tpr[i], _ = roc_curve(y_test_binarized[:, i],
scores_roc[:, i])

    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot all ROC curves

plt.figure(figsize=(8, 6))

colors = ['blue', 'red', 'green']

for i, color in zip(range(n_classes), colors):

    plt.plot(fpr[i], tpr[i], color=color, lw=2,

```

```

        label=f'ROC curve of class {le.classes_[i]} (area =
{roc_auc[i]:0.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curves')

plt.legend(loc="lower right")

plt.grid(True)

plt.show()

# =====

# 19. Plot Precision-Recall Curves

# =====

# Compute Precision-Recall curve and average precision for each class

precision_dict = {}

recall_dict = {}

average_precision_dict = {}

for i in range(n_classes):

    precision_dict[i], recall_dict[i], _ =
precision_recall_curve(y_test_binarized[:, i], scores_roc[:, i])

    average_precision_dict[i] =
average_precision_score(y_test_binarized[:, i], scores_roc[:, i])

# Plot Precision-Recall curves

plt.figure(figsize=(8, 6))

for i, color in zip(range(n_classes), colors):

    plt.plot(recall_dict[i], precision_dict[i], color=color, lw=2,

```

```

        label=f'PR curve of class {le.classes_[i]} (AP =
{average_precision_dict[i]:0.2f})')

plt.xlabel('Recall')

plt.ylabel('Precision')

plt.title('Precision-Recall Curves')

plt.legend(loc="lower left")

plt.grid(True)

plt.show()

# =====

# 20. Save the Trained Model

# =====

# Save the trained model

torch.save(model.state_dict(), 'lstm_model_with_class_weights.pth')

print("Model saved to 'lstm_model_with_class_weights.pth'")

```

Appendix G: Source Code for Experiment 4 (Comparative Evaluation of Network-Delay Prediction Models)

This appendix provides the source code for the delay prediction models evaluated in Experiment 4, including the hybrid Linear-LSTM-Transformer model, a standalone LSTM model, and an RF baseline. The model designs and implementation strategies are described in *Section 4.8*, with performance evaluations presented in *Section 5.5*, and their implications discussed in *Section 6.5*.

Appendix G.1: Source Code for the Hybrid (Linear+LSTM+Transformer) Model for QoS Prediction

```

"""

Hybrid Time-Series Regressor (Linear-LSTM-Transformer) for delay_ms

- Processes time-series sequences, simplified projection layer.

```

```

- leakage-safe preprocessing, device-wise split, Optuna H-param
search,

    early stopping, LR scheduler

Dataset : pf_wlan_qos_metrics.csv

-----

"""

# — 0 Imports & global settings -----

import random, math, os, warnings

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

plt.rcParams["font.size"] = 16          # global font size

from sklearn.compose      import ColumnTransformer

from sklearn.pipeline     import Pipeline

from sklearn.preprocessing import StandardScaler, OrdinalEncoder

from sklearn.impute       import SimpleImputer

from sklearn.model_selection import GroupShuffleSplit

from sklearn.metrics      import mean_squared_error,
mean_absolute_error, r2_score

import torch

from torch import nn, optim

from torch.utils.data import Dataset, DataLoader, TensorDataset

from torch.optim.lr_scheduler import ReduceLROnPlateau

import optuna

warnings.filterwarnings("ignore", category=UserWarning)

warnings.filterwarnings("ignore", category=FutureWarning) # Ignore
squared=False deprecation

```

```

# — Hyper-parameters & constants —————

SEED                = 42

BATCH_DEFAULT      = 128

EPOCHS_MAX         = 80  # Max epochs for Optuna & final training

PATIENCE           = 8   # Early stopping patience

CSV_PATH           = "pf_wlan_qos_metrics.csv"

USE_PF_METRIC      = False    # set True to include pf_metric in the
feature set

SEQUENCE_LENGTH    = 10      # <<< How many time steps to look back

random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Using device: {DEVICE}")

print(f"Using Sequence Length: {SEQUENCE_LENGTH}")

# —————

# — 1 Load, Preprocess & Sequence Data —————

df = pd.read_csv(CSV_PATH)

df = df.dropna(subset=["delay_ms"]).sort_values(by=['device_name',
'timestamp']).reset_index(drop=True) # Ensure order

TARGET = "delay_ms"

GROUP_COL = "device_name"

CATEG = [GROUP_COL]

NUMERIC = [c for c in df.columns

            if c not in ["timestamp", "device_ip", "round_id", TARGET]
+ CATEG]

if not USE_PF_METRIC and "pf_metric" in NUMERIC:

    NUMERIC.remove("pf_metric")

FEATURES = NUMERIC + CATEG

```

```

NUM_FEATURES_PRE_TRANSFORM = len(FEATURES) # Store this before
transformation

# --- Group-wise split (no device overlap) ---

print("Splitting data by device...")

gss = GroupShuffleSplit(n_splits=1, test_size=0.2, random_state=SEED)

train_idx, val_idx = next(gss.split(df, groups=df[GROUP_COL]))

train_df,      val_df      =      df.iloc[train_idx].copy(),
df.iloc[val_idx].copy() # Use copy to avoid SettingWithCopyWarning

# --- Preprocessing transformers ---

num_pipe = Pipeline([

    ("imp", SimpleImputer(strategy="median")),

    ("sc" , StandardScaler())

])

# Use OrdinalEncoder for device_name as it's only used for grouping
later

cat_pipe = Pipeline([

    ("enc",      OrdinalEncoder(handle_unknown="use_encoded_value",
unknown_value=-1))

])

preproc = ColumnTransformer([

    ("num", num_pipe, NUMERIC),

    ("cat", cat_pipe, CATEG) # Encode device_name too for easy grouping
id

], remainder='passthrough') # Keep other columns if needed (like
target)

# --- Fit on Train, Transform Train & Val ---

print("Preprocessing data...")

# Fit and transform train data

```



```

# Add target column temporarily for alignment, then drop it

train_df['temp_target'] = train_df[TARGET]

X_train_transformed = preproc.fit_transform(train_df[FEATURES +
['temp_target']])

y_train_flat = X_train_transformed[:, -1] # Extract target

X_train_flat = X_train_transformed[:, :-1] # Extract features

train_groups = X_train_flat[:, NUMERIC.index(GROUP_COL) if GROUP_COL
in NUMERIC else -1] # Get encoded group ID

# Transform val data

val_df['temp_target'] = val_df[TARGET]

X_val_transformed = preproc.transform(val_df[FEATURES +
['temp_target']])

y_val_flat = X_val_transformed[:, -1]

X_val_flat = X_val_transformed[:, :-1]

val_groups = X_val_flat[:, NUMERIC.index(GROUP_COL) if GROUP_COL in
NUMERIC else -1] # Get encoded group ID

NUM_FEATURES = X_train_flat.shape[1] # Number of features AFTER
preprocessing

print(f"Number of features after preprocessing: {NUM_FEATURES}")

# --- Sequence Creation Function ---

def create_sequences(data, targets, groups, sequence_length):

    sequences_X = []

    sequences_y = []

    unique_groups = np.unique(groups)

    print(f"Creating sequences for {len(unique_groups)} groups...")

    for group_id in unique_groups:

        group_mask = (groups == group_id)

        group_data = data[group_mask]

```

```

        group_targets = targets[group_mask]

        num_samples = len(group_data)

        if num_samples >= sequence_length:

            for i in range(num_samples - sequence_length + 1):

                # Sequence ends at i + sequence_length - 1

                # Target corresponds to the end of the sequence

                sequences_X.append(group_data[i : i +
sequence_length])

                sequences_y.append(group_targets[i + sequence_length
- 1])

            if not sequences_X:

                raise ValueError("No sequences created. Check sequence_length
and group sizes.")

            print(f"Total sequences created: {len(sequences_X)}")

            return np.array(sequences_X), np.array(sequences_y)

# --- Create Sequence Data ---

X_train_seq, y_train_seq = create_sequences(X_train_flat,
y_train_flat, train_groups, SEQUENCE_LENGTH)

X_val_seq, y_val_seq = create_sequences(X_val_flat, y_val_flat,
val_groups, SEQUENCE_LENGTH)

# --- Create TensorDatasets ---

train_ds = TensorDataset(torch.tensor(X_train_seq,
dtype=torch.float32),

                           torch.tensor(y_train_seq,
dtype=torch.float32))

val_ds = TensorDataset(torch.tensor(X_val_seq,
dtype=torch.float32),

                           torch.tensor(y_val_seq,
dtype=torch.float32))

```

```

print(f"Train sequences shape: X-{X_train_seq.shape}, y-
{y_train_seq.shape}")

print(f"Validation sequences shape: X-{X_val_seq.shape}, y-
{y_val_seq.shape}")

# -----

# — 2 Model definition -----

class HybridTimeSeriesRegressor(nn.Module):

    def __init__(self, num_features, sequence_length, d_model=64,
hidden=64, nhead=4, num_tf_layers=1, dropout=0.0):

        super().__init__()

        # Simplified "CNN": Linear projection applied per time step
feature vector

        self.feature_proj = nn.Linear(num_features, d_model)

        self.relu = nn.ReLU() # Activation after projection

        # LSTM processes the projected sequence

        self.lstm = nn.LSTM(d_model, hidden, batch_first=True,
dropout=dropout if num_tf_layers > 0 else 0) # Dropout only if layers
follow

        # Transformer Encoder processes the LSTM output sequence

        tf_encoder_layer = nn.TransformerEncoderLayer(

            d_model=hidden, nhead=nhead, dim_feedforward=hidden * 2,
# Common practice for dim_ff

            dropout=dropout, batch_first=True, activation='relu'

        )

        self.transformer_encoder =
nn.TransformerEncoder(tf_encoder_layer, num_layers=num_tf_layers)

        # Regression head uses the output of the *last* time step from
the transformer

        self.head = nn.Sequential(

            #nn.LayerNorm(hidden), # Optional normalization

```

```

        nn.Linear(hidden, 64),

        nn.ReLU(),

        nn.Dropout(dropout), # Dropout before final layer

        nn.Linear(64, 1)

    )

    def forward(self, x):

        # x shape: [Batch, SequenceLength, NumFeatures]

        batch_size, seq_len, _ = x.shape

        # Apply linear projection to features at each time step

        # Reshape for projection: [B * T, F] -> [B * T, d_model] ->
[B, T, d_model]

        x_proj = self.feature_proj(x.view(batch_size * seq_len, -1))

        x_proj = self.relu(x_proj)

        x_proj = x_proj.view(batch_size, seq_len, -1) # Shape: [B, T,
d_model]

        # Pass through LSTM

        lstm_out, _ = self.lstm(x_proj) # Shape: [B, T, Hidden]

        # Pass through Transformer Encoder

        tf_out = self.transformer_encoder(lstm_out) # Shape: [B, T,
Hidden]

        # Get output from the last time step

        last_time_step_out = tf_out[:, -1, :] # Shape: [B, Hidden]

        # Pass through regression head

        output = self.head(last_time_step_out) # Shape: [B, 1]

        return output

# -----
# — 3 Training helpers -----

```

```

def train_epoch(model, loader, crit, opt):

    model.train(); total_loss = 0

    for xb, yb in loader:

        xb, yb = xb.to(DEVICE), yb.to(DEVICE).unsqueeze(1) # Ensure
target is [B, 1]

        opt.zero_grad()

        preds = model(xb)

        loss = crit(preds, yb)

        loss.backward(); opt.step()

        total_loss += loss.item() * xb.size(0) # Accumulate loss
weighted by batch size

    return total_loss / len(loader.dataset) # Return average loss per
sample

def eval_epoch(model, loader, crit):

    model.eval(); total_loss = 0

    with torch.no_grad():

        for xb, yb in loader:

            xb, yb = xb.to(DEVICE), yb.to(DEVICE).unsqueeze(1)

            preds = model(xb)

            total_loss += crit(preds, yb).item() * xb.size(0)

    return total_loss / len(loader.dataset) # Return average loss per
sample

def full_metrics(model, loader):

    model.eval(); preds_list, trues_list = [], []

    with torch.no_grad():

        for xb, yb in loader:

            xb = xb.to(DEVICE)

```

```

        batch_preds = model(xb).cpu().numpy().flatten()

        preds_list.extend(batch_preds)

        trues_list.extend(yb.numpy()) # yb is already on CPU from
dataloader

    trues = np.array(trues_list)

    preds = np.array(preds_list)

    # Handle potential NaN/inf in predictions if model is unstable

    valid_indices = np.isfinite(preds)

    if not np.all(valid_indices):

        print(f"Warning: Found {np.sum(~valid_indices)} non-finite
predictions. Evaluating metrics on finite values only.")

        trues = trues[valid_indices]

        preds = preds[valid_indices]

        if len(trues) == 0:

            print("Error: No finite predictions left for metric
calculation.")

            return np.inf, np.inf, -np.inf # Return worst possible
scores

    rmse = mean_squared_error(trues, preds, squared=False)

    mae = mean_absolute_error(trues, preds)

    r2 = r2_score(trues, preds)

    return rmse, mae, r2

# -----

# — 4 Optuna objective -----

def objective(trial):

    params = {

        "d_model" : trial.suggest_categorical("d_model", [32,
64]), # Smaller projection

```

```

        "hidden"          : trial.suggest_categorical("hidden", [32,
64, 128]), # LSTM/TF hidden size

        "nhead"           : trial.suggest_categorical("nhead", [2, 4,
8]), # Must be divisor of hidden

        "num_tf_layers"   : trial.suggest_int("num_tf_layers", 1, 2), #
Transformer layers

        "dropout"         : trial.suggest_float("dropout", 0.05, 0.35),
# Slightly higher min dropout

        "lr"              : trial.suggest_float("lr", 1e-5, 1e-3,
log=True),

        "batch_size"      : trial.suggest_categorical("batch_size", [64,
128, 256]),

        "wd"              : trial.suggest_float("wd", 1e-6, 1e-3,
log=True), # Weight decay

    }

    # Ensure nhead is compatible with hidden size

    if params["hidden"] % params["nhead"] != 0:

        # Adjust nhead to the nearest smaller divisor or skip trial

        possible_nheads = [h for h in [1, 2, 4, 8] if params["hidden"]
% h == 0 and h <= params["hidden"]]

        if not possible_nheads:

            raise optuna.TrialPruned("No valid nhead found for this
hidden size.")

        params["nhead"] = random.choice(possible_nheads) # Or choose
max(possible_nheads)

        #print(f"Adjusted      nhead      to      {params['nhead']}      for
hidden={params['hidden']}")

        train_loader      =      DataLoader(train_ds,
batch_size=params["batch_size"], shuffle=True, drop_last=True)

        val_loader        =      DataLoader(val_ds,
batch_size=params["batch_size"], shuffle=False, drop_last=False)

        model = HybridTimeSeriesRegressor(

```

```

num_features=NUM_FEATURES,

sequence_length=SEQUENCE_LENGTH,

d_model=params["d_model"],

hidden=params["hidden"],

nhead=params["nhead"],

num_tf_layers=params["num_tf_layers"],

dropout=params["dropout"]

).to(DEVICE)

crit = nn.MSELoss()

opt = optim.Adam(model.parameters(), lr=params["lr"],
weight_decay=params["wd"])

sched = ReduceLROnPlateau(opt, mode='min', factor=0.5, patience=3,
verbose=False)

best_val_loss, patience_counter = math.inf, 0

for epoch in range(EPOCHS_MAX):

    train_loss = train_epoch(model, train_loader, crit, opt)

    val_loss = eval_epoch(model, val_loader, crit)

    sched.step(val_loss)

    trial.report(val_loss, epoch)

    if trial.should_prune():

        raise optuna.TrialPruned()

    if val_loss + 1e-5 < best_val_loss: # Added small tolerance

        best_val_loss = val_loss

        patience_counter = 0

    else:

        patience_counter += 1

```



```

        if patience_counter >= PATIENCE:

            #print(f"Early stopping triggered in Optuna trial at
epoch {epoch+1}")

            break

        return best_val_loss

print("\nStarting Optuna hyperparameter search...")

study = optuna.create_study(direction="minimize",

pruner=optuna.pruners.MedianPruner(n_startup_trials=5,
n_warmup_steps=10)) # Added warmup steps

study.optimize(objective, n_trials=30, show_progress_bar=True) #
Reduced trials for demo, use more (e.g., 50-100) in practice

best_params = study.best_params

print("Best hyper-parameters found:")

print(best_params)

# -----

# — 5 Train final model with best params -----

print("\nTraining final model with best parameters...")

batch_size = best_params.get("batch_size", BATCH_DEFAULT) # Use
Optuna's best batch size

train_loader = DataLoader(train_ds, batch_size=batch_size,
shuffle=True, drop_last=True)

val_loader = DataLoader(val_ds, batch_size=batch_size,
shuffle=False, drop_last=False)

# Ensure final model uses compatible nhead

if best_params["hidden"] % best_params["nhead"] != 0:

    possible_nheads = [h for h in [1, 2, 4, 8] if best_params["hidden"]
% h == 0 and h <= best_params["hidden"]]

    best_params["nhead"] = max(possible_nheads) if possible_nheads
else 1 # Default to 1 if necessary

```

```

    print(f"Adjusted final nhead to {best_params['nhead']} for
hidden={best_params['hidden']}")

final_model = HybridTimeSeriesRegressor(

    num_features=NUM_FEATURES,

    sequence_length=SEQUENCE_LENGTH,

    d_model=best_params["d_model"],

    hidden=best_params["hidden"],

    nhead=best_params["nhead"],

    num_tf_layers=best_params["num_tf_layers"],

    dropout=best_params["dropout"]

).to(DEVICE)

criterion = nn.MSELoss()

optimizer = optim.Adam(final_model.parameters(),
lr=best_params["lr"], weight_decay=best_params["wd"])

scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5,
patience=3, verbose=True)

history_train_loss, history_val_loss = [], []

best_val_loss, patience_counter = math.inf, 0

best_model_state = None

for epoch in range(1, EPOCHS_MAX + 1):

    train_loss = train_epoch(final_model, train_loader, criterion,
optimizer)

    val_loss = eval_epoch(final_model, val_loader, criterion)

    scheduler.step(val_loss)

    history_train_loss.append(train_loss)

    history_val_loss.append(val_loss)

```

```

    print(f"Epoch      {epoch:03d}/{EPOCHS_MAX}      |      Train      Loss:
{train_loss:.4f}      |      Val      Loss:      {val_loss:.4f}      |      LR:
{optimizer.param_groups[0]['lr']:.1e}")

    if val_loss + 1e-5 < best_val_loss:

        best_val_loss = val_loss

        patience_counter = 0

        best_model_state = final_model.state_dict()

        # Optional: Save checkpoint

        # torch.save(best_model_state, "best_model_checkpoint.pth")

    else:

        patience_counter += 1

        if patience_counter >= PATIENCE:

            print(f"\nEarly stopping triggered at epoch {epoch}.")

            break

# Load the best model state found during training

if best_model_state:

    print(f"\nLoading best model state from epoch {epoch -
patience_counter} with Val Loss: {best_val_loss:.4f}")

    final_model.load_state_dict(best_model_state)

else:

    print("\nWarning: No best model state saved (possibly due to
immediate divergence or error). Using last state.")

# -----

# — 6 Metrics & plotting -----

print("\nEvaluating final model...")

# Recalculate metrics using the *best loaded model*

# Ensure loaders reflect the data used for training (especially batch
size)

```

```

train_loader_eval = DataLoader(train_ds, batch_size=batch_size,
                               shuffle=False) # Use non-shuffled for consistency

val_loader_eval = DataLoader(val_ds, batch_size=batch_size,
                              shuffle=False)

rmse_tr, mae_tr, r2_tr = full_metrics(final_model, train_loader_eval)

rmse_va, mae_va, r2_va = full_metrics(final_model, val_loader_eval)

print("\n--- Final Model Evaluation ---")

print(f"Sequence Length Used : {SEQUENCE_LENGTH}")

print(f"Best Validation Loss (MSE) Achieved: {best_val_loss:.4f}")

print("\nFinal Scores (train / val)")

print(f"RMSE : {rmse_tr:.3f} / {rmse_va:.3f}")

print(f"MAE : {mae_tr:.3f} / {mae_va:.3f}")

print(f"R² : {r2_tr:.3f} / {r2_va:.3f}")

print("-----")

# --- 6-A Loss curves ---

plt.figure(figsize=(10, 5))

plt.plot(history_train_loss, label="Train Loss", color="tab:blue")

plt.plot(history_val_loss, label="Val Loss", color="tab:orange",
         linewidth=2)

plt.axhline(best_val_loss, color='tab:red', linestyle='--',
            label=f'Best Val Loss ({best_val_loss:.3f})')

plt.xlabel("Epoch")

plt.ylabel("Average MSE Loss")

plt.title(f"Hybrid Time-Series Model Training (Seq Len:
{SEQUENCE_LENGTH})")

plt.legend()

plt.grid(True, which='both', linestyle='--', linewidth=0.5)

plt.tight_layout()

```

```

plt.show()

# --- 6-B Bar chart of metrics ---

labels = ["RMSE", "MAE", "R\u00b2"]

train_vals = [rmse_tr, mae_tr, r2_tr]

val_vals    = [rmse_va, mae_va, r2_va]

x_pos = np.arange(len(labels))

bar_width = 0.35

fig, ax = plt.subplots(figsize=(8, 5))

rects1 = ax.bar(x_pos - bar_width/2, train_vals, bar_width,
label="Train", color="tab:green", alpha=0.8)

rects2 = ax.bar(x_pos + bar_width/2, val_vals, bar_width,
label="Val", color="tab:red", alpha=0.8)

ax.set_ylabel("Score")

ax.set_title("Final Regression Metrics")

ax.set_xticks(x_pos)

ax.set_xticklabels(labels)

ax.legend()

ax.grid(axis='y', linestyle='--', linewidth=0.5)

# Add value labels on bars

def autolabel(rects):

    for rect in rects:

        height = rect.get_height()

        ax.annotate(f'{height:.3f}',

                    xy=(rect.get_x() + rect.get_width() / 2, height),

                    xytext=(0, 3), # 3 points vertical offset

                    textcoords="offset points",

```

```

        ha='center', va='bottom')

autolabel(rects1)

autolabel(rects2)

plt.ylim(top=max(max(train_vals,          default=0),          max(val_vals,
default=0)) * 1.15) # Adjust y-limit

plt.tight_layout()

plt.show()

# --- 6-C Scatter plot: True vs Predicted (validation) ---

final_model.eval()

all_preds = []

all_trues = []

with torch.no_grad():

    for xb, yb in val_loader_eval: # Use the eval loader

        xb = xb.to(DEVICE)

        batch_preds = final_model(xb).cpu().numpy().flatten()

        all_preds.extend(batch_preds)

        all_trues.extend(yb.numpy()) # yb already on CPU

all_preds = np.array(all_preds)

all_trues = np.array(all_trues)

# Filter non-finite predictions if any occurred during final evaluation

valid_indices = np.isfinite(all_preds)

if not np.all(valid_indices):

    print(f"Warning: Filtering {np.sum(~valid_indices)} non-finite
predictions for scatter plot.")

    all_preds = all_preds[valid_indices]

    all_trues = all_trues[valid_indices]

```

```

plt.figure(figsize=(7, 7))

plt.scatter(all_trues, all_preds, s=10, alpha=0.5,
            color="tab:purple", label='Predictions')

# Add identity line

min_val = min(np.min(all_trues), np.min(all_preds)) * 0.95

max_val = max(np.max(all_trues), np.max(all_preds)) * 1.05

plt.plot([min_val, max_val], [min_val, max_val], "--k", linewidth=1,
         label='Ideal (y=x)')

plt.xlabel("True delay_ms")

plt.ylabel("Predicted delay_ms")

plt.title("True vs Predicted delay_ms (Validation Set)")

plt.legend()

plt.grid(True, linestyle='--', linewidth=0.5)

plt.axis('equal') # Ensure square aspect ratio

plt.xlim(min_val, max_val)

plt.ylim(min_val, max_val)

plt.tight_layout()

plt.show()

print("\n--- Experiment Summary ---")

print(f"Dataset: {CSV_PATH}")

print(f"Target Variable: {TARGET}")

print(f"Sequence Length: {SEQUENCE_LENGTH}")

print(f"Features Used ({NUM_FEATURES}): {' , '.join(NUMERIC)} + {' , '
'.join(CATEG)}") # Show features used

print(f"Best Hyperparameters via Optuna:")

for key, value in best_params.items():

    print(f"    - {key}: {value}")

```

```

print(f"Final Validation RMSE: {rmse_va:.4f}")

print(f"Final Validation MAE : {mae_va:.4f}")

print(f"Final Validation R2 : {r2_va:.4f}")

print("-----")

```

Appendix G.2: Source Code for the pure-LSTM Model for QoS Prediction

```

"""
Pure-LSTM Time-Series Regressor for predicting delay_ms

- Processes time-series sequences.

- leakage-safe preprocessing, device-wise split, Optuna H-param
search,

    early stopping, LR scheduler

Dataset : pf_wlan_qos_metrics.csv

-----

"""

# — 0 Imports & global settings


---



import random, math, os, warnings

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

plt.rcParams["font.size"] = 16          # global font size

from sklearn.compose      import ColumnTransformer

from sklearn.pipeline     import Pipeline

from sklearn.preprocessing import StandardScaler, OrdinalEncoder

from sklearn.impute       import SimpleImputer

```



```

from sklearn.model_selection import GroupShuffleSplit

from sklearn.metrics      import mean_squared_error,
mean_absolute_error, r2_score

import torch

from torch import nn, optim

from torch.utils.data import Dataset, DataLoader, TensorDataset #
Added Dataset for clarity, TensorDataset used

from torch.optim.lr_scheduler import ReduceLROnPlateau

import optuna

warnings.filterwarnings("ignore", category=UserWarning)

warnings.filterwarnings("ignore", category=FutureWarning) # Ignore
squared=False deprecation

# — Hyper-parameters & constants


---



SEED          = 42

BATCH_DEFAULT = 128

EPOCHS_MAX    = 80 # Max epochs for Optuna & final training

PATIENCE      = 8  # Early stopping patience

CSV_PATH      = "pf_wlan_qos_metrics.csv"

USE_PF_METRIC = False # set True to include pf_metric in the
feature set

SEQUENCE_LENGTH = 10 # <<< How many time steps to look back

random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

print(f"Using device: {DEVICE}")

print(f"Using Sequence Length: {SEQUENCE_LENGTH}")

#


---



```

```

# — 1 Load, Preprocess & Sequence Data


---



# (This section is identical to the corrected Hybrid Model)

df = pd.read_csv(CSV_PATH)

df = df.dropna(subset=["delay_ms"]).sort_values(by=['device_name',
'timestamp']).reset_index(drop=True) # Ensure order

TARGET = "delay_ms"

GROUP_COL = "device_name"

CATEG = [GROUP_COL]

NUMERIC = [c for c in df.columns

            if c not in ["timestamp", "device_ip", "round_id",
TARGET] + CATEG]

if not USE_PF_METRIC and "pf_metric" in NUMERIC:

    NUMERIC.remove("pf_metric")

FEATURES = NUMERIC + CATEG

NUM_FEATURES_PRE_TRANSFORM = len(FEATURES)

# --- Group-wise split (no device overlap) ---

print("Splitting data by device...")

gss = GroupShuffleSplit(n_splits=1, test_size=0.2,
random_state=SEED)

train_idx, val_idx = next(gss.split(df, groups=df[GROUP_COL]))

train_df, val_df = df.iloc[train_idx].copy(),
df.iloc[val_idx].copy()

# --- Preprocessing transformers ---

num_pipe = Pipeline([

    ("imp", SimpleImputer(strategy="median")),

    ("sc" , StandardScaler())

])

```

```

cat_pipe = Pipeline([

    ("enc", OrdinalEncoder(handle_unknown="use_encoded_value",
unknown_value=-1))

])

preproc = ColumnTransformer([

    ("num", num_pipe, NUMERIC),

    ("cat", cat_pipe, CATEG)

], remainder='passthrough')

# --- Fit on Train, Transform Train & Val ---

print("Preprocessing data...")

train_df['temp_target'] = train_df[TARGET]

X_train_transformed = preproc.fit_transform(train_df[FEATURES +
['temp_target']])

y_train_flat = X_train_transformed[:, -1]

X_train_flat = X_train_transformed[:, :-1]

# Find the index of the group column *after* transformation

# Assuming OrdinalEncoder puts the category feature at the end of
numeric features + original numeric features

group_col_index = len(NUMERIC) # Corrected index for OrdinalEncoder
output in this setup

train_groups = X_train_flat[:, group_col_index]

val_df['temp_target'] = val_df[TARGET]

X_val_transformed = preproc.transform(val_df[FEATURES +
['temp_target']])

y_val_flat = X_val_transformed[:, -1]

X_val_flat = X_val_transformed[:, :-1]

val_groups = X_val_flat[:, group_col_index]

```

```

NUM_FEATURES = X_train_flat.shape[1] # Number of features AFTER
preprocessing

print(f"Number of features after preprocessing: {NUM_FEATURES}")

# --- Sequence Creation Function ---

def create_sequences(data, targets, groups, sequence_length):

    sequences_X = []

    sequences_y = []

    unique_groups = np.unique(groups)

    print(f"Creating sequences for {len(unique_groups)} groups...")

    for group_id in unique_groups:

        group_mask = (groups == group_id)

        group_data = data[group_mask]

        group_targets = targets[group_mask]

        num_samples = len(group_data)

        if num_samples >= sequence_length:

            for i in range(num_samples - sequence_length + 1):

                sequences_X.append(group_data[i : i +
sequence_length])

                sequences_y.append(group_targets[i + sequence_length
- 1]) # Target is from the last step

            if not sequences_X:

                raise ValueError("No sequences created. Check
sequence_length and group sizes.")

            print(f"Total sequences created: {len(sequences_X)}")

            return np.array(sequences_X), np.array(sequences_y)

# --- Create Sequence Data ---

X_train_seq, y_train_seq = create_sequences(X_train_flat,
y_train_flat, train_groups, SEQUENCE_LENGTH)

```

```

X_val_seq, y_val_seq = create_sequences(X_val_flat,
y_val_flat, val_groups, SEQUENCE_LENGTH)

# --- Create TensorDatasets ---

train_ds = TensorDataset(torch.tensor(X_train_seq,
dtype=torch.float32),

                        torch.tensor(y_train_seq,
dtype=torch.float32))

val_ds = TensorDataset(torch.tensor(X_val_seq,
dtype=torch.float32),

                        torch.tensor(y_val_seq,
dtype=torch.float32))

print(f"Train sequences shape: X-{X_train_seq.shape}, y-
{y_train_seq.shape}")

print(f"Validation sequences shape: X-{X_val_seq.shape}, y-
{y_val_seq.shape}")

#

```

```

# — 2 LSTM model definition

```

```

class LSTMTimeSeriesRegressor(nn.Module):

    def __init__(self, input_size, hidden_size=64, num_layers=1,
dropout=0.0):

        super().__init__()

        # LSTM layer processes sequences along time dimension

        # input_size is the number of features per time step

        self.lstm = nn.LSTM(input_size=input_size,

                            hidden_size=hidden_size,

                            num_layers=num_layers,

                            batch_first=True, # Expect input shape
[Batch, SeqLen, Features]

```

```

                                dropout=dropout if num_layers > 1 else
0) # Dropout between LSTM layers only

    # Regression head uses the output of the *last* time step

    self.head = nn.Sequential(

        #nn.LayerNorm(hidden_size), # Optional normalization

        nn.Linear(hidden_size, 64), # Input is LSTM hidden size

        nn.ReLU(),

        nn.Dropout(dropout), # Apply dropout before final layer

        nn.Linear(64, 1)          # Output is single value

    )

    def forward(self, x):

        # x shape: [Batch, SequenceLength, NumFeatures]

        # Pass sequence through LSTM

        # out shape: [Batch, SequenceLength, HiddenSize]

        # hn shape: [NumLayers, Batch, HiddenSize] (final hidden
state)

        # cn shape: [NumLayers, Batch, HiddenSize] (final cell
state)

        out, (hn, cn) = self.lstm(x)

        # Use the output from the last time step for prediction

        # last_time_step_out shape: [Batch, HiddenSize]

        last_time_step_out = out[:, -1, :]

        # Pass through regression head

        prediction = self.head(last_time_step_out) # Shape: [Batch,
1]

        return prediction

#

```

— 3 Training helpers

```
# (Identical to the Hybrid Model)

def train_epoch(model, loader, crit, opt):

    model.train(); total_loss = 0

    for xb, yb in loader:

        xb, yb = xb.to(DEVICE), yb.to(DEVICE).unsqueeze(1) # Ensure
        target is [B, 1]

        opt.zero_grad()

        preds = model(xb)

        loss = crit(preds, yb)

        loss.backward(); opt.step()

        total_loss += loss.item() * xb.size(0)

    return total_loss / len(loader.dataset)

def eval_epoch(model, loader, crit):

    model.eval(); total_loss = 0

    with torch.no_grad():

        for xb, yb in loader:

            xb, yb = xb.to(DEVICE), yb.to(DEVICE).unsqueeze(1)

            preds = model(xb)

            total_loss += crit(preds, yb).item() * xb.size(0)

    return total_loss / len(loader.dataset)

def full_metrics(model, loader):

    model.eval(); preds_list, trues_list = [], []

    with torch.no_grad():

        for xb, yb in loader:
```

```

        xb = xb.to(DEVICE)

        batch_preds = model(xb).cpu().numpy().flatten()

        preds_list.extend(batch_preds)

        trues_list.extend(yb.numpy())

    trues = np.array(trues_list)

    preds = np.array(preds_list)

    valid_indices = np.isfinite(preds)

    if not np.all(valid_indices):

        print(f"Warning: Found {np.sum(~valid_indices)} non-finite
predictions. Evaluating metrics on finite values only.")

        trues = trues[valid_indices]

        preds = preds[valid_indices]

        if len(trues) == 0:

            print("Error: No finite predictions left for metric
calculation.")

            return np.inf, np.inf, -np.inf

    rmse = mean_squared_error(trues, preds, squared=False)

    mae = mean_absolute_error(trues, preds)

    r2 = r2_score(trues, preds)

    return rmse, mae, r2

#


---


# — 4 Optuna objective


---



def objective(trial):

    params = {

        "hidden_size" : trial.suggest_categorical("hidden_size",
[32, 64, 128, 256]), # LSTM hidden size

```



```

        "num_layers"      : trial.suggest_int("num_layers", 1, 3), #
LSTM layers

        "dropout"         : trial.suggest_float("dropout", 0.05, 0.4),
# Dropout (applied between LSTM layers & in head)

        "lr"              : trial.suggest_float("lr", 1e-5, 1e-3,
log=True),

        "batch_size"      : trial.suggest_categorical("batch_size",
[64, 128, 256]),

        "wd"              : trial.suggest_float("wd", 1e-6, 1e-3,
log=True), # Weight decay

    }

    train_loader = DataLoader(train_ds,
batch_size=params["batch_size"], shuffle=True, drop_last=True)

    val_loader    = DataLoader(val_ds,
batch_size=params["batch_size"], shuffle=False, drop_last=False)

    model = LSTMTimeSeriesRegressor(

        input_size=NUM_FEATURES, # Fixed based on data

        hidden_size=params["hidden_size"],

        num_layers=params["num_layers"],

        dropout=params["dropout"]

    ).to(DEVICE)

    crit = nn.MSELoss()

    opt  = optim.Adam(model.parameters(), lr=params["lr"],
weight_decay=params["wd"])

    sched = ReduceLROnPlateau(opt, mode='min', factor=0.5,
patience=3, verbose=False)

    best_val_loss, patience_counter = math.inf, 0

    for epoch in range(EPOCHS_MAX):

        train_loss = train_epoch(model, train_loader, crit, opt)

        val_loss = eval_epoch(model, val_loader, crit)

```

```

        sched.step(val_loss)

        trial.report(val_loss, epoch)

        if trial.should_prune():

            raise optuna.TrialPruned()

        if val_loss + 1e-5 < best_val_loss:

            best_val_loss = val_loss

            patience_counter = 0

        else:

            patience_counter += 1

            if patience_counter >= PATIENCE:

                break

    return best_val_loss

print("\nStarting Optuna hyperparameter search for LSTM model...")

study = optuna.create_study(direction="minimize",

pruner=optuna.pruners.MedianPruner(n_startup_trials=5,
n_warmup_steps=10))

study.optimize(objective, n_trials=30, show_progress_bar=True) # Use
more trials in practice

best_params = study.best_params

print("Best hyper-parameters found:")

print(best_params)

#


---



# — 5 Train final model with best params


---



print("\nTraining final LSTM model with best parameters...")

batch_size = best_params.get("batch_size", BATCH_DEFAULT)

```

```

train_loader = DataLoader(train_ds, batch_size=batch_size,
                           shuffle=True, drop_last=True)

val_loader = DataLoader(val_ds, batch_size=batch_size,
                        shuffle=False, drop_last=False)

final_model = LSTMTimeSeriesRegressor(

    input_size=NUM_FEATURES,

    hidden_size=best_params["hidden_size"],

    num_layers=best_params["num_layers"],

    dropout=best_params["dropout"]

).to(DEVICE)

criterion = nn.MSELoss()

optimizer = optim.Adam(final_model.parameters(),
                        lr=best_params["lr"], weight_decay=best_params["wd"])

scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5,
                               patience=3, verbose=True)

history_train_loss, history_val_loss = [], []

best_val_loss, patience_counter = math.inf, 0

best_model_state = None

for epoch in range(1, EPOCHS_MAX + 1):

    train_loss = train_epoch(final_model, train_loader, criterion,
                             optimizer)

    val_loss = eval_epoch(final_model, val_loader, criterion)

    scheduler.step(val_loss)

    history_train_loss.append(train_loss)

    history_val_loss.append(val_loss)

    print(f"Epoch {epoch:03d}/{EPOCHS_MAX} | Train Loss:
{train_loss:.4f} | Val Loss: {val_loss:.4f} | LR:
{optimizer.param_groups[0]['lr']:.1e}")

    if val_loss + 1e-5 < best_val_loss:

```

```

        best_val_loss = val_loss

        patience_counter = 0

        best_model_state = final_model.state_dict()

    else:

        patience_counter += 1

        if patience_counter >= PATIENCE:

            print(f"\nEarly stopping triggered at epoch {epoch}.")

            break

# Load the best model state

if best_model_state:

    print(f"\nLoading best model state from epoch {epoch -
patience_counter} with Val Loss: {best_val_loss:.4f}")

    final_model.load_state_dict(best_model_state)

else:

    print("\nWarning: No best model state saved. Using last state.")

#


---



# — 6 Metrics & plotting


---



print("\nEvaluating final LSTM model...")

train_loader_eval = DataLoader(train_ds, batch_size=batch_size,
shuffle=False)

val_loader_eval = DataLoader(val_ds, batch_size=batch_size,
shuffle=False)

rmse_tr, mae_tr, r2_tr = full_metrics(final_model,
train_loader_eval)

rmse_va, mae_va, r2_va = full_metrics(final_model, val_loader_eval)

print("\n--- Final LSTM Model Evaluation ---")

```

```

print(f"Sequence Length Used : {SEQUENCE_LENGTH}")

print(f"Best Validation Loss (MSE) Achieved: {best_val_loss:.4f}")

print("\nFinal Scores (train / val)")

print(f"RMSE : {rmse_tr:.3f} / {rmse_va:.3f}")

print(f"MAE : {mae_tr :.3f} / {mae_va :.3f}")

print(f"R² : {r2_tr :.3f} / {r2_va :.3f}")

print("-----")

# --- 6-A Loss curves ---

plt.figure(figsize=(10, 5))

plt.plot(history_train_loss, label="Train Loss", color="tab:blue")

plt.plot(history_val_loss, label="Val Loss", color="tab:orange",
linewidth=2)

plt.axhline(best_val_loss, color='tab:red', linestyle='--',
label=f'Best Val Loss ({best_val_loss:.3f})')

plt.xlabel("Epoch")

plt.ylabel("Average MSE Loss")

plt.title(f"LSTM Time-Series Model Training (Seq Len:
{SEQUENCE_LENGTH})")

plt.legend()

plt.grid(True, which='both', linestyle='--', linewidth=0.5)

plt.tight_layout()

plt.show()

# --- 6-B Bar chart of metrics ---

labels = ["RMSE", "MAE", "R\u00b2"]

train_vals = [rmse_tr, mae_tr, r2_tr]

val_vals = [rmse_va, mae_va, r2_va]

x_pos = np.arange(len(labels))

```

```

bar_width = 0.35

fig, ax = plt.subplots(figsize=(8, 5))

rects1 = ax.bar(x_pos - bar_width/2, train_vals, bar_width,
label="Train", color="tab:green", alpha=0.8)

rects2 = ax.bar(x_pos + bar_width/2, val_vals, bar_width,
label="Val", color="tab:red", alpha=0.8)

ax.set_ylabel("Score")

ax.set_title("Final Regression Metrics (LSTM)")

ax.set_xticks(x_pos)

ax.set_xticklabels(labels)

ax.legend()

ax.grid(axis='y', linestyle='--', linewidth=0.5)

# Add value labels
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f'{height:.3f}', xy=(rect.get_x() +
rect.get_width() / 2, height),
                    xytext=(0, 3), textcoords="offset points",
ha='center', va='bottom')

autolabel(rects1)

autolabel(rects2)

plt.ylim(top=max(max(train_vals, default=0), max(val_vals,
default=0)) * 1.15)

plt.tight_layout()

plt.show()

# --- 6-C Scatter plot: True vs Predicted (validation) ---

final_model.eval()

```

```

all_preds = []

all_trues = []

with torch.no_grad():

    for xb, yb in val_loader_eval:

        xb = xb.to(DEVICE)

        batch_preds = final_model(xb).cpu().numpy().flatten()

        all_preds.extend(batch_preds)

        all_trues.extend(yb.numpy())

all_preds = np.array(all_preds)

all_trues = np.array(all_trues)

valid_indices = np.isfinite(all_preds)

if not np.all(valid_indices):

    print(f"Warning: Filtering {np.sum(~valid_indices)} non-finite
    predictions for scatter plot.")

    all_preds = all_preds[valid_indices]

    all_trues = all_trues[valid_indices]

plt.figure(figsize=(7, 7))

plt.scatter(all_trues, all_preds, s=10, alpha=0.5,
color="tab:purple", label='Predictions')

min_val = min(np.min(all_trues), np.min(all_preds)) * 0.95

max_val = max(np.max(all_trues), np.max(all_preds)) * 1.05

plt.plot([min_val, max_val], [min_val, max_val], "--k", linewidth=1,
label='Ideal (y=x)')

plt.xlabel("True delay_ms")

plt.ylabel("Predicted delay_ms")

plt.title("True vs Predicted delay_ms (LSTM - Validation Set)")

plt.legend()

```

```

plt.grid(True, linestyle='--', linewidth=0.5)

plt.axis('equal')

plt.xlim(min_val, max_val)

plt.ylim(min_val, max_val)

plt.tight_layout()

plt.show()

print("\n--- LSTM Experiment Summary ---")

print(f"Dataset: {CSV_PATH}")

print(f"Target Variable: {TARGET}")

print(f"Sequence Length: {SEQUENCE_LENGTH}")

print(f"Features Used ({NUM_FEATURES}): {' , '.join(NUMERIC)} + {' , '
'.join(CATEG)}")

print(f"Best Hyperparameters via Optuna:")

for key, value in best_params.items():

    print(f"    - {key}: {value}")

print(f"Final Validation RMSE: {rmse_va:.4f}")

print(f"Final Validation MAE : {mae_va:.4f}")

print(f"Final Validation R2 : {r2_va:.4f}")

print("-----")

```

Appendix G.3: Source Code for the RF Baseline Model for QoS Prediction

```

"""

Baseline Random Forest Regressor for predicting delay_ms

- Operates on flat tabular data (no explicit time sequence).

```



```

- leakage-safe preprocessing, device-wise split, Optuna H-param
search

Dataset : pf_wlan_qos_metrics.csv

-----

"""

# — 0 Imports & global settings


---



import random, math, os, warnings

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

plt.rcParams["font.size"] = 16          # global font size

from sklearn.compose      import ColumnTransformer

from sklearn.pipeline     import Pipeline

from sklearn.preprocessing import StandardScaler, OrdinalEncoder

from sklearn.impute       import SimpleImputer

from sklearn.model_selection import GroupShuffleSplit

from sklearn.ensemble     import RandomForestRegressor

from sklearn.metrics      import mean_squared_error,
mean_absolute_error, r2_score

import optuna

warnings.filterwarnings("ignore", category=UserWarning)

warnings.filterwarnings("ignore", category=FutureWarning)

# — Hyper-parameters & constants


---



SEED          = 42

EPOCHS_MAX    = 1    # Not applicable for RF, just placeholder if
needed elsewhere

```

```

PATIENCE          = 1    # Not applicable for RF

CSV_PATH          = "pf_wlan_qos_metrics.csv"

USE_PF_METRIC     = False    # set True to include pf_metric in the
                             feature set

random.seed(SEED); np.random.seed(SEED) # Set seed for
reproducibility where possible

DEVICE = "CPU" # RF runs on CPU

print(f"Using device: {DEVICE}")

#


---



# --- 1 Load, Preprocess Data (Flat Tabular) -----

# (Preprocessing is identical to previous models up to the flat
stage)

df = pd.read_csv(CSV_PATH)

df = df.dropna(subset=["delay_ms"]).sort_values(by=['device_name',
'timestamp']).reset_index(drop=True)

TARGET = "delay_ms"

GROUP_COL = "device_name"

CATEG   = [GROUP_COL]

NUMERIC = [c for c in df.columns

            if c not in ["timestamp", "device_ip", "round_id",
TARGET] + CATEG]

if not USE_PF_METRIC and "pf_metric" in NUMERIC:

    NUMERIC.remove("pf_metric")

FEATURES = NUMERIC + CATEG

# --- Group-wise split ---

print("Splitting data by device...")

gss = GroupShuffleSplit(n_splits=1, test_size=0.2,
random_state=SEED)

```

```

train_idx, val_idx = next(gss.split(df, groups=df[GROUP_COL]))

train_df, val_df = df.iloc[train_idx].copy(),
df.iloc[val_idx].copy()

# --- Preprocessing transformers ---

num_pipe = Pipeline([

    ("imp", SimpleImputer(strategy="median")),

    ("sc", StandardScaler()) # Scaling is less critical for RF but
doesn't hurt

])

cat_pipe = Pipeline([

    ("enc", OrdinalEncoder(handle_unknown="use_encoded_value",
unknown_value=-1))

])

# Note: No 'remainder=passthrough' needed if we only select FEATURES

preproc = ColumnTransformer([

    ("num", num_pipe, NUMERIC),

    ("cat", cat_pipe, CATEG)

])

# --- Fit on Train, Transform Train & Val ---

print("Preprocessing data...")

X_train_flat = preproc.fit_transform(train_df[FEATURES])

y_train_flat = train_df[TARGET].values

X_val_flat = preproc.transform(val_df[FEATURES])

y_val_flat = val_df[TARGET].values

NUM_FEATURES = X_train_flat.shape[1]

print(f"Number of features after preprocessing: {NUM_FEATURES}")

# Attempt to get feature names after transformation

```

```

try:

    FEATURE_NAMES_OUT = preproc.get_feature_names_out()

except Exception:

    print("Warning: Could not get feature names from
    ColumnTransformer. Using generic names.")

    FEATURE_NAMES_OUT = [f"feature_{i}" for i in
    range(NUM_FEATURES)]

print(f"Train data shape: X-{X_train_flat.shape}, y-
{y_train_flat.shape}")

print(f"Validation data shape: X-{X_val_flat.shape}, y-
{y_val_flat.shape}")

#

```

```

# — 2 Optuna objective for Random Forest

```

```

def objective_rf(trial):

    # Define hyperparameters to search for RandomForestRegressor

    rf_params = {

        'n_estimators': trial.suggest_int('n_estimators', 50, 400),

        'max_depth': trial.suggest_int('max_depth', 5, 50,
log=True), # Log scale for depth

        'min_samples_split': trial.suggest_int('min_samples_split',
2, 20),

        'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1,
20),

        'max_features': trial.suggest_categorical('max_features',
['sqrt', 'log2', 0.3, 0.5]), # Common options

        'random_state': SEED, # for reproducibility within trial

        'n_jobs': -1          # Use all available CPU cores

    }

```

```

model = RandomForestRegressor(**rf_params)

# Train the model

model.fit(X_train_flat, y_train_flat)

# Evaluate on the validation set

y_pred_val = model.predict(X_val_flat)

# Calculate validation MSE (Optuna minimizes this)

val_mse = mean_squared_error(y_val_flat, y_pred_val)

# Pruning is less common/effective for RF trials but can be kept

trial.report(val_mse, step=0)

if trial.should_prune():

    raise optuna.TrialPruned()

return val_mse

print("\nStarting Optuna hyperparameter search for Random
Forest...")

# Create study - direction is minimize MSE

study_rf = optuna.create_study(direction="minimize",

pruner=optuna.pruners.MedianPruner(n_startup_trials=5)) # Keep
pruner just in case

# Optimize study

study_rf.optimize(objective_rf, n_trials=30, show_progress_bar=True)
# Adjust n_trials as needed

best_params_rf = study_rf.best_params

print("Best hyper-parameters found for Random Forest:")

print(best_params_rf)

#

```

```

# — 3 Train final Random Forest model


---



print("\nTraining final Random Forest model with best
parameters...")

final_rf_model = RandomForestRegressor(

    **best_params_rf, # Unpack best parameters found by Optuna

    random_state=SEED,

    n_jobs=-1

)

# Train on the full training data

final_rf_model.fit(X_train_flat, y_train_flat)

print("Final Random Forest model trained.")

#


---



# — 4 Evaluate Final Model


---



print("\nEvaluating final Random Forest model...")

# Predict on training and validation sets

y_pred_train_rf = final_rf_model.predict(X_train_flat)

y_pred_val_rf = final_rf_model.predict(X_val_flat)

# Calculate metrics

rmse_tr = mean_squared_error(y_train_flat, y_pred_train_rf,
squared=False)

mae_tr = mean_absolute_error(y_train_flat, y_pred_train_rf)

r2_tr = r2_score(y_train_flat, y_pred_train_rf)

rmse_va = mean_squared_error(y_val_flat, y_pred_val_rf,
squared=False)

mae_va = mean_absolute_error(y_val_flat, y_pred_val_rf)

```

```

r2_va = r2_score(y_val_flat, y_pred_val_rf)

print("\n--- Final Random Forest Model Evaluation ---")

print(f"Best Validation MSE during Optuna:
{study_rf.best_value:.4f}") # Optuna minimized MSE

print("\nFinal Scores (train / val)")

print(f"RMSE : {rmse_tr:.3f} / {rmse_va:.3f}")

print(f"MAE : {mae_tr:.3f} / {mae_va:.3f}")

print(f"R² : {r2_tr:.3f} / {r2_va:.3f}")

print("-----")

#

```

```

# — 5 Plotting

```

```

# --- 5-A Feature Importances ---

importances = final_rf_model.feature_importances_

indices = np.argsort(importances)[::-1] # Sort features by
importance

plt.figure(figsize=(12, max(6, NUM_FEATURES * 0.4))) # Adjust height
based on num features

plt.title("Random Forest Feature Importances")

plt.barh(range(NUM_FEATURES), importances[indices],
color="tab:blue", align="center")

plt.yticks(range(NUM_FEATURES), [FEATURE_NAMES_OUT[i] for i in
indices])

plt.xlabel("Importance Score")

plt.gca().invert_yaxis() # Display most important features at the
top

plt.tight_layout()

plt.show()

```

```

# --- 5-B Bar chart of metrics ---

labels = ["RMSE", "MAE", "R\u00b2"]

train_vals = [rmse_tr, mae_tr, r2_tr]

val_vals    = [rmse_va, mae_va, r2_va]

x_pos = np.arange(len(labels))

bar_width = 0.35

fig, ax = plt.subplots(figsize=(8, 5))

rects1 = ax.bar(x_pos - bar_width/2, train_vals, bar_width,
label="Train", color="tab:green", alpha=0.8)

rects2 = ax.bar(x_pos + bar_width/2, val_vals, bar_width,
label="Val", color="tab:red", alpha=0.8)

ax.set_ylabel("Score")

ax.set_title("Final Regression Metrics (Random Forest)")

ax.set_xticks(x_pos)

ax.set_xticklabels(labels)

ax.legend()

ax.grid(axis='y', linestyle='--', linewidth=0.5)

# Add value labels

def autolabel(rects):

    for rect in rects:

        height = rect.get_height()

        ax.annotate(f'{height:.3f}', xy=(rect.get_x() +
rect.get_width() / 2, height),

                    xytext=(0, 3), textcoords="offset points",
ha='center', va='bottom')

autolabel(rects1)

autolabel(rects2)

```



```

plt.ylim(top=max(max(train_vals, default=0), max(val_vals,
default=0)) * 1.15)

plt.tight_layout()

plt.show()

# --- 5-C Scatter plot: True vs Predicted (validation) ---

plt.figure(figsize=(7, 7))

plt.scatter(y_val_flat, y_pred_val_rf, s=10, alpha=0.5,
color="tab:purple", label='Predictions')

min_val = min(np.min(y_val_flat), np.min(y_pred_val_rf)) * 0.95

max_val = max(np.max(y_val_flat), np.max(y_pred_val_rf)) * 1.05

plt.plot([min_val, max_val], [min_val, max_val], "--k", linewidth=1,
label='Ideal (y=x)')

plt.xlabel("True delay_ms")

plt.ylabel("Predicted delay_ms")

plt.title("True vs Predicted delay_ms (Random Forest - Validation
Set)")

plt.legend()

plt.grid(True, linestyle='--', linewidth=0.5)

plt.axis('equal')

plt.xlim(min_val, max_val)

plt.ylim(min_val, max_val)

plt.tight_layout()

plt.show()

#
_____

# — 6 Experiment Summary —————

print("\n--- Random Forest Baseline Experiment Summary ---")

print(f"Dataset: {CSV_PATH}")

```

```

print(f"Target Variable: {TARGET}")

print(f"Features Used ({NUM_FEATURES}): {'',
'.join(FEATURE_NAMES_OUT)}")

print(f"Model Type: RandomForestRegressor")

print(f"Best Hyperparameters via Optuna:")

for key, value in best_params_rf.items():

    print(f"    - {key}: {value}")

print(f"Final Validation RMSE: {rmse_va:.4f}")

print(f"Final Validation MAE : {mae_va:.4f}")

print(f"Final Validation R2 : {r2_va:.4f}")

print("-----")

```

Appendix H: Source Code for Experiment 5 (Transformer-Based RL Scheduler Agent)

This appendix provides the source code for the Transformer-based DRL scheduling agent evaluated in Experiment 5. The implementation details of the model, including state representation, policy and value networks, and training procedures, are described in *Section 4.9*. The performance evaluation is presented in *Section 5.6*, and the broader implications of this RL-based approach to traffic scheduling in 5G networks are discussed in *Section 6.6*.

Appendix H.1: Transformer-Based RL Scheduler Version 2.0

```

#!/usr/bin/env python3

import numpy as np

import icmplib

import matplotlib.pyplot as plt

import matplotlib.cm as cm

import torch

import torch.nn as nn

```

```

import torch.optim as optim

from torch.distributions import Categorical

import time

import optuna

import logging

import statistics

from datetime import datetime

import os

from typing import List, Tuple, Dict, Optional

# -----

# Configuration and Logging Setup

# -----

# Configure logging

logging.basicConfig(

    level=logging.INFO,

    format='[% (levelname)s] %(asctime)s - %(message)s',

    datefmt='%Y-%m-%d %H:%M:%S',

    handlers=[

        logging.StreamHandler(),

        logging.FileHandler(f"rl_transformer_scheduler_log_{datetime.now().s
trftime('%Y%m%d_%H%M%S')}.log")

    ]

)

logger = logging.getLogger(__name__)

# Experiment parameters - Adopted from PF example where applicable

# SITES = [

#     "192.168.1.140", "192.168.1.141", "192.168.1.142",
#     "192.168.1.143"

```

```

# ] # Use local devices for closer comparison to PF example if
needed

SITES = [

    "192.168.1.140", "192.168.1.141", "192.168.1.142",
    "192.168.1.143",

    "1.1.1.1", "8.8.8.8", "8.8.4.4", "1.1.1.2", "9.9.9.9",
    "208.67.222.222",

    "208.67.220.220", "8.26.56.26", "8.20.247.20", "1.0.0.1",

]

SITE_NAMES = {ip: f"site_{i+1}" for i, ip in enumerate(SITES)} #
Assign names like device_1, device_2, etc.

NUM_SITES = len(SITES)

NUM_EPOCHS = 1000

HIDDEN_DIM = 128

STEPS_PER_EPOCH = 100 # Number of steps (scheduling decisions) per
epoch

PING_TIMEOUT = 1.0 # seconds to wait for ping response (adjusted
from PF example's 2.0 for potentially faster RL steps)

PING_SIZE = 64 # payload size in bytes

ALPHA = 0.2 # Weight for exponential moving average for
throughput

EPSILON = 1e-3 # Small epsilon to avoid division by zero

# -----

# Device Class Definition (Adapted from PF Example)

# -----

class RLDevice:

    """

    Represents a target site and tracks its QoS metrics for the RL
    scheduler.

    """

    def __init__(self, ip: str, name: str):

```

```

        self.ip = ip

        self.name = name

        self.packets_sent = 0

        self.packets_received = 0

        self.packets_lost = 0

        # Timestamps

        self.first_ping_time: Optional[float] = None

        self.last_ping_time: Optional[float] = None

        # Performance metrics

        self.delays: List[float] = [] # List of recent delay values
in ms

        self.last_delay: Optional[float] = None

        self.avg_delay: Optional[float] = None

        self.min_delay: Optional[float] = None

        self.max_delay: Optional[float] = None

        # Throughput metrics (in bits per second)

        self.instantaneous_throughput_bps = 0.0

        self.average_throughput_bps = 0.0

        # History for potential detailed analysis (optional)

        self.history = {

            "timestamps": [],

            "delays": [],

            "throughputs_bps": [],

        }

        def update_metrics(self, success: bool, delay: Optional[float],
current_time: float) -> None:

            """

            Updates the device's metrics based on the latest ping
result.

```

Called only when this device *was selected* by the scheduler.

Args:

success: Whether the ping was successful.

delay: Round-trip time in milliseconds (if successful).

current_time: Current timestamp.

"""

self.packets_sent += 1

self.history["timestamps"].append(current_time)

if self.first_ping_time is None:

self.first_ping_time = current_time

if success and delay is not None:

self.packets_received += 1

self.last_delay = delay

Update delay list (keep last N, e.g., 20)

self.delays.append(delay)

if len(self.delays) > 20:

self.delays.pop(0)

Update aggregate delay stats

if self.delays:

self.avg_delay = statistics.mean(self.delays)

self.min_delay = min(self.delays)

self.max_delay = max(self.delays)

Calculate instantaneous throughput in bps:

(PING_SIZE bytes * 8 bits/byte * 1000 ms/s) / delay

(ms)

instantaneous_rate = (PING_SIZE * 8 * 1000) / delay if
delay > 0 else 0.0

self.history["delays"].append(delay)

```

else:

    self.packets_lost += 1

    instantaneous_rate = 0.0

    self.history["delays"].append(None) # Record failure

# Update average throughput using exponential moving average
if self.average_throughput_bps == 0:

    # Initialize average throughput carefully

    if self.packets_received == 1: # First successful ping

        self.average_throughput_bps = instantaneous_rate

        # else: remain 0 until first success

    else:

        self.average_throughput_bps = (ALPHA *
instantaneous_rate +

                                     (1 - ALPHA) *
self.average_throughput_bps)

        self.instantaneous_throughput_bps = instantaneous_rate

self.history["throughputs_bps"].append(self.instantaneous_throughput
_bps)

    self.last_ping_time = current_time

# -----

# Transformer Model

# -----

class TransformerScheduler(nn.Module):

    def __init__(self, input_dim: int, hidden_dim: int, output_dim:
int):

        super(TransformerScheduler, self).__init__()

        self.embedding = nn.Embedding(input_dim, hidden_dim)

        encoder_layer = nn.TransformerEncoderLayer(

```

```

        d_model=hidden_dim,

        nhead=4,

        batch_first=True

    )

    self.transformer_encoder =
nn.TransformerEncoder(encoder_layer, num_layers=2)

    self.decoder = nn.Linear(hidden_dim, output_dim)

    self.softmax = nn.Softmax(dim=-1)

    def forward(self, src: torch.Tensor) -> torch.Tensor:

        src = self.embedding(src)

        out = self.transformer_encoder(src)

        out = self.decoder(out[:, -1, :])

        return self.softmax(out)

# -----
# Helper Functions (Modified Ping)
# -----

def ping_site(site_ip: str) -> Tuple[bool, Optional[float]]:

    """

    Ping a site and return success status and RTT.

    Args:

        site_ip (str): IP address or hostname to ping.

    Returns:

        Tuple[bool, Optional[float]]: (is_alive, average_rtt_ms or
None).

    """

    try:

        # Use bytes payload matching PF example

        payload = b"X" * PING_SIZE

```



```

        result = icmpplib.ping(

            site_ip,

            count=1,

            interval=0.2, # Keep low interval as RL steps might be
fast
            timeout=PING_TIMEOUT,

            payload=payload,

            privileged=False # Avoid needing root

        )

        if result.is_alive:

            return True, result.avg_rtt

        else:

            # logger.debug(f"Ping to {site_ip} timed out or
failed.")

            return False, None

    except icmpplib.exceptions.ICMPError as e:

        logger.warning(f"ICMPError while pinging {site_ip}: {e}")

        return False, None

    except Exception as e: # Catch other potential errors

        logger.error(f"Unexpected error pinging {site_ip}: {e}")

        return False, None

def choose_action(state: int, model: TransformerScheduler) ->
Tuple[int, torch.Tensor]:

    """

    Choose an action based on the current state and model's policy.
(Unchanged)

    Args:

        state (int): Current state represented as an integer (index
of last chosen site).

        model (TransformerScheduler): The trained Transformer model.

```

```

Returns:

    Tuple[int, torch.Tensor]: Chosen action (site index) and its
    log probability.

    """

    state_tensor = torch.tensor([state],
dtype=torch.long).unsqueeze(0)  # Shape: (1, 1)

    action_probs = model(state_tensor)  # Shape: (1, NUM_SITES)

    distribution = Categorical(action_probs.squeeze(0)) # Squeeze
batch dim for Categorical

    action = distribution.sample()

    return action.item(), distribution.log_prob(action)

# -----

# Training Functions (Modified for Metrics)

# -----

def train_model(trial: optuna.trial.Trial) -> float:

    """

    Train the TransformerScheduler model for Optuna, focusing on
    success ratio.

    (Metrics collection is minimal here, focus is on hyperparameter
    tuning objective).

    """

    # Hyperparameter tuning

    learning_rate = trial.suggest_float('lr', 1e-6, 1e-3, log=True)

    # hidden_dim_trial = trial.suggest_categorical('hidden_dim',
[64, 128, 256]) # Example of tuning other params

    # num_layers_trial = trial.suggest_int('num_layers', 1, 3) #
Example

    hidden_dim_trial = HIDDEN_DIM # Keep fixed for now unless tuning

    logger.info(f"Trial {trial.number}: Learning
Rate={learning_rate}, Hidden Dim={hidden_dim_trial}")

    # Initialize the model, optimizer

```

```

    # The input_dim is NUM_SITES + 1 because state 0 might be
    initial, and 1..NUM_SITES represent sites

    model = TransformerScheduler(input_dim=NUM_SITES + 1,
    hidden_dim=hidden_dim_trial, output_dim=NUM_SITES)

    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    successes = 0

    total_steps = 0

    current_state = 0 # Initial state can be 0

    for epoch in range(NUM_EPOCHS):

        epoch_successes = 0

        log_probs = []

        rewards = []

        # Minimal ping caching within an epoch for Optuna speed

        ping_cache: Dict[int, Tuple[bool, Optional[float]]] = {}

        for step in range(STEPS_PER_EPOCH):

            action, log_prob = choose_action(current_state, model)

            # Use cache if available for this epoch

            if action not in ping_cache:

                success, rtt = ping_site(SITES[action])

                ping_cache[action] = (success, rtt)

            else:

                success, rtt = ping_cache[action]

            # Reward based purely on RTT (as before) for RL
objective
            reward_value = (1000.0 - rtt) if success and rtt is not
None else 0.0

            log_probs.append(log_prob)

            rewards.append(reward_value)

            if success:

```

```

        epoch_successes += 1

        current_state = action + 1 # Next state is the index of
the chosen site + 1 (to avoid 0)

        # Update policy (using simple REINFORCE update)

        optimizer.zero_grad()

        # Calculate policy gradient loss

        policy_loss = []

        for log_prob, reward in zip(log_probs, rewards):

            policy_loss.append(-log_prob * reward) # Standard
REINFORCE loss

        # Sum losses and backpropagate

        loss = torch.stack(policy_loss).sum() # Sum losses for the
epoch

        if torch.isnan(loss) or torch.isinf(loss):

            logger.warning(f"Trial {trial.number}, Epoch {epoch+1}:
Invalid loss detected ({loss.item()}). Skipping update.")

        else:

            try:

                loss.backward()

                # Optional: Gradient clipping

                #
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                optimizer.step()

            except RuntimeError as e:

                logger.error(f"Trial {trial.number}, Epoch
{epoch+1}: Runtime error during backward/step: {e}")

                # Potentially skip this epoch's update or handle
more gracefully

        successes += epoch_successes

        total_steps += STEPS_PER_EPOCH

```

```

        # Optuna pruning check (optional)

        trial.report(successes / total_steps, epoch)

        if trial.should_prune():

            raise optuna.exceptions.TrialPruned()

        # Reduced logging during Optuna runs

        if (epoch + 1) % 100 == 0:

            logger.info(f"Trial {trial.number}, Epoch {epoch + 1},
Running Success Ratio: {successes / total_steps:.2%}")

        final_success_ratio = successes / total_steps

        logger.info(f"Trial {trial.number} finished. Final Scheduling
Success Ratio: {final_success_ratio:.2%}")

        # Optuna expects the metric to maximize/minimize

        return final_success_ratio

def final_training(best_params: Dict[str, float]):

    """

    Perform final training with the best hyperparameters, collect
detailed metrics,

    and visualize the results including the final summary.

    Args:

        best_params (Dict[str, float]): Best hyperparameters found
by Optuna.

    """

    learning_rate = best_params['lr']

    # Use best hidden_dim if tuned, otherwise use default

    # hidden_dim_final = best_params.get('hidden_dim', HIDDEN_DIM)

    # num_layers_final = best_params.get('num_layers', 2)

    hidden_dim_final = HIDDEN_DIM # Keep fixed as per original if
not tuned

```

```

    logger.info(f"Starting final training with Learning
Rate={learning_rate}, Hidden Dim={hidden_dim_final}")

    # Initialize the model, optimizer

    model = TransformerScheduler(input_dim=NUM_SITES + 1,
hidden_dim=hidden_dim_final, output_dim=NUM_SITES)

    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Initialize device trackers

    rl_devices = [RLDevice(ip=SITES[i], name=SITE_NAMES[SITES[i]])
for i in range(NUM_SITES)]

    # Lists for plotting overall trends

    epoch_avg_loss_list = []

    epoch_avg_reward_list = []

    epoch_success_ratios = []

    start_time = time.time()

    total_steps = 0

    total_successes = 0

    current_state = 0 # Initial state

    logger.info(f"Starting final training loop for {NUM_EPOCHS}
epochs...")

    for epoch in range(NUM_EPOCHS):

        epoch_total_loss = 0

        epoch_total_reward = 0

        epoch_successes = 0

        log_probs = []

        rewards = []

        # No ping cache during final training to get real-time data
for metrics

        # ping_cache: Dict[int, Tuple[bool, Optional[float]]] = {} #
uncomment if needed for performance

        for step in range(STEPS_PER_EPOCH):

```

```

        step_start_time = time.time()

        total_steps += 1 # Increment total rounds/steps counter

        action, log_prob = choose_action(current_state, model)

        chosen_device = rl_devices[action]

        # Ping the selected device

        success, rtt = ping_site(chosen_device.ip)

        # Update metrics for the chosen device

        chosen_device.update_metrics(success, rtt,
step_start_time)

        # Calculate reward for RL update (based on RTT)

        reward_value = (1000.0 - rtt) if success and rtt is not
None else 0.0

        log_probs.append(log_prob)

        rewards.append(reward_value)

        if success:

            epoch_successes += 1

            total_successes += 1

            epoch_total_reward += reward_value

            current_state = action + 1 # Next state is index + 1

            # Minimal delay between steps if needed, otherwise rely
on ping time

            # time.sleep(0.1) # Optional small delay

        # --- Policy Update for the Epoch ---

        optimizer.zero_grad()

        policy_loss = []

        for lp, r in zip(log_probs, rewards):

            # Check for NaN/Inf in reward, though less likely with
current scheme

            if not np.isnan(r) and not np.isinf(r):

```

```

        policy_loss.append(-lp * r)

    else:

        logger.warning(f"Epoch {epoch+1}: Invalid reward
({r}) encountered. Skipping loss term.")

        policy_loss.append(torch.tensor(0.0)) # Add zero
loss if reward is invalid

    if not policy_loss: # Handle case where all rewards might
have been invalid

        logger.warning(f"Epoch {epoch+1}: No valid rewards,
skipping policy update.")

        avg_loss = 0.0 # Or handle as appropriate

    else:

        loss = torch.stack(policy_loss).sum()

        epoch_total_loss = loss.item() # Store loss value before
potential issues

        if torch.isnan(loss) or torch.isinf(loss):

            logger.warning(f"Epoch {epoch+1}: Invalid total
loss ({loss.item()}). Skipping backward/step.")

            avg_loss = epoch_total_loss / STEPS_PER_EPOCH # Use
stored value for logging

        else:

            try:

                loss.backward()

                # Optional:
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                optimizer.step()

                avg_loss = epoch_total_loss / STEPS_PER_EPOCH

            except RuntimeError as e:

                logger.error(f"Epoch {epoch+1}: Runtime error
during backward/step: {e}")

```



```

        avg_loss = epoch_total_loss / STEPS_PER_EPOCH #
Log the loss value even if update failed

    # --- Logging and Storing Epoch Results ---

    avg_reward = epoch_total_reward / STEPS_PER_EPOCH

    current_success_ratio = total_successes / total_steps if
total_steps > 0 else 0.0

    epoch_avg_loss_list.append(avg_loss)

    epoch_avg_reward_list.append(avg_reward)

    epoch_success_ratios.append(current_success_ratio)

    # Log progress periodically

    if (epoch + 1) % 50 == 0 or epoch == 0:

        logger.info(f"Epoch {epoch + 1}/{NUM_EPOCHS}, Avg Loss:
{avg_loss:.2f}, Avg Reward: {avg_reward:.2f}, Overall Success Ratio:
{current_success_ratio:.2%}")

        # Optional: Log per-device stats periodically

        # summary = ", ".join([f"{d.name}:
{d.packets_received}/{d.packets_sent} pkts,
AvgThr={d.average_throughput_bps:.0f}bps" for d in rl_devices])

        # logger.info(f"  Device Stats: {summary}")

# --- Final Summary ---

execution_time = time.time() - start_time

    final_success_ratio = total_successes / total_steps if
total_steps > 0 else 0.0

    logger.info("\n--- Experiment Summary ---")

    logger.info(f"Duration: {execution_time:.1f} seconds")

    logger.info(f"Total rounds (steps): {total_steps}") # Changed
'Total rounds' to 'Total rounds (steps)' for clarity

    for device in rl_devices:

        success_rate = (device.packets_received /
device.packets_sent * 100) if device.packets_sent > 0 else 0.0

        logger.info(f"{device.name} ({device.ip}):")

```

```

        logger.info(f"  Packets: {device.packets_received} received
/ {device.packets_sent} sent ({success_rate:.1f}%)")

        logger.info(f"  Avg throughput:
{device.average_throughput_bps:.2f} bps")

        # PF metric is specific to PF scheduler, removed here.

        if device.delays: # Check if any delays were recorded

            logger.info(f"  Delay (last {len(device.delays)}):
avg={statistics.mean(device.delays):.2f}ms,
min={min(device.delays):.2f}ms, max={max(device.delays):.2f}ms")

        elif device.packets_sent > 0:

            logger.info(f"  Delay: No successful pings recorded.")

        else:

            logger.info(f"  Delay: Device was never selected.")

    logger.info(f"Final RL Scheduling Success Ratio:
{final_success_ratio:.3%}") # Keep RL specific metric too

    # --- Visualization ---

    visualize_results(

        sites=SITES,

        site_names=SITE_NAMES,

        rl_devices=rl_devices, # Pass device data for potential
visualization

        learning_rate=learning_rate,

        epoch_avg_loss_list=epoch_avg_loss_list,

        epoch_avg_reward_list=epoch_avg_reward_list,

        epoch_success_ratios=epoch_success_ratios,

        execution_time=execution_time

    )

    # -----

    # Visualization (Modified)

```

```

# -----

def visualize_results(

    sites: List[str],

    site_names: Dict[str, str],

    rl_devices: List[RLDevice],

    learning_rate: float,

    epoch_avg_loss_list: List[float],

    epoch_avg_reward_list: List[float],

    epoch_success_ratios: List[float],

    execution_time: float

):

    """

    Visualize training results, plotting each graph individually
    with font size 16.

    Args:

        sites (List[str]): List of site IPs.

        site_names (Dict[str, str]): Mapping IP to device name.

        rl_devices (List[RLDevice]): List of device objects with
        final metrics.

        learning_rate (float): Learning rate used for training.

        epoch_avg_loss_list (List[float]): List of average loss
        values per epoch.

        epoch_avg_reward_list (List[float]): List of average reward
        values per epoch.

        epoch_success_ratios (List[float]): List of overall success
        ratios per epoch.

        execution_time (float): Total execution time in seconds.

    """

    num_epochs_run = len(epoch_avg_loss_list)

    epoch_nums = range(1, num_epochs_run + 1)

```

```

# Set global font size for plots

plt.rcParams.update({'font.size': 16})

# --- Plot 1: Final Average Latency per Site ---

plt.figure(figsize=(12, 7)) # Adjusted size for readability

final_latency = []

site_display_names = []

colors = cm.tab10.colors[:len(sites)] # Use tab10 for distinct
colors

for i, device in enumerate(rl_devices):

    site_display_names.append(f"{device.name}\n({device.ip})")

    # Use the calculated avg_delay if available, otherwise
    indicate no data or high latency

    if device.avg_delay is not None:

        final_latency.append(device.avg_delay)

    elif device.packets_sent > 0 and device.packets_received ==
0 :

        final_latency.append(PING_TIMEOUT * 1000 * 1.1) #
Indicate failure with value slightly above timeout

        logger.warning(f"Site {device.name} had no successful
pings, plotting high latency.")

    else: # Never selected

        final_latency.append(0) # Or None - plotting 0 might be
misleading, let's use a marker? Or skip? Let's use high val too

        final_latency.append(PING_TIMEOUT * 1000 * 1.1)

        logger.warning(f"Site {device.name} was never selected,
plotting high latency.")

plt.bar(site_display_names, final_latency, color=colors)

plt.title('Final Average Latency (RTT) per Site')

plt.xlabel('Site')

plt.ylabel('Average Latency (RTT in ms)')

```

```

plt.xticks(rotation=45, ha='right') # Rotate labels for better
fit

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()

plt.show()

# --- Plot 2: Training Loss ---

plt.figure(figsize=(10, 6))

plt.plot(epoch_nums, epoch_avg_loss_list, label='Avg Training
Loss per Epoch', color='blue')

plt.title('Training Loss over Epochs')

plt.xlabel('Epoch')

plt.ylabel('Average Loss')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()

plt.show()

# --- Plot 3: Rewards ---

plt.figure(figsize=(10, 6))

plt.plot(epoch_nums, epoch_avg_reward_list, label='Avg Reward
per Epoch', color='green')

plt.title('Average Reward over Epochs')

plt.xlabel('Epoch')

plt.ylabel('Average Reward')

plt.legend()

plt.grid(True, linestyle='--', alpha=0.7)

plt.tight_layout()

plt.show()

# --- Plot 4: Reliability (Success Ratio) ---

plt.figure(figsize=(10, 6))

```

```

    plt.plot(epoch_nums, epoch_success_ratios, label='Overall
Success Ratio', color='red')

    plt.title('Overall Reliability (Ping Success Ratio) over
Epochs')

    plt.xlabel('Epoch')

    plt.ylabel('Success Ratio')

    plt.ylim(0, 1.05) # Ensure y-axis goes from 0 to 1

    plt.legend()

    plt.grid(True, linestyle='--', alpha=0.7)

    plt.tight_layout()

    plt.show()

# --- Plot 5: Final Average Throughput per Site ---

plt.figure(figsize=(12, 7)) # Adjusted size

    final_throughput = [dev.average_throughput_bps for dev in
rl_devices]

    plt.bar(site_display_names, final_throughput, color=colors)

    plt.title('Final Average Throughput per Site')

    plt.xlabel('Site')

    plt.ylabel('Average Throughput (bps)')

    plt.xticks(rotation=45, ha='right')

    plt.grid(axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()

    plt.show()

# -----

# Main Execution

# -----

def main():

    """

```

```

    Main function to perform hyperparameter optimization (optional)
    and final training.

    """

    # --- Optuna Optimization (Optional) ---

    # Uncomment the following lines to run Optuna first

    # logger.info("Starting Optuna hyperparameter search...")

    # study = optuna.create_study(

    #     direction='maximize', # Maximize success ratio

    #     pruner=optuna.pruners.MedianPruner(n_warmup_steps=50) #
    Example pruner

    # )

    # study.optimize(train_model, n_trials=10, timeout=600) # Limit
    trials/time

    # # Display the best trial from Optuna

    # best_trial = study.best_trial

    # logger.info("\n--- Optuna Results ---")

    # logger.info(f'Best trial value (Success Ratio):
    {best_trial.value:.2%}')

    # logger.info(f'Best hyperparameters: {best_trial.params}')

    # best_params = best_trial.params # Use best params found

    # --- Direct Final Training (Use this if not running Optuna) ---

    logger.info("Skipping Optuna, using default hyperparameters for
    final training.")

    # Define default or pre-selected parameters if Optuna is skipped

    best_params = {'lr': 1e-4} # Example: Set a default learning
    rate

    # --- Perform Final Training ---

    logger.info("\n--- Starting Final Training ---")

    final_training(best_params=best_params)

    logger.info("\nRL Scheduler Experiment Finished.")

```

```
if __name__ == "__main__":

    # Ensure the script can run icmpIib without root if possible

    # Note: Raw socket access might still require privileges
    depending on the OS.

    # If you encounter permission errors, you might need to run with
    sudo

    # or configure capabilities (e.g., `setcap cap_net_raw+eip
    $(which python)`

    # Check icmpIib documentation for details.

    logger.info("Attempting to run with standard user privileges.")

    main()
```