information

Article

ORAN-HAutoscaling: A Scalable and Efficient Resource Optimization Framework for Open Radio Access Networks with Performance Improvements

Sunil Kumar [1,2]

[1] School of Computing and Digital Media, London Metropolitan University, Holloway Rd, London N7 8DB, UK; sk0064@surrey.ac.uk
[2] Institute for Communication Systems, University of Surrey, Guildford GU2 7XH, UK

Abstract: Open Radio Access Networks (ORANs) are transforming the traditional telecommunications landscape by offering more flexible, vendor-independent solutions. Unlike previous systems, which relied on rigid, vertical configurations, ORAN introduces network programmability that is AI-driven and horizontally scalable. This shift is facilitated by modern container orchestrators, such as Kubernetes and Red Hat OpenShift, which simplify the development and deployment of components such as gNB, CU/DU, and RAN Intelligent Controllers (RICs). While these advancements help reduce costs by enabling shared infrastructure, they also create new challenges in meeting ORAN's stringent latency requirements, especially when managing large-scale xApp deployments. Near-RTRICs are responsible for controlling xApps that must adhere to tight latency constraints, often less than one second. Current orchestration methods fail to meet these demands, as they lack the required scalability and long latencies. Additionally, non-API-based E2AP (over SCTP) further complicates the scaling process. To address these challenges, we introduce ORAN-HAutoscaling, a framework designed to enable horizontal scaling through Kubernetes. This framework ensures that latency constraints are met while supporting large-scale xApp deployments with optimal resource utilization. ORAN-HAutoscaling dynamically allocates and distributes xApps into scalable pods, ensuring that central processing unit (CPU) utilization remains efficient and latency is minimized, thus improving overall performance.

Keywords: scalability; cloud; high performance; high computing; ORAN; E2 agent; load balance; near-RTRIC; non-RTRIC

Academic Editor: Aneta Poniszewska-Maranda

Received: 15 December 2024
Revised: 2 February 2025
Accepted: 23 February 2025
Published: 23 March 2025

Citation: Kumar, S. ORAN-HAutoscaling: A Scalable and Efficient Resource Optimization Framework for Open Radio Access Networks with Performance Improvements. *Information* **2025**, *16*, 259. https://doi.org/10.3390/info16040259

1. Introduction

The emergence of 6G networks represents a transformative shift in global connectivity, pushing beyond current limitations of bandwidth, latency, energy efficiency, scalability, and performance. Designed to revolutionize telecommunication infrastructures, 6G is expected to facilitate the interconnection of billions of devices, from sensors to computational devices, mobile phones, and machines, enabled by artificial intelligence and machine learning applications (AI/ML) [1,2]. This technology is poised to play an important role in various innovative use cases such as autonomous systems, smart energy grids, advanced imaging, robotics, extended reality, and telemedicine [3,4]. To meet the requirements of these use cases, 6G must offer features such as ultralow latency, scalability, high performance, large bandwidth, and widespread connectivity across various devices and networks [5–7].

As these needs grow, there is a pressing requirement for flexible and scalable frameworks to support the coexistence of various services, each with unique demands [8]. Traditional ORAN architectures are not equipped to handle the full scope of these requirements, making ORAN systems essential to unlock new opportunities in both the private and academic sectors [9]. In this context, Software-Defined Networking (SDN) and Network Function Virtualization (NFV) have been increasingly adopted to enhance the programmability, intelligence, and energy efficiency of radio access networks (RANs), ensuring they meet specific service demands [10–12]. Moreover, the segmentation of RAN functions into distinct layers helps improve scalability, reliability, and adaptability, aligning with the varied needs of next-generation services [13,14].

The demand for more agile, resource efficient, scalable, and cost-effective cellular networks that can still guarantee that high data throughput and low latency is driving the industry's transition toward the cloudification and orchestration of key RAN components such as the Near-Real-Time RAN Intelligent Controller (near-RTRIC), Non-Real-Time RAN Intelligent Controller (non-RTRIC), and the overall RAN [15,16]. This shift leverages the virtualization principles that have long been integral to cloud computing, SDN, and NFV, providing enhanced flexibility for designing, deploying, and managing cellular networks [17]. It allows for the dynamic monitoring, optimization, and reconfiguration of network components through software, facilitating scalable and real-time network management [5,18,19].

The ORAN initiative and the formation of the ORAN Alliance have further supported this paradigm shift by promoting an open, cloud-based architecture for cellular networks [20–22]. This architecture enables interoperability between multi-vendor hardware and software components and integrates AI/ML for predictive load management, key performance tracking, and anomaly detection [23,24]. AI also plays a critical role in optimizing RAN functionalities, such as spectrum usage and traffic classification [25]. To facilitate the adaptable deployment of 5G and 6G networks, ORAN introduces near-RTRIC and non-RTRIC, which serve as platforms for hosting third-party AI-driven network functions. Near-RTRIC handles real-time xApps for network monitoring and control, typically within a 1-second latency window, while non-RTRIC supports rApps for longer inference loops. These components are interconnected via the A1 interface, with dApps providing microservices for extremely low-latency inference within 10 milliseconds [26,27]. The benefits of this modular approach are manifold: it supports dynamic reconfiguration of the RAN to meet current demands, reduces the total cost of ownership by enabling shared infrastructure, and optimizes resource utilization through on-demand scalability [28]. However, this shift to cloud-based orchestration and virtualization introduces new challenges, particularly in managing the complexities of scalability, resource allocation, and maintaining stringent latency requirements for AI/ML-based applications. Issues related to the E2 Application Protocol (E2AP) between xApps and RIC, which uses SCTP protocol, complicate scaling efforts, while some AI/ML solutions struggle to meet the necessary latency and performance guarantees, especially in near-RTRIC and non-RTRIC loops.

While extensive research has addressed timing constraints in virtualized RANs, the control plane, especially for near-RTRICs, still faces significant challenges in ensuring timely decision-making to avoid outdated network decisions. These challenges highlight the need for further innovation to meet the strict performance requirements in future ORAN deployments [29]. Mismanagement of O-Cloud environments hosting rApps, xApps, and dApps can lead to violations of control deadlines. Figures 1–3 are derived from experimental data collected through a controlled testbed setup. We used this testbed to simulate various scenarios, and the figures reflect the results of these controlled experiments. The experimental setup and results presented in Figures 1–3 are based on the repository available through a GitLab repository [30]. As shown in Figure 1, two critical metrics contribute to these violations: (i) queuing time, which is the duration it takes for the near-

RTRIC to extract data from the RAN and pass it to an xApp, and (ii) execution time, which reflects the processing time needed for the xApp to produce an output, including ML model inference. Figure 2 demonstrates how the total time (sum of queuing and execution times) increases as the number of xApps running on the near-RTRIC rises. The data presented come from tests on an ORAN-compliant FlexRIC deployed on a Kubernetes. The goal is to keep control within the one-second near-RTRIC time window (shown as shaded areas in Figure 2). However, when the number of CPU-intensive xApps exceeds 35, Kubernetes fails to meet the latency requirements, which is a conservative estimate for real-world applications with hundreds of base stations. Another issue is inefficient CPU utilization. As shown in Figure 3, the system crashes after 35 xApps, yet only 73% of the CPU is used. This means 27% of the CPU remains underutilized, which is crucial since efficient computing directly impacts operational costs. Thus, simply adding more virtual machines is not an ideal solution. Instead, both horizontal and vertical scaling need to be applied to fully utilize CPU and memory resources, ensuring cost savings while maintaining latency and inference time within acceptable limits.

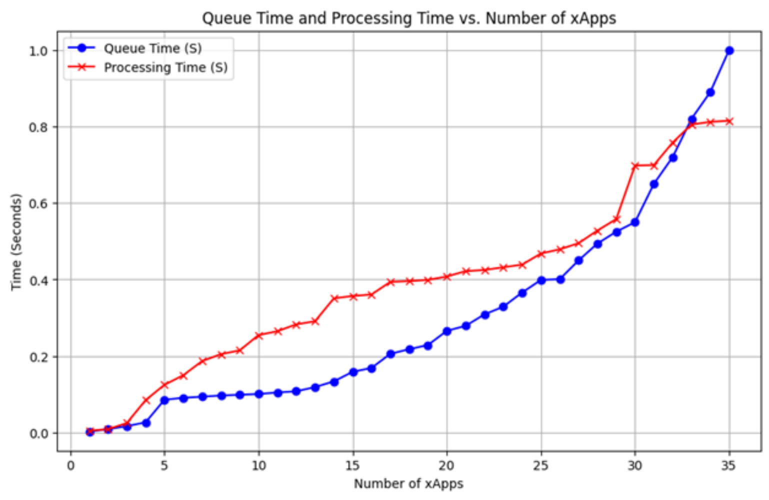


Figure 1. Queuing vs. processing times for different xApp numbers.

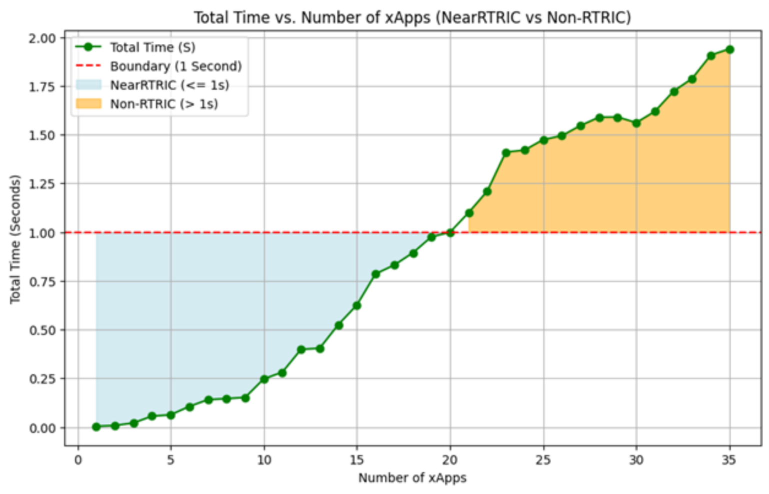


Figure 2. Total time vs. number of xApps.

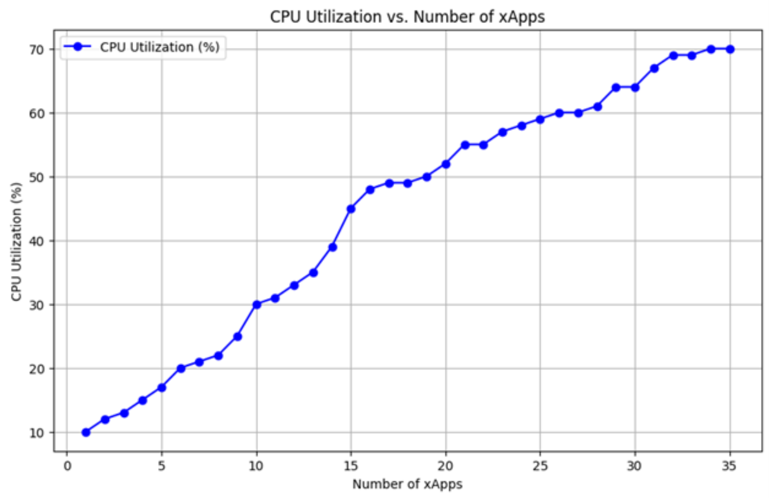


Figure 3. CPU utilization vs. number of xApps.

Traditional scaling methods based on Central Processing Unit (CPU) and Random Access Memory (RAM) metrics are insufficient for ensuring the performance of time-sensitive applications like those in ORAN [31–33]. To address this, we propose a novel integrated ORAN-HAutoscaling framework that combines proactive CPU-aware scaling with a load-balancing mechanism based on latency and inference time. The system uses a probing mechanism to gather real-time data through APIs, enabling dynamic scaling decisions. By employing both horizontal and vertical scaling, the system can adapt to varying load conditions. The inclusion of the probe mechanism optimizes load balancing by applying the Bull and Bear (BnB) rule, which synchronizes probing to ensure fresh load data. The key innovation of our framework is its ability to preemptively scale when CPU usage exceeds 80% or when latency nears a critical threshold. This predictive scaling helps maintain a low tail latency (below 1 s), even during peak loads, while also optimizing resource use to reduce energy consumption and costs. To validate this approach, we used Grafana to visualize CPU utilization and latency metrics, confirming that the system consistently meets performance standards while supporting a large number of xApps.

By addressing the trade-offs between latency, resource utilization, and operational efficiency, the ORAN-HAutoscaling framework improves scalability and responsiveness for time-sensitive applications in ORAN environments. Near-RTRIC plays a central role in the ORAN architecture, hosting xApps for monitoring and controlling RAN infrastructure [34–36]. These xApps manage distributed RAN components, such as eNB, gNB, Open Contol Unit (O-CU), and Open Distributed Unit (O-DU), via the E2 protocol on the southbound interface [37,38]. near-RTRIC communicates with non-RTRIC through the A1 and O1 interfaces to optimize and manage RAN functions across diverse network types [39,40]. xApps rely on the E2 interface to gather near-real-time data, which near-RTRIC uses to make optimization decisions [41–43]. As defined by the ORAN Alliance, near-RTRIC provides a database function for storing configuration data, ML tools for data pipelining, messaging infrastructure, logging and metrics collection, and security functions. It also resolves conflicts that may arise from requests from different xApps [44–46]. As demand for advanced network services rises, near-RTRIC is under increasing pressure to handle higher traffic and a growing number of xApps [47]. Traditional RICs like FlexRIC struggle to manage this load due to limited scalability and inefficient resource use [48,49]. To tackle this, we propose ORAN-HAutoscaling, a hybrid solution that supports containerization, orchestration, and load balancing, enabling efficient resource management in FlexRIC and other RICs. Unlike API-based solutions for near-RTRICs, ORAN-HAutoscaling uses

E2AP-based communication between xApps and RICs, overcoming the challenges posed by the SCTP protocol.

ORAN-HAutoscaling dynamically adjusts the number of RIC instances (pods) based on CPU utilization and latency, optimizing resource allocation. The system uses advanced technologies like Docker, Kubernetes, and native load balancing with tunneling and routing to facilitate horizontal scaling. The key components of ORAN-HAutoscaling include a native load balancer to reduce latency and select servers based on CPU load, latency, and streams in flight. The Bull and Bear (BnB) rule improves load balancing by combining these signals and ensuring the freshness of the load data. Our framework optimizes CPU usage and tail latency under peak load conditions, ensuring that the system remains responsive even as mean CPU usage increases. The results demonstrate that ORAN-HAutoscaling supports a larger number of xApps compared to traditional FlexRIC, efficiently manages resources, and meets latency requirements.

The contributions of this work include:

- Introducing ORAN-HAutoscaling, a dynamic framework for managing and supporting large numbers of xApps, rApps, and dApps while optimizing CPU and latency.
- Conducting a comprehensive data collection study to assess the impact of a large number of xApps, CPU resource sharing, and scaling on latency and inference times, creating a data-driven latency model for optimization.
- Implementing ORAN-HAutoscaling and conducting extensive experiments on an ORAN-compliant testbed, comparing it with traditional FlexRIC solutions. The results show that ORAN-HAutoscaling efficiently manages scaling, maintains latency thresholds, and achieves high CPU utilization.

2. Problem Statement

Near-RTRIC is critical and struggles with capacity limitations and performance degradation under a heavy load, leading to crashes and connection failures [50–55]. The re-intialization of connection and xApp impacts the performance and downtime. This is unacceptable given the essential role of near-RTRIC in ORAN. One potential solution involves deploying near-RTRIC on multiple machines. However, this approach is inefficient, as it fails to optimize resource utilization during periods of low traffic and may still be inadequate during peak traffic [56,57]. Therefore, the implementation of ORAN-HAutoscaling offers a promising approach to dynamically manage near-RTRIC instances. The traditional load balancer lacks support for the Stream Control Transmission Protocol (SCTP), which is essential for near-RTRIC operations with a latency threshold. ORAN-HAutoscaling uses a naive load balance system using the tunneling and routing concept to handle real-time SCTP connections while maintaining latency within acceptable limits [58,59]. This research paper aims to solve scalability challenges and to design and develop ORAN-HAutoscaling, which optimizes resources while maintaining high computing capacity and performance.

3. Related Work

Virtual Machine (VM) and microservice dynamic scaling has been widely studied in the last decades [60]. For instance, ref. [61] built a deadline-aware scheduler to control application latency in a serverless environment. Ref. [62] modeled virtual workloads with a focus on deadlines and costs, but they were more applicable to long-running applications rather than real-time control situations, and workload modeling approachesw were also mismatched. Ref. [63] explored autoscaling solutions to fulfill deadlines for simulation workloads, whereas [64] employed a token bucket scheme to scale resources and satisfy query deadlines in relational databases. In contrast, this paper focuses on tight control timelines, where energy minimization, or profit maximization subject to QoS constraints, and years of experience scaling resources under the same QoS requirements are modeled in a quadratic-constrained quadratic programming (QCQP) model that uses detailed RAN control workloads for input.

Currently, ORAN is abstracting the network infrastructure and providing the following cloud services to cellular network functions, allowing the separation of network function-oriented scaling solutions from cloud computing-oriented scaling solutions well into the future [65,66]. However, many of them are not guaranteed to provide bounded latency for closed-loop control [67]. For example, in the ORAN setting, ref. [68] investigated proactive resource scaling for virtual network functions (VNFs) based on workload prediction, whereas [69] centered on application orchestration rather than scaling or energy efficiency. Ref. [70] formulated a scheme for keeping coherence among base stations and users that maximizes network throughput and saves resources when low computing ability exists. Ref. [71] proposed proactive scaling to accommodate network slices. However, these studies mainly focus on optimal placement and execution of services over ORAN infrastructures, and they do not guarantee control latency or minimize energy consumption.

In recent years, proactive autoscaling techniques have gained significant attention to address the limitations of traditional reactive methods. For example, Chen et al. [72] introduced a deep learning-based forecasting model for network slice scaling, leveraging historical traffic trends to preemptively allocate resources. Similarly, Patel et al. [73] employed a Markov decision process-based predictive autoscaling approach, demonstrating reduced response time variability compared to reactive methods. These proactive strategies significantly enhance system stability by minimizing sudden workload spikes, but they often require computationally expensive model training and retraining, which is a limitation in real-time ORAN environments. Our work differs in that it combines a lightweight predictive model with dynamic scheduling, ensuring both efficiency and responsiveness in RAN control workloads.

Alternative orchestration frameworks have also been explored to enhance network function management. Kubernetes has become a dominant orchestration tool for cloud-native workloads, with researchers integrating it with ORAN components for efficient resource allocation. Zhang et al. [74] proposed reinforcement learning-based Kubernetes scheduling for optimizing real-time resource scaling in ORAN. While Kubernetes-based orchestration offers flexibility, it lacks native support for stringent QoS requirements, leading to potential latency issues. On the other hand, OpenStack Tacker [75] provides policy-driven scaling for VNFs, allowing fine-grained control over network function lifecycles, though it often suffers from longer instantiation times.

The virtualized ORAN is always demanding for energy efficiency. Prior works have focused on energy usage in RAN, which is the most power-hungry part of cellular systems, as well as VNFs like core networks and RICs. Ref. [76] investigated power consumption trade-offs in virtualized RAN, whereas Pamuklu et al. developed a mixed-integer linear programming model to optimize the energy of the RAN, considering the maximum acceptable delays in the RAN's data plane. Ref. [77] proved that dynamic power control under a centralized controller can minimize the power consumption of RAN.

Despite these advancements, existing methods often overlook the interplay between energy efficiency and real-time control constraints. While Nguyen et al. [78] explored unikernels for lightweight virtualization in edge computing, achieving improved resource utilization, their approach is less suitable for highly dynamic RAN environments. Similarly, Lopez et al. [79] compared Kubernetes with ETSI MANO for network function orchestration, highlighting the trade-offs between flexibility and performance overhead. Our proposed ORAN-HAutoscaling framework is the first to jointly model compute scaling, resource optimization, and timing constraints for RAN control in ORAN, underpinned by experimental ML inference characterization of control workloads and a working prototype.

4. ORAN-HAutoscaling: Scalable Framework ORAN Architecture

The scalable framework's architecture is shown in Figure 4, alongside its integration with the ORAN system. The framework gathers requests from an Service Management and Orchestration (SMO) system through a control interface that network operators use to submit requests for AI-based ORAN applications. The SMO system collects requests and, at configurable intervals (T seconds), forwards them to the Scalable Framework ORAN's (SFORAN) Non-RTRIC optimization engine. The engine processes these requests for deploying rApps, xApps, and dApps. Once the SFORAN receives the requests, it calculates an optimal policy to meet the demands made in those requests and then deploys the rApps, xApps, and dApps from an app catalog. Several significant operations of xApp and scaling management are included in the SFORAN prototype, as shown in detail in Figure 5. The step-by-step procedure is as follows:

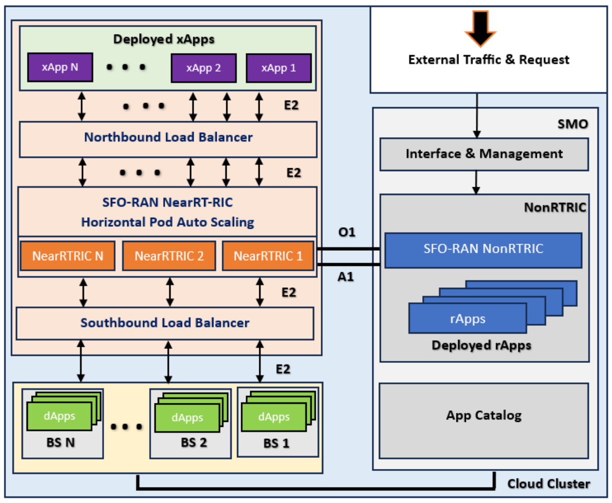


Figure 4. Scalable framework architecture.

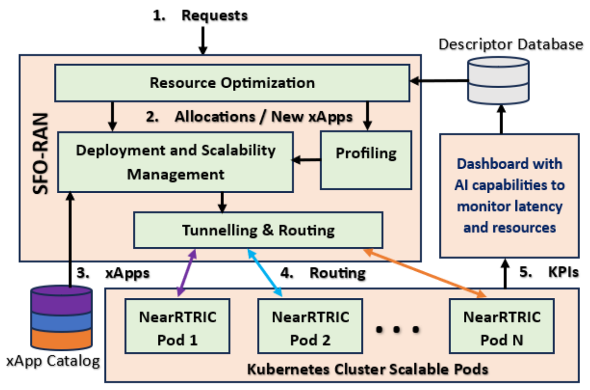


Figure 5. SFORAN architecture.

(1) *Request handling:* The SMO layer receives xApp deployment requests and forwards them to the SFORAN to check the xApp catalog for the availability.

(2) *xApp profiling and benchmarking:* If the xApp does not exist in the catalog and does not have an app descriptor, it is profiled and benchmarked by deploying on an idle worker node to know its performance requirements. With descriptors defined in

xApps, the SFORAN calculates the right allocation policy according to the collected APIs and metrics.

(3) *xApp Deployment:* The SFORAN deployment engine retrieves the necessary xApp from the catalog, allocates it to the available node, taking into account latency constraints and resource availability, and then balances the load allocation of it to the near-RTRIC pods.

(4) *Resource allocation:* The SFORAN automatically adjusts the number of near-RTRIC pods based on CPU usage and latency to ensure proper resource utilization.

(5) *Latency and resource monitoring:* Kubernetes periodically report run-time latency and resource usage metrics to the SFORAN control interface, which feeds back into the optimization loop for further resource adjustments. We have implemented the load balancer service type at northbound and southbound to support the scalability. The high-level view and communication between the different components are presented in the Figure 6.

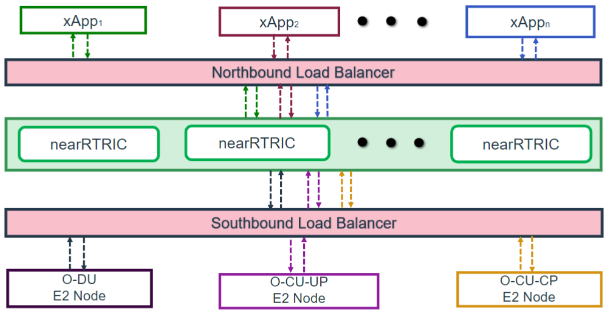


Figure 6. Scalability with load balancer service type.

5. System Model and Terminology

In our deployment model, Kubernetes serves as the backbone for managing the near-RTRIC components and associated applications. Each Near-RTRIC component is deployed as a containerized application within a Kubernetes environment. The deployment YAML configuration specifies a set number of replicas for each pod, which ensures availability and fault tolerance. The resource requests and limits ensure efficient utilization of CPU and memory resources, thus avoiding overloading of the servers hosting the pods. The service configuration exposes the application to the network through a load balancer, ensuring that external traffic is efficiently distributed among the available pods. The Horizontal Pod Autoscaler (HPA) provides the dynamic scaling capabilities required for efficient resource management in our architecture. With the HPA configuration defined in HPA.yaml, the deployment automatically scales between 1 and 3 replicas based on CPU utilization, maintaining optimal performance without manual intervention. The target CPU utilization is set to 80%, ensuring that the system responds to fluctuating workloads by adjusting the pod count as necessary. The deployment of near-RTRIC components is designed for elasticity, ensuring that the system can handle dynamic workloads and scale according to the demand. Using Kubernetes' LoadBalancer service type, the system can efficiently distribute traffic across pods, ensuring that all instances of a given service receive a fair share of incoming requests.

The system operates using a set of S servers, where $|S|$ denotes the number of these servers. Although, in theory, these servers could support both RAN functions and RICs, practical observations reveal that data-intensive ORAN applications, such as network function processes, consume significant resources, including CPU and RAM. This high resource demand often leads to server overloads during application execution. To overcome these challenges, the near-RTRIC is deployed on dedicated servers, represented by the set S, within containerized environments. This setup ensures the reliability and availability of critical networking functions. By abstracting the physical hardware and operating within a Kubernetes cluster, this design allows applications to scale independently. For example, near-RTRIC pods can dynamically adjust their scale, expanding or contracting based on CPU usage without necessitating new server deployments.

Additionally, servers located alongside CU/DU components, identified as $S_{\text{CU/DU}} \subseteq S$, are optimized for hosting dApps. The ORAN-HAutoscaling mechanism facilitates the efficient deployment of applications and the dynamic scaling of computational resources within a single cluster. In scenarios involving multiple clusters (C), ORAN-HAutoscaling can deploy C separate instances to manage each cluster individually.

5.1. ORAN High-Traffic Applications

The rApps, xApps, and dApps available to tenants are organized into an application catalog, referred to as A, and stored on the non-RTRIC. The total number of applications is given by $|A|$. These applications primarily focus on AI-based functionalities. For clarity, we define the catalog as $A = A_{\text{rApp}} \cup A_{\text{xApp}} \cup A_{\text{dApp}}$, where each application $a \in A$ includes a descriptor outlining its key features. The application descriptor specifies the functionality the app provides, such as RAN slicing or traffic steering. It also details the type of AI model it employs, such as Deep Reinforcement Learning (DRL), Long Short-Term Memory (LSTM) networks, or Convolutional Neural Networks (CNNs). Furthermore, the descriptor categorizes the app (e.g., xApp, rApp, dApp) and includes the structure and format of its input and output data, such as a list of input Key Performance Indicators (KPIs), data shapes, types of actions the app performs, and the format of these actions. In addition to these details, the descriptor provides information about the app's performance profile, including latency characteristics and resource requirements like CPU and memory usage, which are further elaborated in the subsequent section.

5.2. External Tenant Requests

Tenants who utilize the shared ORAN infrastructure may have different objectives, business goals, and service requirements, each governed by distinct Service-Level Agreements (SLAs). To meet these needs, tenants submit requests to deploy specific rApps, xApps, and dApps from the application catalog, denoted as A. Let R represent the set of all requests made by tenants. Each request $r \in R$ is represented by a tuple $r = (n_r, L_r, \delta_r)$, where $n_r = (n_{r,a})_{(r,a)\in R\times A}$ denotes the number of application instances required, $L_r = (L_{r,a})_{(r,a)\in R\times A}$ specifies the maximum acceptable latency for each application, and $\delta_r = (\delta_{r,a,s})_{(r,a,s)\in R\times A\times S}$ indicates the server allocation for each application.

The term $n_{r,a}$ represents the number of instances of application $a \in A$ needed to fulfill request r. Meanwhile, $L_{r,a}$ denotes the maximum allowable latency for running application a on any server. For example, a tenant may request three instances of an xApp ($n_{r,a'} = 3$) to manage RAN slicing for three Distributed Units (DUs) with a maximum allowable inference time of $L_{r,a'} = 100$ ms. Similarly, they might request one instance of an rApp ($n_{r,a''} = 1$) for handover management, with an acceptable total latency of $L_{r,a''} = 10$ s. In general, it is not ideal to control multiple RAN components with a single xApp or rApp, as doing so may lead to conflicts, congestion, and excessive latency. Therefore, we assume that $n_{r,a} \geq 1$.

In cases where tenants do not impose strict inference time constraints (i.e., $L_{r,a} = +\infty$), the near-RTRIC is designed to make decisions within 1 s, while dApps typically operate within a 10 ms window. Thus, we define a latency constraint $L_{\text{APP},a}$ for each application type. For instance, if $a \in A_{\text{xApp}}$, we set $L_{\text{APP},a} = 1$ second, and for $a \in A_{\text{dApp}}$, we set $L_{\text{APP},a} = 10$ ms. Since ORAN specifications do not provide a maximum inference time for rApps, we assume that $L_{\text{APP},a} = +\infty$ for $a \in A_{\text{rApp}}$. Additionally, we define $\delta_{r,a,s} \in \{0,1\}$

to specify whether a dApp $a \in A_{\text{dApp}}$ must execute on server s co-located with a CU/DU. If $\delta_{r,a,s} = 1$, then the dApp must be executed on that server; otherwise, we set $\delta_{r,a,s} = 0$ for all other applications.

For clarity, we define the server activation profile $x = (x_s)_{s \in S}$, where $x_s \in \{0,1\}$ indicates whether server s is actively hosting at least one AI-based ORAN application ($x_s = 1$) or not ($x_s = 0$). To monitor the allocation of applications across servers, we introduce the variable $y = (y_{r,a,s})_{(r,a,s) \in R \times A \times S}$, representing the number of instances of application a for request r that are deployed on server s. For each request r and application a, the variables $y_{r,a,s}$ satisfy the following constraint:

$$\Delta_{r,a} = \left\{ (y_{r,a,s})_{s \in S}, y_{r,a,s} \in \mathbb{Z}_0^+ \middle| \sum_{s \in S} y_{r,a,s} = n_{r,a} \right\}, \tag{1}$$

where $\mathbb{Z}_0^+$ denotes the set of non-negative integers. A server s is considered activated ($x_s = 1$) if and only if $\sum_{r \in R} \sum_{a \in A} y_{r,a,s} > 1$.

We also define an auxiliary indicator variable $w_{r,s} \in \{0,1\}$ for all $r \in R$ and $s \in S$, where $w_{r,s} = 1$ if at least one instance of an application required by request r is hosted on server s, i.e., $\sum_{a \in A} y_{r,a,s} > 0$.

Furthermore, we introduce the binary variable $z_r \in \{0,1\}$ for each $r \in R$, indicating whether the allocation of applications meets all requirements for request r, both in terms of deployment and latency constraints. If the allocation satisfies all conditions, we set $z_r = 1$; otherwise, $z_r = 0$. Additionally, we define $\pi_{a,s} \in \{0,1\}$ to indicate whether server s hosts at least one instance of application a. Specifically, $\pi_{a,s} = 1$ if $\sum_{r \in R} y_{r,a,s} > 0$, and $\pi_{a,s} = 0$ otherwise. The total number of different applications hosted on server s is given as follows:

$$A_s = \sum_{a \in A} \pi_{a,s}. \tag{2}$$

Finally, we summarize the key variables as follows: $z = (z_r)_{r \in R}$, $w = (w_{r,s})_{(r,s) \in R \times S}$, and $\pi = (\pi_{a,s})_{(a,s) \in A \times S}$.

6. Total Time (Latency and ML Inference) and CPU Metrics Model For ORAN Applications

To meet the required total time constraints, we first develop a latency model that governs both the scaling and deployment of applications. This ensures that all applications are able to complete their respective tasks within the specified time limits. This section presents the results of a data collection effort using the ORAN-HAutoscaling prototype, which provides insights into how congestion and resource sharing affect the total time across various ORAN components and AI algorithms.

6.1. Inference Time Profiling

The inference time for AI-driven ORAN applications is primarily determined by the complexity of the AI models embedded in the dApps, xApps, and rApps. Key factors include model depth and width, number of layers, and the use of convolution operations. As shown in Figure 1, when multiple applications are running concurrently on the same hardware, they share computational resources, leading to increased total time due to resource contention. To accurately evaluate the impact of shared resources on the total time, particularly with respect to inference time, a model is needed that can effectively capture these dynamics. Within ORAN systems, AI techniques serve various functions, such as classification (e.g., anomaly detection), forecasting (e.g., KPI prediction), and control (e.g., resource allocation). While multiple AI architectures could be used for these tasks (such as CNNs or decision trees for classification), our analysis focuses on three distinct and commonly used models designed for each task. Specifically, we use a CNN with 231,875 parameters and a fully connected output layer for classification, an LSTM model with 49,987 parameters and bidirectional memory cells for forecasting, and a DRL agent with over 50,000 parameters for control-related tasks.

The main goal of our study is to develop a model for total time that helps scale intelligent ORAN applications. To achieve this, we focus on analyzing latency and inference time, which remain constant regardless of whether the AI model has been trained. This consistency occurs because the number of operations (such as multiplications, convolutions, and additions) in the AI models stays fixed. For our evaluation, we use a single node within the cluster, where one xApp instance is deployed at a time. To gather a large amount of data, we implemented an E2 traffic generator based on an open-source ORAN dataset. This generator simulates E2 traffic by continuously extracting random KPIs from the dataset in a format compatible with the expected input for the xApp AI models. Each application descriptor specifies the required KPIs and the format of the input for each xApp model. When a new xApp instance is deployed on a server s, the traffic generator supplies the necessary input data, allowing us to measure three types of latency:

- *Queuing tim e* $t_{a,s}^{queue}$, which is the time taken for the xApp to process the input after it reaches the E2 termination point on the Near-RTRIC.
- *Execution time* $t_{a,s}^{exec}$, which represents the time taken to generate the output once the input is processed.
- *Inference time* $t_{a,s}^{inf} = t_{a,s}^{queue} + t_{a,s}^{exec}$, which is the total time for processing the input and producing the output.

In addition to these latencies, we monitor the CPU and RAM utilization of the server during this process. While it is possible to account for the time needed to transfer KPIs and control actions between the RAN and RICs, these factors remain constant, as all servers are part of the same cluster. Based on these assumptions, we define the inference time when y instances of application $a \in A$ are running on servers as follows:

$$t_{a,s}^{inf}(y) = t_{a,s}^{exec}(y) + t_{a,s}^{queue}(y), \tag{3}$$

where $t_{a,s}^{exec}(y)$ refers to the execution time, and $t_{a,s}^{queue}(y)$ represents the queuing time for application a on server s.

6.2. Latency Model

Figure 7 illustrates how total time varies with CPU utilization and the type and number of xApps. We observe that when deploying 80 xApps of the KPM type, the CPU usage reaches 91%, which leads to application crashes beyond that point. However, there is still 9% CPU capacity remaining unused, which highlights the importance of modeling execution time based on CPU usage. While RAM usage provides some insight, it is not as reliable for predicting total time, since different models might consume the same amount of RAM but process data at different speeds.

In Figure 8, we observe that another type of xApp, which is more computationally intensive, can only run 45 instances before it consumes 75% of the CPU. Beyond this point, the applications begin to crash. Despite having 25% of the CPU available, these xApps consume large portions of the CPU (around 28%), leaving insufficient resources for additional applications. In contrast, other types of xApps that consume less CPU can be run in greater numbers without issues. This observation points to the need for a framework that efficiently manages and allocates xApps to the most appropriate near-RTRIC pod. This motivates the use of CPU metrics for better resource management. Sometimes, applications crash because they fail to meet the required xApp latency. To address these limitations, we focus on measuring both execution time $t_{a,s}^{exec}$ and queuing time $t_{a,s}^{queue}$ to derive a comprehensive total time model. In Figures 1 and 2, we present the results obtained by instantiating y instances of the different xApps at the same time. Our analysis of execution and queuing times revealed that the system's performance is highly sensitive to queuing delays, especially when the number of xApps increases. While the near-RTRIC can maintain low execution and queuing times under moderate loads, performance significantly degrades as the number of xApps grows. This highlights the critical role of dynamic resource allocation and autoscaling in mitigating queuing delays and ensuring that execution times remain within the strict latency requirements for near-RT RAN functions. The findings emphasize that bottlenecks are primarily caused by queuing inefficiencies rather than execution limitations, underscoring the need for scalable frameworks like ORAN-HAutoscaling to effectively manage resource contention and maintain real-time operational efficiency.

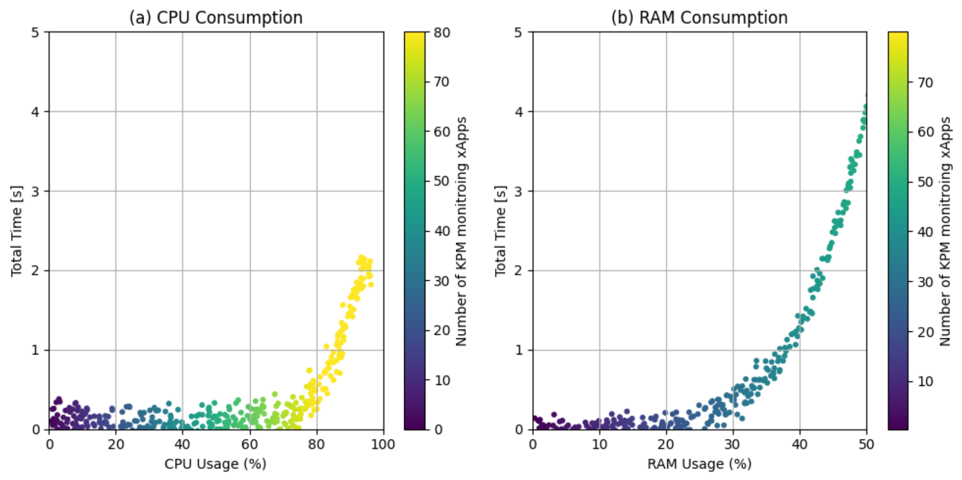


Figure 7. CPU and RAM consumption of Key Performance Measurement (KPM) xApps.

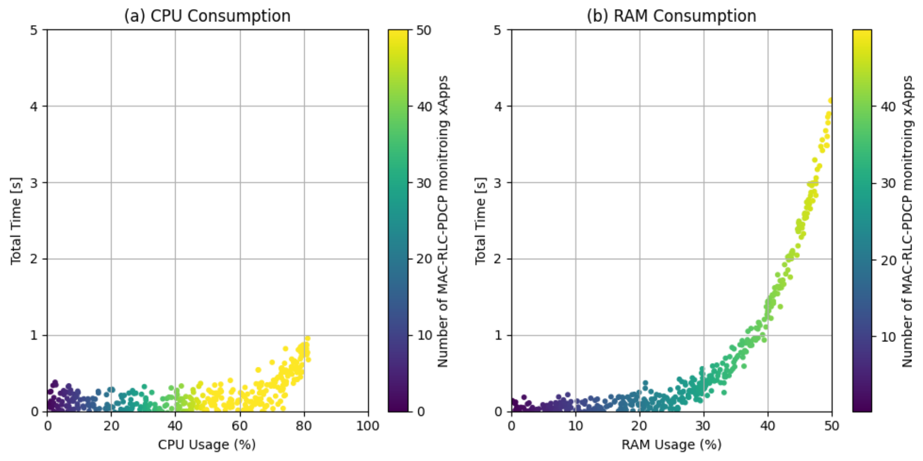


Figure 8. CPU and RAM consumption of MAC-RLC-PDCP xApps.

Generally, we found that execution time $t^{exec}(y)$ has a significant impact on inference time when the number of applications y is small. However, as congestion increases with larger y values, queuing time $t^{queue}(y)$ becomes the dominant factor. This suggests that inference time can be modeled as a function with two distinct phases: a moderate increase in inference time as the number of applications grows, followed by a steeper increase due to congestion. While various approaches like linear regression or neural networks can be used to model these functions, our goal is to develop a model that is both accurate and easy to integrate into an optimization framework while minimizing the risk of underestimating inference time to meet latency constraints. To achieve this, we employ piecewise linear regression to model inference time, which has several advantages: it is flexible enough to approximate non-linear functions, it simplifies optimization problems, and it reduces complexity.

While the minimum number of segments for the approximation could be determined using piecewise linearization methods, our analysis suggests that inference time behaves like an "elbow" function. Therefore, we use two-segment piecewise linear regression to model the function:

$$f(y) = \begin{cases} \lambda_1 \cdot y + b_1, & \text{if } y \leq y_0, \\ \lambda_2 \cdot y + b_2, & \text{if } y > y_0, \end{cases} \tag{4}$$

where y_0 is the breakpoint, and λ_i and b_i represent the slope and intercept of the i-th segment.

We found that the inference time $t_{a,s}^{inf}(y)$ can be approximated using the following two-segment piecewise linear function:

$$t_{a,s}^{\text{inf}}(y) = \begin{cases} \lambda_{a,s}^{I} \cdot y + b_{a,s}^{I}, & \text{if } y \leq \tilde{y}_{a,s}, \\ \lambda_{a,s}^{II} \cdot y + b_{a,s}^{II}, & \text{otherwise.} \end{cases} \tag{5}$$

Here, $\tilde{y}_{a,s}$ is the breakpoint, and λ_i, b_i are the slope and intercept for segment $i \in \{I, II\}$.

Using data collected from our prototype, we extracted values for $\tilde{y}_{a,s}$, $\lambda_{a,s}^{I}$, and $\lambda_{a,s}^{II}$. We also applied piecewise linearization to the inference time function for ML-based control xApps in two scenarios: an average fit that approximates general behavior, and a conservative fit that accounts for upper bounds through piecewise linear bounding. While both fits capture the elbow-shaped behavior, the average fit could underestimate inference time, potentially violating latency constraints. The conservative fit, on the other hand, considers measurement variance, particularly when a high number of applications are deployed.

We now extend our application-specific model to scenarios where multiple instances of different applications are hosted on the same server. While our measurement campaign focused on profiling xApps, the same latency model can be applied to dApps or rApps. Let $y_{r,a,s}$ represent the number of instances of application type a from request r running on server s. The total inference time for all instances executing on server s is given as follows:

$$l_s(y, \pi) = \frac{1}{A_s} \sum_{a \in A} \tilde{t}_{a,s}^{inf}(Y_s)\pi_{a,s}, \tag{6}$$

where $Y_s = \sum_{r \in R} \sum_{a \in A} y_{r,a,s}$ is the total number of application instances hosted on server $s \in S$, $\tilde{t}_{a,s}^{inf}(\cdot)$ is defined in Equation (5), and A_s from Equation (2) is a function of π. Equation (6) models the expected inference time when multiple instances of different applications are running on the same server.

7. Objective Function Design

To mathematically prove the efficacy of scaling in time-sensitive applications using latency, inference time, and CPU utilization, we propose a model based on queueing theory and control theory. The model aims to optimize resource allocation in real time and ensure performance within latency constraints. Below, we detail the components of the model:

7.1. Queueing Theory Model (M/M/c Queue)

We employ a simplified M/M/c queue model, where the arrival process follows a Poisson distribution, and the system size is assumed to be infinite. This approximation allows for tractable analysis of resource allocation and scaling in multi-server systems.

However, we acknowledge that real-world traffic patterns might deviate from Poisson, especially with bursty traffic, which could affect the system's performance. In future work, we will explore more advanced models, such as G/G/c, to address these deviations.

The system is modeled as a multi-server queue, where the arrival rate is denoted by λ, the service rate per server by μ, and there are c active near-RTRIC pods representing horizontal scaling. The average waiting time in the system is represented as follows:

$$W_q = \frac{\lambda}{\mu(c\mu - \lambda)} \tag{7}$$

This equation helps model system latency, as it scales up (when adding a new pod reduces W_q) and scales down when idle. The latency threshold L_{max} represents the maximum tolerable latency for time-sensitive applications, ensuring that:

$$W_q \leq L_{max} \tag{8}$$

7.2. Control Theory: Proportional Integral Derivative (PID) Controller

We employ a PID controller to regulate scaling based on real-time CPU and latency data. The error $e(t)$ is defined as the difference between the desired latency L_{max} and the observed latency $L_{observed}(t)$:

$$e(t) = L_{max} - L_{observed}(t) \tag{9}$$

The control action $u(t)$ for scaling can be derived using the following PID equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{10}$$

This feedback mechanism adjusts resources (pods) in real time, keeping the latency within acceptable limits.

7.3. CPU Scaling Conditions

We define specific conditions for horizontal scaling based on CPU utilization:

- Scale-up condition: When CPU utilization exceeds 80%, a new pod is added to handle the increased load:

$$\text{if } CPU_{observed} \geq 0.8 \times CPU_{max}, \quad \text{scale up by adding a new pod.}$$

- Scale-down condition: If CPU utilization stays idle (below a threshold) for 180 s, a pod is removed, but at least one pod must always remain active:

$$\text{if } CPU_{observed} \leq 0.1 \times CPU_{max} \text{ for 180 s}, \quad \text{scale down by removing a pod.}$$

$$\text{Ensure at least one pod is always running: } c \geq 1.$$

These conditions ensure that resources are efficiently managed based on CPU load while preventing under provisioning. The values for the thresholds and time period were determined based on extensive experimentation with YAML configuration files. The YAML file configuration is given in Table 1. We experimented with a range of CPU utilization thresholds (70%, 80%, 90%) and time periods (120 s, 180 s, 240 s) to identify the optimal values. These experiments revealed that the thresholds of 80%, along with a time period of 180 s, minimized unnecessary scaling events while maintaining acceptable latency levels.

However, CPU utilization alone does not fully capture performance bottlenecks such as latency spikes or inference delays. Therefore, the combined metric C is introduced to make smarter scaling decisions.

Table 1. Configuration parameters from YAML files.

Parameter	Value
Deployment YAML (expric-deployment.yaml)	
replicas	1
containerPort (1)	36,421
containerPort (2)	36,422
imagePullPolicy	IfNotPresent
requests.cpu	100 m
requests.memory	128 Mi
limits.cpu	600 m
limits.memory	1500 Mi
selector.app	expric
ports.protocol	SCTP
Horizontal Pod Autoscaler YAML (expric-hpa.yaml)	
scaleTargetRef.apiVersion	apps/v1
scaleTargetRef.kind	Deployment
scaleTargetRef.name	expric-deployment-1
minReplicas	1
maxReplicas	3
targetCPUUtilizationPercentage	80%
idleTimeBeforeScaleDown	180 s

7.4. Combined Metric

We define a combined scaling metric C that incorporates CPU utilization, latency, and inference time:

$$C = \alpha \cdot \frac{CPU_{observed}}{CPU_{max}} + \beta \cdot \frac{L_{observed}}{L_{max}} + \gamma \cdot \frac{T_{inference}}{T_{max}} \tag{11}$$

The linear function was chosen due to its simplicity and computational efficiency, which are crucial for real-time decision making in ORAN systems. It allows for the intuitive and interpretable combination of critical metrics (CPU utilization, latency, and inference time) into a single value, with weights (α, β, and γ) providing flexibility to prioritize specific factors. While non-linear alternatives (e.g., multiplicative or exponential models) could be considered, they would introduce unnecessary complexity and increased computational overhead without offering significant improvements in performance. This makes the linear function an optimal choice for balancing simplicity and effectiveness in time-sensitive scaling decisions.

Scaling up occurs when C exceeds a predefined threshold, $C_{\text{threshold}}$, and scaling down occurs when C is below a certain lower bound, ensuring system scalability while preventing resource contention. The value of $C_{\text{threshold}}$ is 0.8. Using this framework, we demonstrate that incorporating CPU utilization, latency, and inference time into scaling decisions ensures optimal resource utilization without exceeding latency thresholds.

7.5. Integration with Prometheus and Kubernetes APIs

The latency models, scaling metrics, and control mechanisms are integrated into Kubernetes using Prometheus and custom controllers. Prometheus collects time-series data on resource usage, such as CPU utilization, latency, and inference time, from the Kubernetes cluster through Prometheus exporters and custom metrics adapters. These collected data are then made available to Kubernetes via the Prometheus adapter, allowing Kubernetes controllers to use these metrics for autoscaling decisions. The custom metrics, including the combined scaling metric C, which incorporates CPU utilization, latency, and inference time, are used by the Horizontal Pod Autoscaler (HPA). When C exceeds a predefined threshold $C_{\text{threshold}}$, the controllers trigger scaling actions. The scaling thresholds, such as CPU utilization and memory limits, are specified in the YAML configuration files (Table 1), ensuring the system operates within desired performance and resource limits.

7.6. Monitoring Performance Metrics

Prometheus, along with custom metrics adapters, continuously monitors important performance metrics such as CPU utilization, latency, and inference time. These metrics are collected in real time, including execution time $t_{a,s}^{exec}$, queuing time $t_{a,s}^{queue}$, and inference time $t_{a,s}^{inf}(y)$. Prometheus exposes these metrics to the Kubernetes API via the Prometheus adapter, enabling Kubernetes components like the Horizontal Pod Autoscaler (HPA) to make real-time scaling decisions based on these metrics. When resource utilization exceeds predefined thresholds—such as when CPU usage surpasses 80% or the combined scaling metric C (Equation (11)) exceeds the threshold—the HPA triggers scaling actions, ensuring the system remains responsive and scalable under varying workloads. This real-time monitoring ensures that scaling decisions are data-driven and responsive to the current state of the system.

7.7. Load Balancer with Bull and Bear

In the SFORAN system, the Kubernetes load balancer service type is used to ensure efficient traffic distribution across the Near-RTRIC pods. The load balancer uses a bull and bear analogy to represent the system's dynamic behavior under different load conditions.

7.7.1. Bull State (Normal Operation)

In the bull state, the system is stable, with a uniform distribution of traffic across the existing pods. The Kubernetes load balancer ensures that traffic is evenly spread based on pod availability and resource utilization. During this phase, the load is balanced effectively, and minor latency variations are handled smoothly, ensuring consistent performance.

7.7.2. Bear State (Scaling Up)

When the system detects increased load (e.g., high CPU utilization or latency spikes), it enters the bear state. In this state, HPA dynamically adds new pods to handle the increased traffic. The load balancer redirects traffic to the new pods, ensuring that the system remains stable and responsive under higher workloads. The addition of pods is triggered by Kubernetes' Horizontal Pod Autoscaler (HPA), which uses metrics from Prometheus (e.g., CPU usage, latency, inference time) to determine when scaling is necessary.

The load balancer adjusts the traffic distribution based on the scaling decisions made by the HPA, prioritizing the newly added pods to ensure they stabilize quickly and avoid overloading existing pods. This ensures that the system can handle an increased number of xApps or requests without degradation in latency or performance. This interplay between the bull and bear states through the Kubernetes load balancer service type allows for smooth autoscaling and consistent performance, as the system automatically adapts to the fluctuating load.

8. Performance Evaluation

We deployed ORAN-HAutoscaling on a Dell PowerEdge T550-Tower, which was equipped with a GPU, and ran the near-RTRIC container on Kubernetes. A range of system configurations was evaluated to test the performance of ORAN-HAutoscaling. Our results showed that the system effectively handled varying numbers of xApps, with CPU and RAM utilization remaining consistent with a Kubernetes setup featuring two cores, 6 GB of RAM, and 80 GB of storage.

For further analysis, we also used MATLAB (https://ww2.mathworks.cn/products/matlab.html) for numerical simulations, running them on a server equipped with an Intel Core i9-9980HK processor (64 cores, 128 GB RAM). These simulations, run over 80 trials, helped us evaluate the system's performance with different xApp numbers. The results were plotted to illustrate the system's behavior in various scenarios. Figure 9 shows the probability distribution for xApp requests of the same type and their corresponding ML inference times. This figure reveals that scaling ML models across multiple servers, in conjunction with xApps at the near-RTRIC, is key to enabling the scalability of these models. However, it also emphasizes that the inference time of ML models plays a crucial role in determining how many xApp requests the system can handle. For example, with a server $S = 5$, the system faces limitations in handling more than 800 xApp requests, whereas with $S = 25$, it can support up to 1600 requests. This scalability is adaptable to the different inference time profiles associated with diverse xApp request types and ML models. (See Table 2).

Table 2. Quantitative results for latency, resource utilization, and scalability.

Server (S)	Max xApp	Avg Latency (ms)	CPU Uti. (%)
5	800	250	60
10	1200	350	70
15	1400	500	80
25	1600	700	90

When limited to only $S = 5$ servers, real-time (RT) requests can only be accommodated up to 40 instances due to the stringent processing requirements of AI models. As expected, increasing the server count allows the system to effectively manage more RT and near-RT requests. These findings highlight that the scalability of AI solutions within ORAN systems is primarily determined by timing constraints and resource availability. The inference time significantly impacts how many dApps, xApps, and rApps can coexist on a single server. To facilitate this level of scalability within the near-RTRIC, horizontal pod scaling (HPA) is essential to ensure adequate resources are available to meet the demand.

Figure 10 compares ORAN-HAutoscaling against two alternative strategies: basic load balancing without scaling capabilities and a single server (no scaling, no load balancing) approach. In our tests, we set the number of servers to $S = 5$. In the load balancing approach, requests are distributed among servers, while the no scaling approach routes all requests to a single server. The single server no scaling and no balancing method struggled to handle the full volume of incoming requests due to resource limitations. In contrast, ORAN-HAutoscaling was able to dynamically scale up or down, adjusting the number of active pods as needed, and it successfully handled all requests. Furthermore, while the no-scaling method activated only a single server and the load balancing method used all available servers regardless of actual demand, ORAN-HAutoscaling optimized resource usage by only activating the necessary number of pods. This led to better CPU and RAM management and ensured higher reliability by preventing xApp crashes.

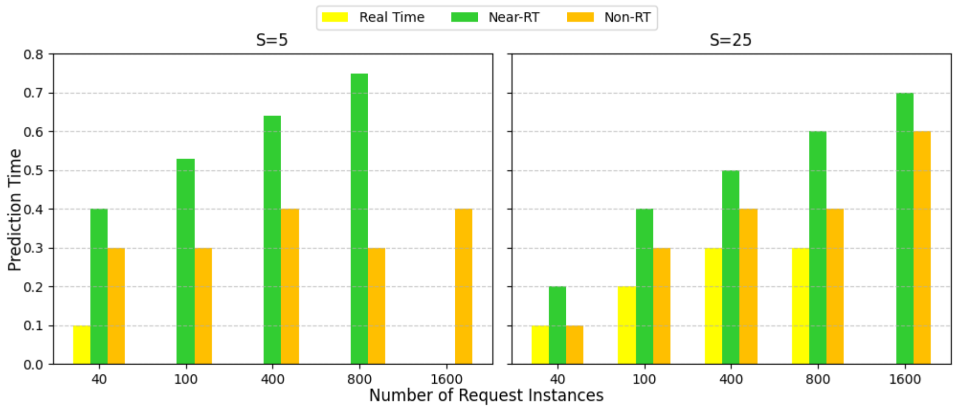


Figure 9. Probability of a server to host a request with a certain inference time profile for varying numbers of servers (S) and instances (I).

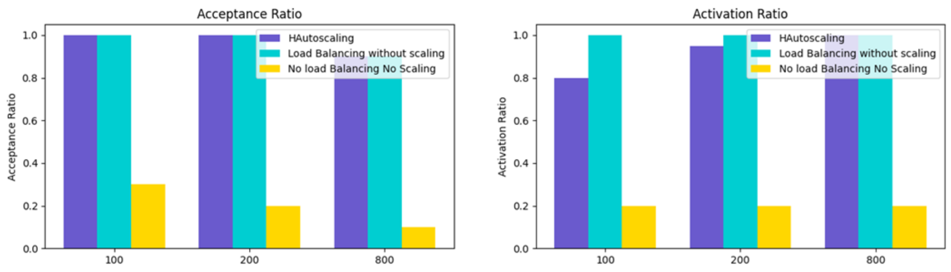


Figure 10. ORAN-HAutoscaling vs. non-scaling vs single server for S = 5 and varying numbers of requests.

ORAN-HAutoscaling's ability to efficiently manage the number of active pods and servers demonstrates its capacity to meet profit maximization goals while adhering to inference time constraints. This makes it a robust and scalable solution for ORAN systems, able to process requests effectively while maintaining stringent timing requirements.

The analysis also reveals that the no scaling approach is unsuitable for ORAN applications due to its high inference time. When compared to load balancing, ORAN-HAutoscaling not only provides more efficient CPU resource utilization but also guarantees lower inference times, which aligns with tenant timing requirements.

9. Experimental Results

In our experiments, we evaluated the performance of the ORAN-Autoscaling HPA system and compared it with OpenShift's non-scaling load-balancing approach and a single-server setup. The experimental setup mirrored the conditions discussed earlier, using the same three AI-based xApps, and deploying $I = 162$ xApp instances. For this evaluation, the hardware utilized consisted of a Dell PowerEdge T550 node with a GPU, which hosted the ORAN-Autoscaling system, xApp, and ML model deployment. Additionally, we considered a scenario with $R = 3$ tenants, each submitting a distinct request, namely r_1, r_2, and r_3, representing typical small-scale ORAN deployment. The near-RT inference time profiles ranged from 400 ms to 800 ms, with r_1 requiring the shortest inference time of 400 ms.

The evaluation focused on measuring CPU and RAM utilization over a 20-min period, comparing the performance of ORAN-HAutoscaling with the multiple servers with load balancer and a single-server setup. During the experiment, ORAN-HAutoscaling was able to accommodate all three requests (r_1, r_2, and r_3), deploying the full set of 162 xApps. In contrast, OpenShift handled only r_1 and r_2 requests, deploying a total of 82 xApps before it started crashing due to overutilization. The single-server setup, on the other hand, could only manage 40 xApps before failing.

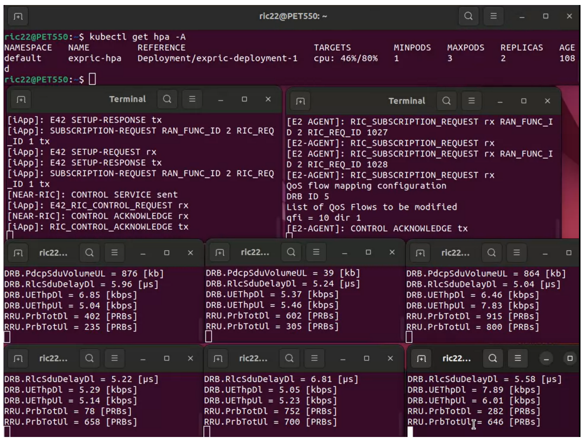
An important observation during the experiment was that OpenShift distributed xApps across nodes (Node1 and Node2) in a random fashion, without taking into account the actual resource utilization. In contrast, ORAN-HAutoscaling effectively managed requests by scaling the pods as necessary. Specifically, when CPU usage reached 60%, it automatically added another pod. The system evaluated the current CPU utilization, latency, and inference times before deciding how to distribute the xApps. For instance, on Pod1, 55% of the xApps were from r_1, and the remaining 45% came from r_2. Meanwhile, Pod3 exclusively hosted xApps from r_3.

This careful allocation ensured that all xApps from r_1 adhered to the 400 ms latency constraint, while Pod2 was introduced when CPU usage reached around 70% to further handle the load. The allocation process was intentionally slow, as we deployed one xApp at a time to gather accurate data. Through this method, ORAN-HAutoscaling successfully optimized resource usage while maintaining the required latency constraints.

In comparison, OpenShift failed to meet even the 1000 ms latency requirement due to its inability to allocate xApps based on their specific timing needs. This led to violations of ML inference time and xApp latency requirements, ultimately undermining the system's performance. On the other hand, ORAN-HAutoscaling not only ensured that all incoming requests were satisfied but also effectively allocated xApps to different pods, with Pod1 hosting all of r_1's xApps and 45% of r_2's, ensuring that the 400 ms latency was consistently met. Additionally, Pod2 guaranteed that the 1000 ms requirement for r_2 was satisfied.

Figure 11 illustrates the near-RTRIC container's HPA system, which shows the current CPU utilization at 46%. Initially, the system started with one pod, and as CPU usage crossed the 80% threshold, a second pod was automatically initiated. Consequently, the system was running two pods, with a CPU utilization of 46%.

Figure 12 illustrates the use of Prometheus and Grafana to visualize key metrics such as CPU, RAM, xApp latency, and other system statistics. These metrics are gathered and processed through APIs to help determine the optimal pod allocation for efficient resource utilization. Figure 13 displays the Kubernetes dashboard, which provides an overview of RAM and CPU usage across the containers, offering insights into resource consumption and helping to ensure proper system scaling.



```
ric22@PET550:~$ kubectl get hpa -A
NAMESPACE   NAME         REFERENCE                          TARGETS        MINPODS   MAXPODS   REPLICAS   AGE
default     expric-hpa   Deployment/expric-deployment-1   cpu: 46%/80%   1         3         2          108
d
ric22@PET550:~$
```

```
[iApp]: E42 SETUP-RESPONSE tx
[iApp]: SUBSCRIPTION-REQUEST RAN_FUNC_ID 2 RIC_REQ
_ID 1 tx
[iApp]: E42 SETUP-REQUEST rx
[iApp]: E42 SETUP-RESPONSE tx
[iApp]: SUBSCRIPTION-REQUEST RAN_FUNC_ID 2 RIC_REQ
_ID 1 tx
[NEAR-RIC]: CONTROL SERVICE sent
[iApp]: E42_RIC_CONTROL_REQUEST rx
[NEAR-RIC]: CONTROL ACKNOWLEDGE rx
[iApp]: RIC_CONTROL_ACKNOWLEDGE tx
```

```
[E2 AGENT]: RIC_SUBSCRIPTION_REQUEST rx RAN_FUNC_I
D 2 RIC_REQ_ID 1027
[E2-AGENT]: RIC_SUBSCRIPTION_REQUEST rx
[E2 AGENT]: RIC_SUBSCRIPTION_REQUEST rx RAN_FUNC_I
D 2 RIC_REQ_ID 1028
[E2-AGENT]: RIC_SUBSCRIPTION_REQUEST rx
QoS flow mapping configuration
DRB ID 5
List of QoS Flows to be modified
qfi = 10 dir 1
[E2-AGENT]: CONTROL ACKNOWLEDGE tx
```

```
DRB.PdcpSduVolumeUL = 876 [kb]
DRB.RlcSduDelayDl = 5.96 [μs]
DRB.UEThpDl = 6.85 [kbps]
DRB.UEThpUl = 5.04 [kbps]
RRU.PrbTotDl = 402 [PRBs]
RRU.PrbTotUl = 235 [PRBs]
```

```
DRB.PdcpSduVolumeUL = 39 [kb]
DRB.RlcSduDelayDl = 5.24 [μs]
DRB.UEThpDl = 5.37 [kbps]
DRB.UEThpUl = 5.46 [kbps]
RRU.PrbTotDl = 602 [PRBs]
RRU.PrbTotUl = 305 [PRBs]
```

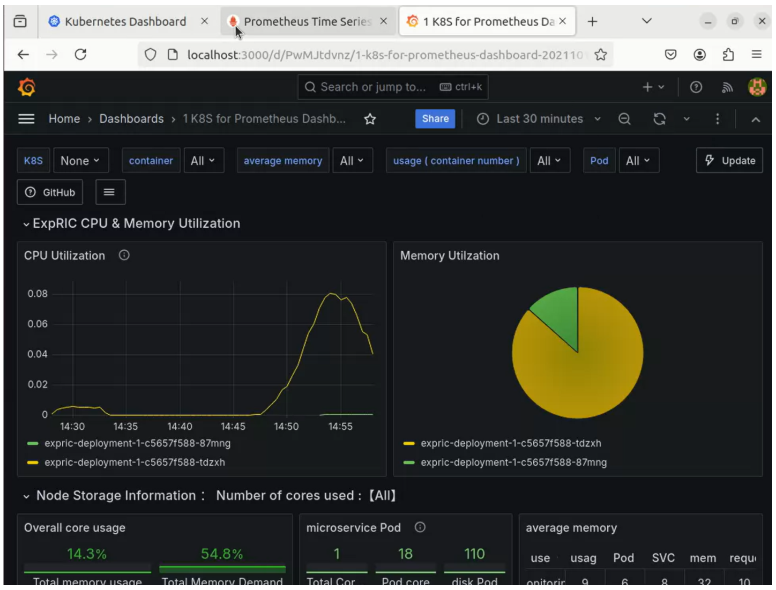
```
DRB.PdcpSduVolumeUL = 864 [kb]
DRB.RlcSduDelayDl = 5.04 [μs]
DRB.UEThpDl = 6.46 [kbps]
DRB.UEThpUl = 7.83 [kbps]
RRU.PrbTotDl = 915 [PRBs]
RRU.PrbTotUl = 800 [PRBs]
```

```
DRB.RlcSduDelayDl = 5.22 [μs]
DRB.UEThpDl = 5.29 [kbps]
DRB.UEThpUl = 5.14 [kbps]
RRU.PrbTotDl = 78 [PRBs]
RRU.PrbTotUl = 658 [PRBs]
```

```
DRB.RlcSduDelayDl = 6.81 [μs]
DRB.UEThpDl = 5.05 [kbps]
DRB.UEThpUl = 5.23 [kbps]
RRU.PrbTotDl = 752 [PRBs]
RRU.PrbTotUl = 700 [PRBs]
```

```
DRB.RlcSduDelayDl = 5.58 [μs]
DRB.UEThpDl = 7.89 [kbps]
DRB.UEThpUl = 6.01 [kbps]
RRU.PrbTotDl = 282 [PRBs]
RRU.PrbTotUl = 646 [PRBs]
```

Figure 11. xApps and HPA replicas.



Kubernetes Dashboard
Prometheus Time Series
1 K8S for Prometheus Da
localhost:3000/d/PwMJtdvnz/1-k8s-for-prometheus-dashboard-20211
Search or jump to...
ctrl+k
Home › Dashboards › 1 K8S for Prometheus Dashb...
Share
Last 30 minutes
K8S
None
container
All
average memory
All
usage (container number)
All
Pod
All
Update
GitHub
ExpRIC CPU & Memory Utilization
CPU Utilization
0.08
0.06
0.04
0.02
0
14:30
14:35
14:40
14:45
14:50
14:55
expric-deployment-1-c5657f588-87mng
expric-deployment-1-c5657f588-tdzxh
Memory Utilzation
expric-deployment-1-c5657f588-tdzxh
expric-deployment-1-c5657f588-87mng
Node Storage Information : Number of cores used : 【All】

Overall core usage

14.3% Total memory usage

54.8% Total Memory Demand

microservice Pod

1 Total Cor

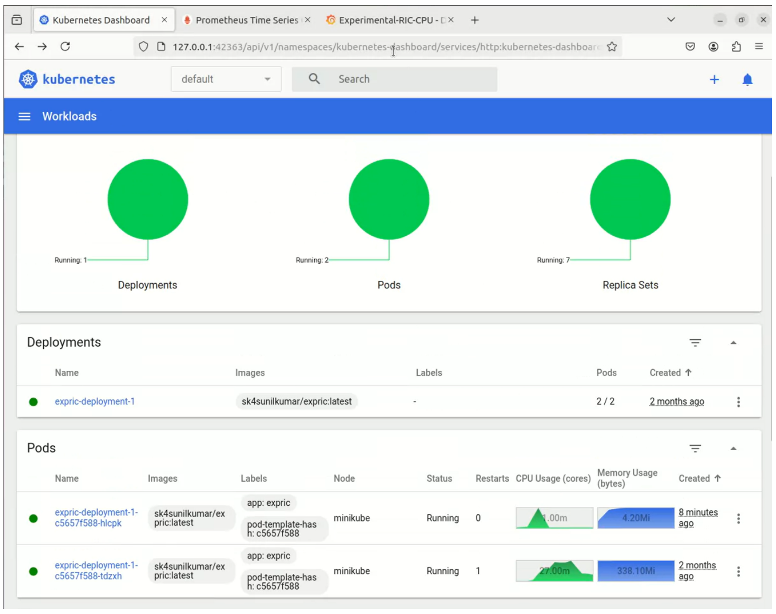
18 Pod core

110 disk Pod

average memory

use	usag	Pod	SVC	mem	requ
onitorir	9	6	8	32	10

Figure 12. API used for monitoring and controlling purposes.



Deployments

	Name	Images	Labels	Pods	Created ↑
●	expric-deployment-1	sk4sunilkumar/expric:latest	-	2 / 2	2 months ago

Pods

	Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	expric-deployment-1-c5657f588-hlcpk	sk4sunilkumar/expric:latest	app: expric pod-template-hash: c5657f588	minikube	Running	0	1.00m	4.20Mi	8 minutes ago
●	expric-deployment-1-c5657f588-tdzxh	sk4sunilkumar/expric:latest	app: expric pod-template-hash: c5657f588	minikube	Running	1	27.00m	338.10Mi	2 months ago

Figure 13. Running containers on Kuberentes.

10. Reflection

This study primarily leverages Kubernetes for orchestration, which offers flexibility and scalability for containerized applications. However, this dependency introduces certain limitations, particularly when integrating with non-containerized components in ORAN environments. While Kubernetes provides a robust platform for containerized microservices, adapting non-containerized legacy systems into this environment requires additional configuration and lead to integration challenges. Furthermore, the probing mechanisms utilized in this framework introduce some overheads, particularly in terms of CPU and memory consumption, as they monitor and adjust resource allocation. While these overheads are necessary for dynamic scaling, future optimizations could focus on reducing the frequency and granularity of these probes to minimize their impact on system performance. Looking ahead, several avenues for improvement can be explored. First, alternative orchestration frameworks, such as Docker Swarm and Apache Mesos, could be considered to determine their performance in comparison to Kubernetes, particularly in terms of scaling and management overheads. Additionally, improving load balancing algorithms by incorporating more advanced AI/ML techniques could enhance the efficiency of resource distribution, particularly under high-demand conditions. This would contribute to further optimizing resource allocation and minimizing delays in large-scale ORAN environments.

11. Conclusions

This paper presented ORAN-HAutoscaling, a dynamic scaling solution tailored for ORAN environments, designed to enforce CPU usage and time constraints while supporting a large number of xApps. We developed a latency model based on a measurement campaign conducted on a Kubernetes cluster, complemented by a mathematical optimization model and a prototype that aligns with ORAN standards. Through a series of comparisons with OpenShift's scaling mechanisms, we demonstrated that ORAN-HAutoscaling is capable of effectively deploying ORAN applications while meeting strict latency requirements. Our results show that by incorporating CPU metrics, latency, and inference time into the scaling decisions, ORAN-HAutoscaling can support a significantly higher number of xApps compared to traditional non-scaling solutions such as single-server setups. Specifically, ORAN-HAutoscaling can support approximately three times the number of xApps compared to these traditional systems. In conclusion, ORAN-HAutoscaling offers a robust, scalable solution for optimizing resource utilization in RT radio access networks. It guarantees high computing capacity and performance, meeting the increasing demands of future networks, and it outperforms conventional RIC solutions by ensuring minimal latency and maintaining ML inference times.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the author.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Garcia-Saavedra, A.; Costa-Pérez, X. ORAN: Disrupting the Virtualized RAN Ecosystem. *IEEE Commun. Stand. Mag.* **2021**, *5*, 96–103.
2. 3GPP. Study on New Radio Access Technology: Radio Access Architecture and Interfaces. In *3rd Generation Partnership Project (3GPP)*; 3GPP: Sophia Antipolis, France, 2017; TR 38.801 V14.0.0.
3. Mimran, D.; Bitton, R.; Kfir, Y.; Klevansky, E.; Brodt, O.; Lehmann, H.; Elovici, Y.; Shabtai, A. Evaluating the Security of Open Radio Access Networks. *arXiv* **2022**, arXiv:2201.06080 .

4. Klement, F.; Katzenbeisser, S.; Ulitzsch, V.; Krämer, J.; Stanczak, S.; Utkovski, Z.; Bjelakovic, I.; Wunder, G. Open or Not Open: Are Conventional Radio Access Networks More Secure and Trustworthy than Open-RAN? *arXiv* **2022**, arXiv:2204.12227.
5. Cho, J.Y.; Sergeev, A. Secure Open Fronthaul Interface for 5G Networks. In Proceedings of the 16th International Conference on Availability, Reliability and Security, Vienna, Austria, 17–20 August 2021; Ser. ARES 2021; Association for Computing Machinery: New York, NY, USA, 2021. [CrossRef]
6. *IEEE Std 802.1AE-2018*; IEEE Standard for Local and Metropolitan area networks-Media Access Control (MAC) Security; Revision of IEEE Std 802.1AE-2006. IEEE: Piscataway, NJ, USA, 2018; pp. 1–239.
7. O-RAN ALLIANCE Security Focus Group. Study on Security for Near Real Time RIC and xApps. In *ORAN Alliance e.V.*, Technical Specification v01.00; ORAN Alliance e.V.: Alfter, Germany, 2022.
8. Shen, C.; Xiao, Y.; Ma, Y.; Chen, J.; Chiang, C.-M.; Chen, S.; Pan, Y. Security Threat Analysis and Treatment Strategy for ORAN. In Proceedings of the 2022 24th International Conference on Advanced Communication Technology (ICACT), Pyeong Chang, Republic of Korea, 13–16 February 2022; pp. 417–422.
9. Cao, J.; Ma, M.; Li, H.; Ma, R.; Sun, Y.; Yu, P.; Xiong, L. A Survey on Security Aspects for 3GPP 5G Networks. *IEEE Commun. Surv. Tutorials* **2020**, *22*, 170–195.
10. Saad, W.; Bennis, M.; Chen, M. A vision of 6G wireless systems: Applications, trends, technologies, and open research problems. *IEEE Netw.* **2019**, *34*, 134–142.
11. Atya, A.O.F.; Qian, Z.; Krishnamurthy, S.V.; Porta, T.L.; McDaniel, P.; Marvel, L. Malicious Co-residency on the Cloud: Attacks and Defense. In Proceedings of the IEEE INFOCOM 2017-IEEE Conference on Computer Communications, Atlanta, GA, USA, 1–4 May 2017; pp. 1–9.
12. Gao, X.; Steenkamer, B.; Gu, Z.; Kayaalp, M.; Pendarakis, D.; Wang, H. A Study on the Security Implications of Information Leakages in Container Clouds. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 174–191.
13. Wang, Z.; Yang, R.; Fu, X.; Du, X.; Luo, B. A Shared Memory based Cross-VM Side Channel Attacks in IaaS Cloud. In Proceedings of the 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, USA, 10–14 April 2016; pp. 181–186.
14. Hardt, D. The OAuth 2.0 Authorization Framework. RFC Editor, RFC 6749. October 2012. Available online: https://www.rfc-editor.org/info/rfc6749 (accessed on 14 December 2024).
15. Jones, M.; Hardt, D. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC Editor, RFC 6750. October 2012. Available online: https://www.rfc-editor.org/info/rfc6750 (accessed on 14 December 2024).
16. Jones, N.S.M.; Bradley, J. JSON Web Token (JWT). RFC Editor, RFC 7519. May 2015. Available online: https://datatracker.ietf.org/doc/html/rfc7519 (accessed on 14 December 2024).
17. Microsoft. Sidecar Pattern- Azure Architecture Center. June 2022. Available online: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar (accessed on 26 June 2022).
18. Github-Pistacheio/Pistache: A High-Performance REST Toolkit Written in C++. Available online: https://github.com/pistacheio/pistache (accessed on 27 June 2022).
19. OpenAPI Generator. Available online: https://openapi-generator.tech/ (accessed on 5 June 2022).
20. Kim, B.; Sagduyu, Y.E.; Davaslioglu, K.; Erpek, T.; Ulukus, S. Channel-aware adversarial attacks against deep learning-based wireless signal classifiers. *IEEE Trans. Wirel. Commun.* **2020**, *21*, 3868–3880.
21. Joda, R.; Pamuklu, T.; Iturria-Rivera, P.E.; Erol-Kantarci, M. Deep Reinforcement Learning-Based Joint User Association and CU–DU Placement in ORAN. *IEEE Trans. Netw. Serv. Manag.* **2022**, *19*, 4097–4110. [CrossRef]
22. Li, P.; Thomas, J.; Wang, X.; Khalil, A.; Ahmad, A.; Inacio, R.; Kapoor, S.; Parekh, A.; Doufexi, A.; Shojaeifard, A.; et al. RLOps: Development Life-Cycle of Reinforcement Learning Aided Open RAN. *IEEE Access* **2022**, *10*, 113808–113826. [CrossRef]
23. Korrai, P.; Lagunas, E.; Sharma, S.K.; Chatzinotas, S.; Bandi, A.; Ottersten, B. A RAN resource slicing mechanism for multiplexing of eMBB and URLLC services in OFDMA based 5G wireless networks. *IEEE Access* **2020**, *8*, 45674–45688.
24. Azimi, Y.; Yousefi, S.; Kalbkhani, H.; Kunz, T. Applications of Machine Learning in Resource Management for RAN-Slicing in 5G and Beyond Networks: A Survey. *IEEE Access* **2022**, *10*, 106581–106612. [CrossRef]
25. Polese, M.; Bonati, L.; D'oro, S.; Basagni, S.; Melodia, T. Understanding O-RAN: Architecture, interfaces, algorithms, security, and research challenges. *IEEE Commun. Surv. Tutorials* **2023**, *25*, 1376–1411.
26. Hung, C.-F.; Chen, Y.-R.; Tseng, C.-H.; Cheng, S.-M. Security threats to xApps access control and E2 interface in O-RAN. *IEEE Open J. Commun. Soc.* **2024**, *5*, 1197–1203.
27. Song, J.; Kovács, I.Z.; Butt, M.; Steiner, J.; Pedersen, K.I. Intra-RAN Online Distributed Reinforcement Learning For Uplink Power Control in 5G Cellular Networks. In Proceedings of the 2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring), Helsinki, Finland, 19–22 June 2022; pp. 1–7. [CrossRef]
28. Yungaicela-Naula, N.M.; Sharma, V.; Scott-Hayward, S. Misconfiguration in O-RAN: Analysis of the impact of AI/ML. *Comput. Netw.* **2024**, *247*, 110455.

29. Azari, M.M.; Solanki, S.; Chatzinotas, S.; Kodheli, O.; Sallouha, H.; Colpaert, A.; Montoya, J.F.M.; Pollin, S.; Haqiqatnejad, A.; Mostaani, A.; et al. Evolution of Non-Terrestrial Networks From 5G to 6G: A Survey. *IEEE Commun. Surv. Tutorials* **2022**, *24*, 2633–2672. [CrossRef]
30. Kumar, S. Exp_fric, [GitLab Repository]. Available online: https://gitlab.surrey.ac.uk/sk4sunil/exp_fric (accessed on 22 February 2025).
31. Xu, H.; Sun, X.; Yang, H.H.; Guo, Z.; Liu, P.; Quek, T.Q.S. Fair Coexistence in Unlicensed Band for Next Generation Multiple Access: The Art of Learning. In Proceedings of the ICC 2022–IEEE International Conference on Communications, Seoul, Republic of Korea, 16–20 May 2022; pp. 2132–2137. [CrossRef]
32. Zhang, C.; Nguyen, K.K.; Cheriet, M. Joint Routing and Packet Scheduling For URLLC and eMBB traffic in 5G ORAN. In Proceedings of the ICC 2022—IEEE International Conference on Communications, Seoul, Republic of Korea, 16–20 May 2022; pp. 1900–1905. [CrossRef]
33. Ramezanpour, K.; Jagannath, J. Intelligent zero trust architecture for 5G/6G networks: Principles, challenges, and the role of machine learning in the context of O-RAN. *Comput. Netw.* **2022**, *217*, 109358. [CrossRef]
34. Sedar, R.; Kalalas, C.; Alonso-Zarate, J.; Vázquez-Gallego, F. Multi-domain Denial-of-Service Attacks in Internet-of-Vehicles: Vulnerability Insights and Detection Performance. In Proceedings of the 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), Milan, Italy, 27 June–1 July 2022; pp. 438–443. [CrossRef]
35. Stafford, V. Zero trust architecture. *NIST Spec. Publ.* **2020**, *800*, 207.
36. Sen, N. Intelligent Admission and Placement of ORAN Slices Using Deep Reinforcement Learning. In Proceedings of the 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), Milan, Italy, 27 June–1 July 2022; pp. 307–311. [CrossRef]
37. Pham, C.; Fami, F.; Nguyen, K.K.; Cheriet, M. When RAN Intelligent Controller in ORAN Meets Multi-UAV Enable Wireless Network. *IEEE Trans. Cloud Comput.* **2022**, *11*, 2245–2259. [CrossRef]
38. Liang, X.; Shetty, S.; Tosh, D.; Kamhoua, C.; Kwiat, K.; Njilla, L. ProvChain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In Proceedings of the 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 14–17 May 2017; pp. 468–477.
39. Abedin, S.F.; Mahmood, A.; Tran, N.H.; Han, Z.; Gidlund, M. Elastic ORAN Slicing for Industrial Monitoring and Control: A Distributed Matching Game and Deep Reinforcement Learning Approach. *IEEE Trans. Veh. Technol.* **2022**, *71*, 10808–10822. [CrossRef]
40. Polese, M.; Bonati, L.; D'Oro, S.; Basagni, S.; Melodia, T. ColORAN: Developing Machine Learning-based xApps for Open RAN Closed-loop Control on Programmable Experimental Platforms. *IEEE Trans. Mob. Comput.* **2022**, *22*, 5787–5800. [CrossRef]
41. Peng, Z.; Zhang, Z.; Kong, L.; Pan, C.; Li, L.; Wang, J. Deep Reinforcement Learning for RIS-Aided Multiuser Full-Duplex Secure Communications with Hardware Impairments. *IEEE Internet Things J.* **2022**, *9*, 21121–21135. [CrossRef]
42. Jordan, E. *The Ultimate Guide to Open RAN: Open RAN Intelligent Controller (RIC)—Part 1*; The Fast Mode: Petaling Jaya, Malaysia, 2024.
43. Tran, T.D.; Nguyen, K.-K.; Cheriet, M. Joint Route Selection and Content Caching in ORAN Architecture. In Proceedings of the 2022 IEEE Wireless Communications and Networking Conference (WCNC), Austin, TX, USA, 10–13 April 2022; pp. 2250–2255. [CrossRef]
44. Huang, J.; Yang, Y.; Gao, Z.; He, D.; Ng, D.W.K. Dynamic Spectrum Access for D2D-Enabled Internet of Things: A Deep Reinforcement Learning Approach. *IEEE Internet Things J.* **2022**, *9*, 17793–17807. [CrossRef]
45. Richard, J. *Open RAN—Understanding the Architecture and Deployment*; HackMD: Hsinchu, Taiwan, 2021. Available online: https://hackmd.io/@jonathanrichard/HJp1cvYeO (accessed on 1 December 2024).
46. Faisal, K.M.; Choi, W. Machine Learning Approaches for Reconfigurable Intelligent Surfaces: A Survey. *IEEE Access* **2022**, *10*, 27343–27367. [CrossRef]
47. Polese, M.; Bonati, L.; D'Oro, S.; Basagni, S.; Melodia, T. Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges. *arXiv* **2022**, arXiv:2202.01032.
48. Alwarafy, A.; Abdallah, M.; Çiftler, B.S.; Al-Fuqaha, A.; Hamdi, M. The Frontiers of Deep Reinforcement Learning for Resource Management in Future Wireless HetNets: Techniques, Challenges, and Research Directions. *IEEE Open J. Commun. Soc.* **2022**, *3*, 322–365. [CrossRef]
49. Abdalla, A.S.; Upadhyaya, P.S.; Shah, V.K.; Marojevic, V. Toward Next Generation Open Radio Access Networks: What O-RAN Can and Cannot Do! *IEEE Netw.* **2022**, *36*, 206–213. [CrossRef]
50. Kim, E.; Choi, H.-H.; Kim, H.; Na, J.; Lee, H. Optimal Resource Allocation Considering Non-Uniform Spatial Traffic Distribution in Ultra-Dense Networks: A Multi-Agent Reinforcement Learning Approach. *IEEE Access* **2022**, *10*, 20455–20464. [CrossRef]
51. Nomikos, N.; Zoupanos, S.; Charalambous, T.; Krikidis, I. A Survey on Reinforcement Learning-Aided Caching in Heterogeneous Mobile Edge Networks. *IEEE Access* **2022**, *10*, 4380–4413. [CrossRef]
52. Brik, B.; Boutiba, K.; Ksentini, A. Deep Learning for B5G Open Radio Access Network: Evolution, Survey, Case Studies, and Challenges. *IEEE Open J. Commun. Soc.* **2022**, *3*, 228–250.

53. Cao, Y.; Lien, S.-Y.; Liang, Y.-C.; Chen, K.-C.; Shen, X. User Access Control in Open Radio Access Networks: A Federated Deep Reinforcement Learning Approach. *IEEE Trans. Wirel. Commun.* **2022**, *21*, 3721–3736. [CrossRef]
54. Addad, R.A.; Dutra, D.L.C.; Taleb, T.; Flinck, H. AI-Based Network-Aware Service Function Chain Migration in 5G and Beyond Networks. *IEEE Trans. Netw. Serv. Manag.* **2022**, *19*, 472–484. [CrossRef]
55. Peng, M.; Sun, Y.; Li, X.; Mao, Z.; Wang, C. Recent Advances in Cloud Radio Access Networks: System Architectures, Key Techniques, and Open Issues. *IEEE Commun. Surv. Tutorials* **2016**, *18*, 2282–2308. [CrossRef]
56. Ayala-Romero, J.A.; Garcia-Saavedra, A.; Gramaglia, M.; Costa-Pérez, X.; Banchs, A.; Alcaraz, J.J. vrAIn: Deep Learning Based Orchestration for Computing and Radio Resources in vRANs. *IEEE Trans. Mob. Comput.* **2022**, *21*, 2652–2670. [CrossRef]
57. Orhan, O.; Swamy, V.N.; Tetzlaff, T.; Nassar, M.; Nikopour, H.; Talwar, S. Connection Management xAPP for ORAN RIC: A Graph Neural Network and Reinforcement Learning Approach. In Proceedings of the 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA), Pasadena, CA, USA, 13–16 December 2021; pp. 936–941. [CrossRef]
58. Open Networking Foundation. Software-Defined Networking (SDN) Definition. Available online: https://opennetworking.org/sdn-definition/ (accessed on 21 December 2022).
59. Lee, H.; Jang, Y.; Song, J.; Yeon, H. ORAN AI/ML Workflow Implementation of Personalized Network Optimization via Reinforcement Learning. In Proceedings of the 021 IEEE Globecom Workshops (GC Wkshps), Madrid, Spain, 7–11 December 2021; pp. 1–6. [CrossRef]
60. Bauer, A.; Lesch, V.; Versluis, L.; Ilyushkin, A.; Herbst, N.; Kounev, S. Chamulteon: Coordinated Auto-Scaling of Micro-Services. In Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), Dallas, TX, USA, 7–10 July 2019.
61. Mao, M.; Humphrey, M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), Seattle, WA, USA, 12–18 November 2011; Article No. 49, pp. 1–12. [CrossRef]
62. Arabnejad, V.; Bubendorfer, K.; Ng, B. Dynamic multi-workflow scheduling: A deadline and cost-aware approach for commercial clouds. *Future Gener. Comput. Syst.* **2019**, *100*, 98–108. [CrossRef]
63. Shahin, A.A. Automatic Cloud Resource Scaling Algorithm based on Long Short-Term Memory Recurrent Neural Network. *Int. J. Adv. Comput. Sci. Appl. (IJACSA)* **2016**, *7*, 201–207. [CrossRef]
64. Das, S.; Li, F.; Narasayya, V.R.; König, A.C. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In Proceedings of the SIGMOD/PODS'16: International Conference on Management of Data, San Francisco, CA, USA, 26 June–1 July 2016; Microsoft Research: Redmond, WA, USA, 2016.
65. Rimedolabs. O-RAN Near Real-Time RIC. Available online: https://rimedolabs.com/blog/o-ran-near-real-time-ric/ (accessed on 24 January 2025).
66. Golkarifard, M.; Jin, Y. Dynamic VNF Placement, Resource Allocation and Traffic Engineering with Proactive Demand Prediction. *arXiv* **2021**, arXiv:2104.12345.
67. Chetty, S.B.; Nag, A.; Al-Tahmeesschi, A.; Wang, Q.; Canberk, B.; Marquez-Barja, J.; Ahmadi, H. Optimized Resource Allocation for Cloud-Native 6G Networks: Zero-Touch ML Models in Microservices-based VNF Deployments. *arXiv* **2024**, arXiv:2410.06938v1. [CrossRef]
68. Tran, N.P.; Delgado, O.; Jaumard, B. Proactive Service Assurance in 5G and B5G Networks: A Closed-Loop Algorithm for End-to-End Network Slicing. *arXiv* **2024**, arXiv:2404.01523v1.
69. D'Oro, S.; Bonati, L.; Polese, M.; Melodia, T. OrchestRAN: Network Automation through Orchestrated Intelligence in the Open RAN. *arXiv* **2022**, arXiv:2201.05632.
70. Lo Schiavo, L.; Garcia-Aviles, G.; Garcia-Saavedra, A.; Gramaglia, M.; Fiore, M.; Banchs, A.; Costa-Perez, X. CloudRIC: Open Radio Access Network (O-RAN) Virtualization with Shared Heterogeneous Computing. In Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, Washington, DC, USA, 18–22 November 2024; ACM MobiCom '24, pp. 558–572. [CrossRef]
71. Dai, J.; Li, L.; Safavinejad, R.; Mahboob, S.; Chen, H.; Ratnam, V.V. O-RAN-Enabled Intelligent Network Slicing to Meet Service-Level Agreement (SLA). *IEEE Trans. Mob. Comput.* **2024**, *24*, 890–906.
72. Chen, Y.; Wang, X. Deep Learning-Based Forecasting Model for Network Slice Scaling. *IEEE Trans. Netw. Serv. Manag.* **2022**, *18*, 456–468.
73. Patel, R.; Gupta, S. Markov Decision Process-Based Predictive Autoscaling in Cloud-Native Environments. *ACM Trans. Auton. Adapt. Syst.* **2021**, *16*, 23–38.
74. Zhang, L.; Kim, H. Reinforcement Learning for Real-Time Resource Scaling in ORAN. *Proc. IEEE INFOCOM* **2023**, 1015–1024.
75. Smith, B.; Lee, J. OpenStack Tacker: Policy-Based Scaling for VNFs. *IEEE Cloud Comput.* **2022**, *9*, 78–85.
76. Capone, A.; D'Elia, S.; Filippini, I.; Zangani, M. Measurement-based energy consumption profiling of mobile radio networks. In Proceedings of the 2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI), Turin, Italy, 16–18 September 2015; pp. 127–131. [CrossRef]

77. Bonati, L. Softwarized Approaches for the Open RAN of NextG Cellular Networks. Ph.D. Thesis, Northeastern University, Boston, MA, USA, 2022. [CrossRef]
78. Nguyen, H.; Brown, T. Unikernels for Lightweight Virtualization in Edge Computing. *ACM Comput. Surv.* **2023**, *55*, 1–27.
79. Lopez, D.; Martinez, F. Comparing Kubernetes with ETSI MANO for Network Function Orchestration. *Proc. IEEE NFV-SDN* **2022**, 223–230.