FlexScale: Scalable and Efficient Management Approach for Near-RT RIC in O-RAN

Sunil Kumar*, Rafik Zitouni†, Ayhan Akbas†, Chuan Heng Foh†
* School of Computing and Digital Media, London Metropolitan University, London, United Kingdom
† 5G/6GIC, Institute for Communication Systems, University of Surrey, Guildford, United Kingdom
Email: s.kumar@londonmet.ac.uk, {r.zitouni, a.akbas, c.foh}@surrey.ac.uk

***Abstract*—The Near-Real-Time (Near-RT) Radio Access Network (RAN) Intelligent Controller (RIC) in the Open RAN (O-RAN) architecture provides flexibility and programmability, enabling dynamic network management through Machine Learning based applications known as xApps. However, scalability and strict latency requirements hinder the support of numerous xApps. Existing orchestration solutions struggle to efficiently manage large-scale xApp deployments while maintaining latency below one second. Connecting multiple next-generation NodeBs (gNBs) to a single Near-RT RIC risks performance bottlenecks and single points of failure in the O-RAN architecture. To address these challenges, we propose FlexScale, a scalable approach to enhance O-RAN system capacity for extensive xApp deployments. It dynamically scales Near-RT RIC instances connected to gNB modules (E2 Nodes) using Kubernetes-based Horizontal and Vertical Pod Autoscaling (HPA, VPA) and native load balancing. FlexScale optimizes CPU utilization while meeting latency requirements. Simulations show that FlexScale efficiently scales xApps across multiple Near-RT RIC pods, addressing system limitations. Under high E2AP traffic, HPA and VPA autoscaling reduce latency by up to 96% compared to a single Near-RT RIC deployment. Additionally, CPU usage is reduced by approximately 70%, ensuring balanced resource utilization during traffic fluctuations. FlexScale demonstrates its capability to support large-scale xApp deployments while maintaining performance and efficiency.**

***Index Terms*—Scalability, Resource Utilisation, Load Balancing, Near-RT RIC, O-RAN, 6G networks, Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), E2AP interface.**

I. Introduction

Next-generation 6G networks will revolutionize telecommunications by overcoming issues like bandwidth, latency, and energy efficiency. Moving from the Internet of Everything to Intelligent Networks, 6G aims to support ML-driven applications linking trillions of devices. These advancements will enable real-time applications such as autonomous systems, smart grids, remote sensing, extended reality, drones, robotics, and telemedicine, all requiring low latency, scalability, and intelligent automation [1].

The Open Radio Access Network (O-RAN) paradigm emerges as a pivotal framework for achieving 6G goals. Leveraging virtualized and programmable technologies like Software-Defined Networking (SDN) and Network Function Virtualization (NFV), O-RAN enhances flexibility, scalability, and energy efficiency while enabling cloud-based architectures for Near-RT. Through ML integration, O-RAN supports crucial functionalities, including spectrum optimization, mobility monitoring, conflict mediation, and anomaly detection [2]. Despite these advancements, challenges persist. Current designs relying on Stream Control Transmission Protocol (SCTP) and E2 Application Protocol (E2AP) face scalability limitations [3]. Furthermore, some ML-based solutions fail to meet latency and performance guarantees for control loops in Near-RT and Non-RT RICs. As user equipment (UE) connections and traffic volumes increase, a single Near-RT RIC can become a bottleneck, limiting scalability and system efficiency. Addressing this issue through the addition of multiple Near-RT RICs which was not specified in the O-RAN alliance specifications [4] allows better distribution of processing loads, ensuring scalability and higher system performance. O-RAN also introduces complexities due to the dynamic traffic generated by multivendor xApps designed to monitor, report, and control RAN functions. The Near-RT RIC, central to these operations, often suffers from capacity limitations, causing performance degradation, connection failures, and extended downtimes under heavy traffic loads. Deploying Near-RT RIC instances across multiple machines provides partial relief but proves inefficient during low traffic and inadequate during peak traffic scenarios.

Cloud-native load balancer like LoxiLB [5] could address these gaps. However, the overall success of the deployment using LoxiLB depends on adapted configuration to the complex O-RAN architecture. This work proposes FlexScale, a novel framework with O-RAN-specific features for dynamic Near-RT RIC management. FlexScale dynamically scales Near-RT RIC instances and xApps according to traffic load, optimizing resource utilization and ensuring system stability. It incorporates a native load-balancing mechanism based on routing concepts to handle real-time SCTP connections and maintain latency below 1 second, essential for effective Near-RT RIC operations. Key Contributions of the paper are:

1) Scalable Approach for O-RAN: FlexScale's Auto-scaling approach is designed to support large number of xApps, Near-RT RIC instances and CU/DU/gNB. The approach ensures latency guarantees while optimizing the CPU resource. It integrates advanced monitoring tools like Grafana [6] and Prometheus [7] to track Kubernetes metrics, identifying bottlenecks and performance spikes.
2) Comparison with existent solution: FlexScale significantly outperforms a static deployment of FlexRIC [8], which lacks scalability and resource efficiency. Tests reveal that FlexScale achieves CPU utilization of up to 98%, ensuring efficient resource management and better scalability.

II. FlexScale Approach

The proposed FlexScale approach addresses both Near-RT RIC interfaces: the Northbound interface, connecting to xApps, and the Southbound interface, connecting to gNBs (or E2 Nodes).

A. Architectural Design

The architecture of the FlexScale is illustrated in Fig. 1, showing how it integrates seamlessly with Near-RT RIC. It works by collecting requests from the Service Management and Orchestration (SMO) system through a control interface. This interface allows network operators to submit their requests for ML-driven O-RAN applications. These requests are then gathered by the SMO system, and every T seconds (where T is a configurable interval). FlexScale Northbound is implemented on Kubernetes. This allows flexible and scalable orchestration of the O-RAN components. Each Kubernetes pod acts as a Near-RT RIC container that runs key FlexScale workloads.This allows it to dynamically scale near-RT RIC pods horizontally and vertically within the node. This scaling is based on the combined metrics of CPU utilization and latency. These metrics are continuously monitored to ensure efficiency and maintain the performance needed for real-time applications like xApps [4]. FlexScale Northbound load balancer proves how Near-RT RIC pods can be dynamically orchestrated and scaled up or down within a Kubernetes environment. This dynamic approach ensures efficient resource utilization and real-time performance, making it ideal for deploying large number of xApps. The detailed view of process is given in Fig. 2.

The SMO layer processes xApp deployment requests and verifies availability through the xApp catalog. Once the descriptors are defined, FlexScale calculates metrics using APIs from the Grafana dashboard, which gathers data from Kubernetes [9]. FlexScale retrieves the requested xApp, assigns it to an available node, and ensures latency constraints and resource availability are met. Load balancing is employed to distribute the workload across Near-RT RIC pods, while

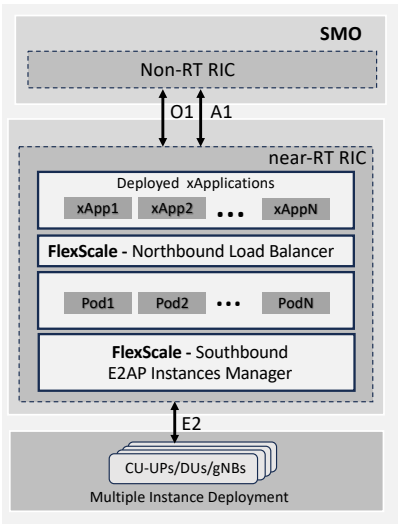

Fig. 1. The proposed FlexScale architecture

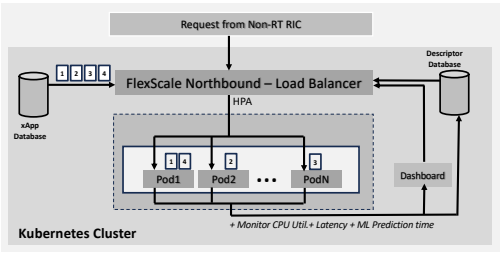

Fig. 2. Components of FlexScale Kubernetes Cluster

FlexScale dynamically adjusts the number of pods based on real-time metrics, such as latency and CPU usage, to maintain system performance and resource efficiency.

Kubernetes automates the deployment, scaling, and management of Near-RT RIC components, with each component deployed as a container. The cluster includes a single control plane and nodes capable of hosting multiple containers. A native load balancer dynamically allocates traffic across nodes, ensuring resilience and operational efficiency. Deployment configuration files specify the number of pod replicas for high availability, resource requests and limits for CPU and memory optimization, and load balancer settings for smooth traffic distribution. The Horizontal Pod Autoscaler (HPA) manages dynamic scaling by adjusting the number of replicas (1–3) based on CPU utilization, targeting 80%. This elasticity enables the system to handle changing workloads without manual intervention.

xApps are deployed in containerized formats with application descriptors that detail their functionality, such as RAN slicing or traffic steering, the ML models they utilize, and the

structure of input and output data. Descriptors also specify key performance indicators (KPIs), data formats, types of actions performed, and their respective outputs. Additionally, performance profiles are included, providing metrics such as latency and resource requirements like CPU and memory usage. These descriptors offer a comprehensive operational profile of each application, further elaborated in the next section.

B. Latency Modeling

To meet the required time constraints, we first developed 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. The primary objective of our design is to develop a model for total processing time that supports scaling intelligent O-RAN applications effectively. To achieve this, we focus on analyzing two key factors: latency and inference time. These factors remain consistent irrespective of whether the ML model is pretrained, as the number of operations (such as multiplications, convolutions, or additions) in the ML models is fixed. For evaluation, we use a single node within the kubernetes cluster, deploying one xApp instance at a time. To collect extensive data, we implemented an E2 traffic emulator using an open-source O-RAN dataset [10]. This generator simulates E2 traffic by continuously extracting random KPIs from the dataset in a format tailored to the input requirements of the xApp ML models. Each application descriptor specifies the necessary KPIs and input format for its corresponding xApp model. When a new xApp instance a is deployed on a server s, the traffic generator provides the required input data, enabling us to measure three types of latency:

- *Queuing time* ($t_{a,s}^{queue}$): The time taken for the xApp to process input after it reaches the E2 termination point on the Near-RT RIC.
- *Execution time* ($t_{a,s}^{exec}$): The time required to generate output once the input is processed.
- *ML inference time* ($t_{a,s}^{inf} = t_{a,s}^{queue} + t_{a,s}^{exec}$): The total time needed to process input and produce output.

In addition to monitoring these latencies, we track CPU and RAM utilization on the server throughout the process. While the time required for transferring KPIs and control actions between the RAN and RICs could be included, these values remain constant since all servers are part of the same cluster. Based on these assumptions, we define the inference time (Eqn. 1) when y instances of application $a \in A$ are running on servers as:

$$t_{a,s}^{inf}(y) = t_{a,s}^{exec}(y) + t_{a,s}^{queue}(y), \tag{1}$$

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

III. Analytical Model

We propose a queueing and control theory model to mathematically validate scaling effectiveness in time sensitive applications through latency, inference time, and CPU usage.

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

The system can be 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-RT RIC pods (representing horizontal scaling). The average waiting time (Eqn. 2) in the system can be represented by:

$$W_q = \frac{\lambda}{\mu(c\mu - \lambda)} \tag{2}$$

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:

$$W_q \leq L_{max} \tag{3}$$

B. Control Theory (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 $L_{observed}(t)$:

$$e(t) = L_{max} - L_{observed}(t) \tag{4}$$

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

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{5}$$

Here, K_p is the proportional gain and controls the magnitude of the response proportional to the current error $e(t)$. K_i represent integral gain, it addresses the accumulated error over time by considering the integral of $e(t)$. K_d is the derivative gain, predicts the future behavior of $e(t)$ by considering the rate of change of the error $\frac{de(t)}{dt}$. τ is a variable of integration used to compute the integral term. This feedback mechanism adjusts resources (pods) in real-time, keeping the latency within acceptable limits.

C. 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. If $CPU_{observed} \geq 0.8 \times CPU_{max}$ scale-up by adding a new pod.

- *Scale-Down Condition*: If CPU utilization stays idle (below a threshold) for 180 seconds, a pod is removed, but at least one pod must always remain active. If $CPU_{observed} \leq 0.1 \times CPU_{max}$ for 180 seconds, scale-down by removing a pod. 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.

D. Combined Metrics

Metrics are collected from individual pods and aggregated into system-level metrics (e.g., average CPU usage, maximum latency). These aggregated metrics are then used to calculate a combined scaling metric C, which represents the overall system performance. C integrates CPU utilization, latency, and inference time using weighted parameters. When C exceeds a predefined threshold, scaling decisions are triggered.

$$C = \alpha \cdot \frac{CPU_{observed}}{CPU_{max}} + \beta \cdot \frac{L_{observed}}{L_{max}} + \gamma \cdot \frac{T_{inf}}{T_{max}} \quad (6)$$

Where α, β, and γ are weights assigned to CPU, latency, and inference time, respectively. T_{max} is the maximum acceptable inference time. This combined metric reflects the overall system's resource utilization and performance. When C exceeds a predefined threshold, the system triggers scaling events, either to scale up or down to meet performance targets. This approach ensures a balance between optimal resource utilization and meeting the stringent latency requirements of time-sensitive applications. Scaling up occurs when $C > C_{\text{threshold}}$, and scaling down occurs when C is below a certain lower bound, ensuring system scalability while preventing resource contention. This approach is applied to both horizontal scaling (by adding more Near-RT RIC pods) and vertical scaling (by adjusting resources allocated to existing pods).

IV. Performance Evaluation

The performance evaluations of the northbound and southbound are discussed separately.

A. FlexScale Northbound

We modernize the Near-RT RIC using containerization, orchestration, and load balancer configurations to improve scalability, resource management, and deployment efficiency. By packaging the Near-RT RIC as a Docker container, each instance is uniquely identified for easy tracking and management. Kubernetes is used as the orchestration platform, deploying multiple container instances with unique IP addresses to streamline communication and control. A load balancer, positioned northbound of the containers, dynamically distributes incoming xApp connections evenly across instances, reducing latency and maintaining efficiency. xApps connect via a predefined IP address and port, with the load balancer routing requests to available containers. Kubernetes' orchestration allows dynamic scaling of Near-RT RIC instances based on demand, ensuring optimal resource utilization and performance. This scalable solution supports high traffic volumes, balanced load distribution, and seamless horizontal scaling, adapting to varying loads while maximizing efficiency. Initial efforts focused on containerizing the Near-RT RIC to enable flexible experimentation using Docker and Kubernetes. Components were encapsulated to run in isolated environments, improving scalability and manageability. The next phase implemented horizontal scaling, enabling additional instances to handle increased load. A load balancer was integrated to distribute traffic evenly. Lastly, a Grafana-powered dashboard visualizer was developed to monitor system performance, providing real-time insights into resource utilization, scaling, and load balancing efficiency.

This paper outlines horizontal scaling in an experimental setup using the HPA for a Near-RT RIC system. Initially, the system runs one pod with a CPU utilization of 3%, monitored on Kubernetes dashboard. The HPA is set to scale between 1 and 5 pods based on CPU usage, monitored through a custom dashboard tracking CPU and memory for each pod. As xApps increase, CPU usage rises to 96%, prompting the HPA to add a pod to balance the load. The Kubernetes dashboard confirms this by showing CPU utilization drop to 46%, demonstrating effective scaling without performance loss. The dashboard also provides insights into KPIs, allowing for detailed resource management. After terminating 4 xApps, the load decreases to 31%, and with one more termination, it drops to 15%, illustrating the system's ability to adjust resources based on demand. When idle for 3 minutes, the system scales down from 2 pods to 1, as seen on the Kubernetes dashboard, thus preventing resource waste. Following this, CPU utilization on the remaining pod rises from 15% to 30%, highlighting dynamic resource allocation while maintaining efficiency.

B. FlexScale Southbound

Our Near-RT RIC E2AP Instances Manager simulates the scenario of the interconnection of one or multiple Near-RT RICs to E2 Nodes (see Fig. 3). The process of testing and performance evaluation begins with traffic generation, where the generator method creates different patterns of E2AP (SCTP) traffic of Indication messages. The traffic between the xApps via Near-RT RIC and E2 Nodes might be periodic or random. The latter is qualified as realistic because we are considering short-term fluctuations and random bursts. This traffic is distributed among the available pods, each calculating SCTP of E2AP interface latency for incoming messages considering

TABLE I
PARAMETERS FOR POD CONFIGURATION, AUTOSCALING SIMULATOR AND TRAFFIC CHARACTERISTICS

Category	Parameter	Value
Pod Configuration	Base Capacity	200 packets/sec
	Max CPU Allocation	4 Cores
	Min CPU Allocation	0.5 Core
	SCTP Base Latency	$50\mu s$
	Network Latency	10 msec
	Heartbeat Interval	1 sec
	Max RTT	200 msec
Autoscaling Simulator	Max. Pods	5
	Scaling Strategies Tested	Single Pod (None) HPA VPA
Traffic Characteristics	Base Rate	1000 packets/sec
	Traffic Variability	Periodic Patterns (10ms, 100ms, 1000ms) with random burst (realistic)

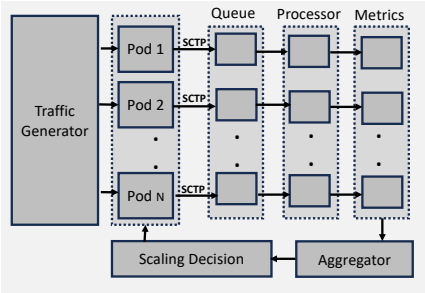

Fig. 3. Architecture of Near-RT RIC E2AP Instances Manager

the network congestion and transmission delays. Messages are added to the pods' queues, with queue latency determined by the current load. Pods process these messages from their queues, calculating processing latency based on allocated CPU resources and workload. After processing, each pod generates a PodMetrics object containing performance data. The Instances Manager uses these metrics to make scaling decisions, either horizontally by adding pods or vertically by adjusting CPU allocations. Finally, metrics from all pods are aggregated for analysis, providing insights into system performance and scaling efficiency. Table I shows the configuration of our O-RAN FlexScale E2AP Instances Manager with realistic configuration parameters for CPU resources, protocol parameters, and traffic patterns.

V. RESULTS AND DISCUSSION

In our experiments, we evaluated the performance of the FlexScale approach and compared it with the O-RAN compliant FlexRIC using the same hardware configuration. The experimental setup replicated the conditions discussed earlier, employing different types of xApps specifically designed to measure performance. The experimental setup demonstrates effective scalability of the pods in response to varying loads,

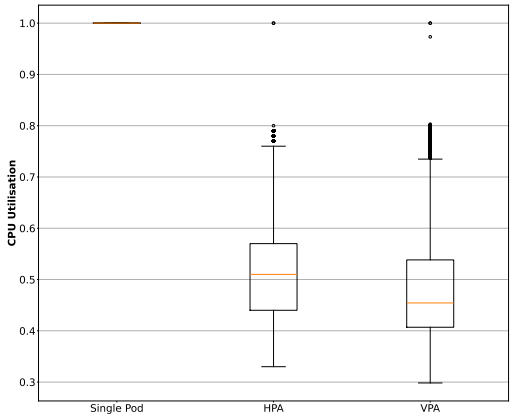

Fig. 4. Comparison of CPU utilisation in the three cases with single and multiple instances of Near-RT RIC

thereby optimizing resource utilization. As the system adapts to the demands of the xApps, the FlexScale ensures that the appropriate number of pods is active, maintaining performance efficiency. Additionally, the visualization dashboard provides comprehensive KPI metrics, facilitating improved management and monitoring capabilities. This integrated approach highlights the system's robustness in handling traffic fluctuations while ensuring optimal resource allocation and performance management.

Fig. 4 shows a statistical overview of CPU utilization across various Near-RT RIC deployment strategies (Single Pod, HPA, VPA) and traffic patterns, illustrating metrics like median, quartiles, and outliers. This assessment would reveal resource usage variability and identifies potential over-utilization issues. With a single Pod deployment, CPU utilization is concentrated at the maximum level (near 1.0), indicating over-utilization and resource bottlenecks. The CPU utilization distribution for HPA is more balanced, with a median around 50%. However, there are noticeable outliers above the upper whisker, indicating occasional spikes in CPU utilization. This suggests that while HPA mitigates high utilization, it may not scale quickly enough to handle sudden traffic bursts. VPA shows even lower CPU utilization compared to HPA, with the median around 40%. However, the wider interquartile range and higher number of outliers at the upper end indicate variability in how efficiently VPA adjusts resources. The lower median could imply under-utilization of resources, potentially leading to inefficiencies.

Fig. 5 shows the percentage of CPU utilisation under different load patterns (10ms, 100ms, 1s, realistic) for three scaling strategies. As expected *Single Pod* strategy shows the highest average utilization (52.8%) compared to HPA (7.9%) and VPA (31.6%), suggesting that it may be less effective at scaling under varied traffic loads. HPA shows the lowest CPU utilization, representative of its ability to dynamically

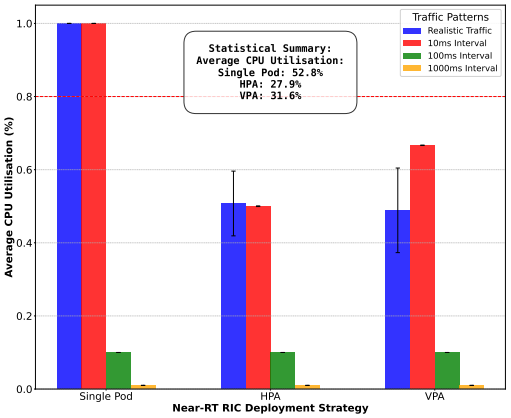

Fig. 5. Statistics of CPU utilisation under different traffic patterns

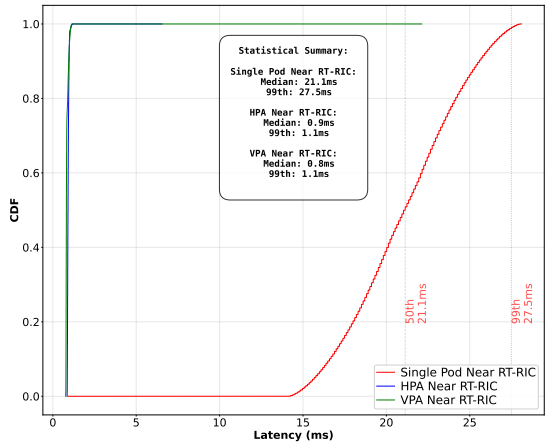

Fig. 6. CDF of latencies of indication messages under high xApp request frequencies under realistic traffic

adjust resources based on real-time demand, making it ideal for fluctuating traffic. VPA also exhibits moderate resource utilization, balancing performance and efficiency, yet it is not as responsive as HPA. HPA appears to have the most consistent utilization, which is a positive indicator for reliability. These findings suggest that, for environments with unpredictable loads, HPA may offer the best scalability and efficiency, while the Single Pod strategy could be sufficient for more stable traffic with lower resource demands.

Cumulative Distribution Function (CDF) of latencies for indication messages under realistic traffic load is shown in Fig. 6. The CDF curves illustrate how latency varies across the three deployment strategies, allowing for a comparison of responsiveness under the same traffic conditions. If a single Near-RT RIC is deployed, a significantly higher median latency of 21.1 ms can be experienced. The 95th percentile latency is 26.5 ms, and the 99th percentile is 27.5 ms, indicating that 95% of the requests are processed within this time, and 1% experience even longer delays. The steep rise in the CDF reflects that most of the requests experience higher latencies, suggesting it may struggle under heavier loads or less effective resource management compared to the other strategies. However, Both HPA and VPA show remarkably lower latencies, with medians of 0.9 ms and 0.8 ms, respectively. The 95th and 99th percentile latencies for these strategies are also low (1.0 ms and 1.1 ms), indicating that nearly all requests are handled very quickly, signifying efficient performance under realistic traffic loads. The curves for HPA and VPA are nearly overlapping, demonstrating similar performance characteristics and suggesting both are highly effective at managing low-latency requirements.

Our findings demonstrate that FlexScale supports nearly twice the number of xApps compared to the traditional approach while maintaining efficient resource utilization. The evaluation included two categories of xApps: highly resource-intensive and less resource-intensive. Highly Resource-Intensive xApps are designed to perform complex processing tasks that consume significant CPU, memory, and GPU resources. To test, we developed a computationally intensive rate adaptation xApp, which controls real-time RAN functions by predicting the Modulation and Coding Scheme (MCS) using an ML-based microservice. This xApp consumes more CPU/GPU resources and requires longer inference times. It is a resource-heavy control task that processes high-frequency data streams. On the other hand, Less Resource-Intensive xApps have minimal computational demands and typically perform lightweight tasks, such as periodic data reporting and monitoring. The number of highly resource-intensive xApps increased from 17 to 35 (which is an increase of approximately 105.88%), while the less resource-intensive xApps increased significantly from 38 to 89 (134.21%).

VI. Conclusion and Future Directions

This paper presented FlexScale, a dynamic scaling solution tailored for Near-RT RIC in O-RAN. It is designed to enforce CPU usage and time constraints while supporting a large number of xApps and E2 Nodes. We developed a latency model based on a measurement campaign conducted on a Kubernetes cluster, complemented by an analytical model and a prototype that aligns with O-RAN specifications. Through a series of comparisons with traditional FlexRIC, we demonstrated that HPA is capable of effectively deploying xApps while meeting strict latency requirements. Specifically, FlexScale supports approximately double the number of xApps compared to traditional FlexRIC. Furthermore, we proposed a Southbound E2AP Instances Manager to allow E2 Nodes to be connected to not only one Near-RT RIC but to multiple instances. We performed intensive simulations to test the efficiency of HPA

and VPA auto-scaling strategies based on CPU utilization. Under realistic and high-frequency traffic, we measured CPU utilization and latencies of E2AP Indication messages. HPA is found to be the most effective strategy due to its capability for horizontal scaling, making it ideal for dynamic and bursty traffic patterns. VPA also proves to be a strong contender, particularly in scenarios where vertical scaling is preferred or possible. Our future objective is to enhance our Southbound FlexScaler decision-maker by incorporating additional metrics like the state of the queue and the network performance.

ACKNOWLEDGEMENT

This work has been done within the framework of the CELTIC-NEXT under the project 6G-SMART supported by Innovate UK (Project No: 640240) and HiPer-RAN, supported by UK Department for Science, Innovation, and Technology. We also would like to acknowledge the support of the 5G/6GIC members for this work.

REFERENCES

[1] N. Luong, D. Hoang, S. Gong, and et al, "Applications of Deep Reinforcement Learning in Communications and Networking: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2664–2732, 2020.

[2] A. Garcia-Saavedra and X. Costa-Pérez, "ORAN: Disrupting the Virtualized RAN Ecosystem," *IEEE Communications Standards Magazine*, vol. 5, no. 4, pp. 96–103, 2021.

[3] M. Polese, L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "Understanding o-ran: Architecture, interfaces, algorithms, security, and research challenges," *IEEE Communications Surveys Tutorials*, vol. 25, no. 2, pp. 1376–1411, 2023.

[4] O-RAN Alliance, "O-RAN Alliance: O-RAN Architecture Specification," April 2024, o-RAN.WG3.E2SM-R003-v04.00. [Online]. Available: https://www.o-ran.org/specifications

[5] LoxiLB, "Loxilb: Cloud-native load balancer," 2023, accessed: [January 2025]. [Online]. Available: https://www.loxilb.io/

[6] Grafana Labs, "Grafana: Open-Source Analytics & Monitoring Solution," https://grafana.com, 2014, accessed: 2024-12-14.

[7] "Prometheus: Monitoring System & Time Series Database," https://prometheus.io, 2012, accessed: 2024-12-14.

[8] R. Schmidt, M. Irazabal, and N. Nikaein, "FlexRIC: An SDK for next-generation SD-RANs," in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 2021, pp. 411–425.

[9] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. ACM, 2015, p. 167.

[10] M. Polese, L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "Colo-ran: Developing machine learning-based xapps for open ran closed-loop control on programmable experimental platforms," *IEEE Transactions on Mobile Computing*, vol. 22, no. 10, pp. 5787–5800, 2022.