KARMA: Knowledge-Aware Resource Management and Autoscaling for Edge Workloads

Dimitrios Selis[§], Kostas Ramantas[†], Luis Alonso[§], John S Vardakas[†], Christos Verikoukis[*]
[§]Dept. of Signals Theory and Telecommunications, Universitat Politècnica de Catalunya, Barcelona, Spain
[†]Iquadrat Informatica, S.L, Barcelona, Spain
[*]Dept. of Computer Engineering and Informatics, University of Patras, Patras, Greece
Emails: {dimitrios.selis, luis.g.alonso}@upc.edu, {kramantas, jvardakas}@iquadrat.com, cveri@ceid.upatras.gr

***Abstract*—In the realm of modern telecommunication ecosystems, resource-aware management is critical in ensuring Quality of Service (QoS) while enabling proactive, zero-touch operations. Furthermore, recent advances in Software Defined Networking (SDN) have significantly contributed to beyond 5G (B5G) network optimization, offering innovative solutions to tackle their growing complexity. Moreover, the inherently non-convex nature of service provisioning challenges presents critical obstacles to delivering a seamless user experience. This paper introduces a novel framework that integrates Dueling Double Deep Q-Networks (Dueling DDQN) within a cloud-native edge infrastructure, enabling zero-touch service scaling. By harnessing the advanced capabilities of Deep Reinforcement Learning (DRL), the proposed method autonomously identifies and addresses scaling demands, thereby improving overall network capabilities. The effectiveness of our approach is validated through experiments conducted on a multi-node Kubernetes testbed, demonstrating its ability to alleviate performance bottlenecks in distributed environments.**

***Index Terms*—Autoscaling, Zero-touch, Deep Reinforcement Learning, Double Deep Q-Networks (DDQN), Dueling Architecture (DA), Kubernetes, 6G Mobile Communications**

I. Introduction

In the context of beyond 5G (B5G) networks, achieving elasticity and scalability is fundamental to satisfying the stringent service-level requirements. Elasticity refers to the system's ability to dynamically and efficiently allocate computational resources to ensure adherence to Service Level Agreements (SLAs). This capability is increasingly realized through the adoption of Cloud-Native Network Functions (CNFs). CNFs are managed through orchestration platforms such as Kubernetes [1], which provides a declarative object-based model to define and control infrastructure and application behavior, enabling enhanced resource management, fault recovery, high availability and overall network optimization.

The increasing need of zero-touch network and service management necessitates the adoption of state-of-the-art Artificial Intelligence (AI) techniques, particularly Deep Learning (DL). DL enables systems to respond intelligently to network events, in a manner that complements the reactive nature of Event-Driven Architectures (EDAs). EDAs enable systems to allocate resources dynamically based on the occurrence of specific events, ensuring sufficient resource provisioning across cloud and edge environments. Additionally, EDAs facilitate prioritized resource management by ensuring that essential workloads are processed ahead of less urgent ones, which is valuable in latency-sensitive applications, where timely execution is critical [2].

The paradigm shift to machine learning-based solutions in complex network environments is driven by recent advances in deep reinforcement learning (DRL). Through agent-based learning, DRL enables systems to iteratively refine decision-making processes based on historical interactions, thereby optimizing network performance within a continuous feedback loop. This integration fosters the emergence of robust, efficient and self-optimizing network infrastructures. Furthermore, advancements in AI have catalyzed the development of intelligent, autonomous networks that adapt to specific service requirements and user demands. These architectures enable efficient computational resource allocation, ensuring that service instances meet their performance objectives while mitigating the risk of network congestion.

Since conventional scaling mechanisms operate based on reactive heuristics or static thresholds, we propose an adaptive, zero-touch autoscaling framework that employs DRL and is designed to operate within edge environments. The autoscaling challenge is modeled as a Markov Decision Process (MDP), enabling learning-based, autonomous scaling decisions that align with future network paradigms, such as those expected in 6G systems. The key contributions of this work are summarized as follows:

- Zero-touch autoscaling framework: We integrate a Dueling DDQN agent for zero-touch management and orchestration, operating over a refined state space consisting of utilization metrics that are enriched with node-level resource distributions. This formulation enables the agent to learn optimal scaling policies by distinguishing between state-value and action advantages, improving convergence in high-dimensional, dynamic environments.
- Entropy-aware decision mechanism for stable scaling: To mitigate instability in autoscaling decisions, we introduce an entropy-based confidence filter. By analyzing the distribution of predicted actions and computing Shannon entropy [3], the agent can assess uncertainty and apply scaling operations only when confidence exceeds a threshold that adapts as learning progresses. This hybrid technique balances exploration with stability, avoiding erratic or over-aggressive scaling behavior.

- RL-driven autoscaling beyond native orchestration: The core contribution of our approach lies in replacing static heuristics with a learned scaling policy, which is context-aware from fine-grained system metrics, service-specific and robust to system variability, offering an adaptive and zero-touch orchestration layer suitable for multi-tenant and B5G edge environments.

The remainder of this paper is organized as follows. Section II reviews related work in complex distributed systems. The proposed methodology and system model are presented in Section III. Section IV provides a comprehensive analysis of our approach through performance experimentation and evaluation. Finally, concluding remarks and directions for future improvements to the framework are presented in Section V.

II. Related Work & Motivation

Balla et al. [4] introduce a hybrid autoscaler for Kubernetes, combining vertical and horizontal scaling in two phases. It estimates optimal CPU limits using regression over short-term performance data and then scales horizontally based on workload intensity thresholds. Zhou et al. [5] propose a predictive autoscaler that combines time-series forecasting and neural models to anticipate workload patterns and adjust scaling decisions accordingly, with a dynamic action space to improve responsiveness. Their approach enhances reaction time and reduces SLA violations under fluctuating traffic conditions. Unlike these approaches, our work adopts a model-free RL paradigm that learns directly from real-time system feedback without relying on workload prediction or handcrafted thresholds and can operate in uncertain, dynamic environments with improved adaptability, stability and scalability.

III. System Model and Proposed Methodology

The system model in Fig. 1 represents a multi-domain edge infrastructure managed by a centralized control plane. The infrastructure consists of multiple logical domains and can be formally represented as an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of domains and $\mathcal{E}$ represents inter-domain connectivity. Each domain is instantiated as a logical execution environment corresponding to a Kubernetes node hosting a collection of containerized microservices and is provisioned with computing resources. The Vertical Pod Autoscaler (VPA), which corresponds to a set of components that automatically adjust the amount of CPU and memory requested by service instances running in the cluster, is deployed in each domain so as to provide recommendations on optimal resource allocation based on historical usage. These recommendations are integrated into the system state, enabling informed scaling decisions. Furthermore, each domain's utilization state is represented by the vector $\mathbf{u}_v(t) = (u_{\text{CPU},v}(t),\ u_{\text{RAM},v}(t),\ u_{\text{CPU-VPA},v}(t),\ u_{\text{RAM-VPA},v}(t)) \in \mathcal{U} \subseteq \mathbb{R}^4$, where $u_{\text{CPU},v}(t)$ and $u_{\text{RAM},v}(t)$ denote the CPU and memory usage of domain v at time t and $u_{\text{CPU-VPA},v}(t)$ and $u_{\text{RAM-VPA},v}(t)$ are resource recommendations from the VPA.

Within the multi-domain network, an RL agent is deployed to make continuous decisions about allocating resources for each node. The agent observes the current state of each domain, represented as $\mathbf{u}_v(t)$ and determines whether to scale resources up, down, or maintain the current allocation. The RL-driven architecture enables intelligent, self-optimizing and zero-touch resource allocation across distributed edge environments. It is designed to meet the demands of future 6G infrastructures, which will require decentralized, scalable and autonomous resource orchestration mechanisms.

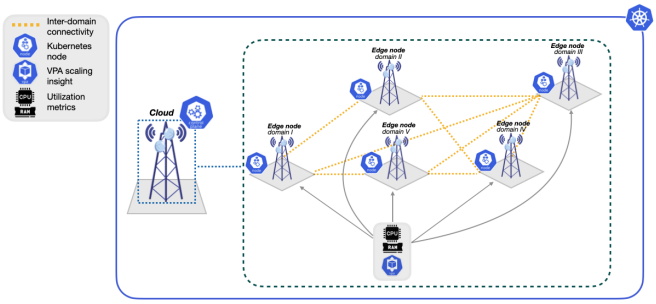
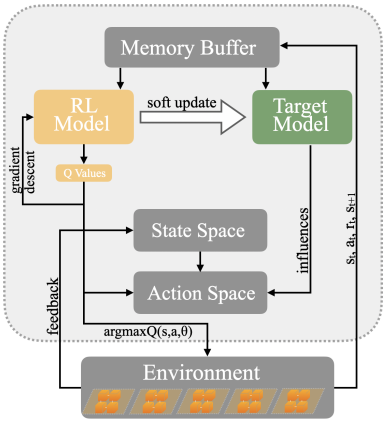


Fig. 1: Multi-domain Edge Infrastructure as an undirected graph $\mathcal{G}$

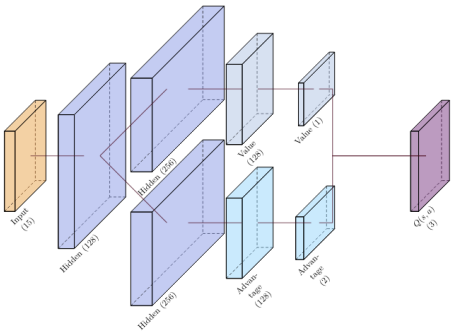
A. Architecture Blocks

As depicted in Fig. 2a, the *Memory Buffer* serves as a central component of the RL agent architecture, storing past experiences in the form of state transitions, actions, rewards and resulting states. These experiences are replayed during training, enabling the *RL Model* to learn from past interactions and gradually improve its policy by consolidating memory and accumulating knowledge over time.

The *RL Model* architecture, depicted in Fig. 2b, is based on the Dueling DDQN architecture. The agent input forms the current *state*, which is processed through a shared feature extraction network composed of three fully connected layers. This is followed by a split into two streams, the *Value Stream* that estimates the overall quality of the current state using two additional layers and the *Advantage Stream* that assesses the relative importance of each possible action. These two outputs are combined to produce the final *Q-values*, forming the Dueling DDQN architecture, which improves upon the traditional Deep Q-Network (DQN) [6] by structurally separating the estimation of state value from the advantage of individual actions, without requiring additional supervision. To select actions, the agent follows an epsilon-greedy strategy, balancing exploration with exploitation. Through a process of continuous improvement known as *Policy Update*, it employs the calculated Q-values to select actions and evolves its strategy for optimal decision-making. This involves adjusting the weights of the neural network based on the observed rewards and the predicted future rewards, aligning the policy more closely with optimal actions and collecting past experiences from the Memory Buffer. The *Target Model*, which is structurally identical to the main RL Model, is updated at a slower rate to serve as a stable reference during training and to provide reliable target Q-values that guide the learning



(a) Dueling DDQN Model



(b) Dueling DDQN Architecture

Fig. 2: Overview and architecture of the Dueling DDQN agent.

process of the RL Model. By decoupling action selection from target evaluation, the use of a separate target network prevents adverse learning behavior. After actions are executed in the *Environment*, the agent monitors the resulting changes in the system state. These observations are stored as new experiences in the Memory Buffer, a process that reflects the most recent behavior of the system. Together, the Memory Buffer, RL Model and Target Model form a cohesive RL framework capable of dynamic, autonomous resource management in distributed environments. This architecture not only enables proactive scaling decisions but also promotes adaptability to changes in workload or infrastructure conditions. From a practical viewpoint, this dual-stream design improves learning efficiency and policy robustness by assisting the model to discern between changes in state significance and action relevance. As a result, the agent is better equipped to generalize across diverse scenarios and make precise decisions under varying conditions [7].

B. Reinforcement Learning Agent Formulation

To effectively apply DRL in our context, we formulate the problem as a single-agent Markov Decision Process (MDP), defined by the tuple $\langle S, A, P, R, \gamma \rangle$, where $\mathcal{S}$ denotes the set of possible environment states, $\mathcal{A}$ represents the action space, $\mathcal{P}$ defines the state transition probabilities, $\mathcal{R}$ is the reward function guiding the agent's learning and γ is the discount factor determining the importance of future rewards. Based on this MDP framework, the foundational components of our DRL model are further detailed as follows:

a) State Space: The agent's state at time step t, denoted by $\mathbf{s}_t \in \mathbb{R}^{5+2N}$, is a continuous-valued vector capturing both microservice-level and cluster-level features, where N is the number of nodes in the Kubernetes cluster. Formally, the state is defined as:

$$\mathbf{s}_t = \Big[\underbrace{c_t, m_t, i_t, \hat{c}_t, \hat{m}_t}_{\text{microservice-level features}}, \underbrace{u_t^{(1)}, \ldots, u_t^{(N)}}_{\text{node-level CPU fractions}}, \underbrace{v_t^{(1)}, \ldots, v_t^{(N)}}_{\text{node-level memory fractions}} \Big], \tag{1}$$

where $c_t, m_t \in [0,1]$ are the normalized CPU and memory utilization of the target microservice, $i_t \in [0,1]$ denotes the current number of running instances normalized by the maximum allowable number of instances, $\hat{c}_t, \hat{m}_t \in [0,1]$ represent the CPU and memory recommendations from VPA, $u_t^{(i)} \in [0,1]$ and $v_t^{(i)} \in [0,1]$ denote the CPU and memory usage fractions for node i, respectively. This high-dimensional state representation enables the RL agent to make context-aware scaling decisions by accounting for both the localized behavior of the target microservice and the global resource utilization across the cluster.

b) Action Space: The agent operates in a discrete hybrid action space defined as $\mathcal{A} = \{-1, 0, 1, 2, \ldots, K\}$, where K denotes the maximum allowable number of replicas. The action $a_t \in \mathcal{A}$ selected at time step t is interpreted as follows:

- $a_t = -1$: decrease the current number of replicas by one.
- $a_t = 0$: take no action.
- $a_t = k$ for $k \in \{1, \ldots, K\}$: scale the target microservice to exactly k replicas.

This hybrid formulation enables both fine-grained reactive scaling and coarse-grained direct adjustments, providing the agent with the flexibility to learn context-aware policies across diverse workload dynamics.

c) Reward Function: The reward function is designed to balance efficiency, cost-awareness and stability in autoscaling decisions. At each time step t, the reward $r_t \in \mathbb{R}$ is computed based on the observed CPU and memory usage, the desired number of replicas and the target microservice's resource footprint across the cluster. The reward is defined as:

$$r_t = \underbrace{\alpha \cdot \frac{c_t + m_t}{2}}_{\text{efficiency gain}} - \underbrace{\beta \cdot \frac{i_t}{i_{\max}} \cdot \left(1 - \frac{c_t + m_t}{2}\right)}_{\text{over-provision penalty}} - \underbrace{\gamma \cdot \mathbb{I}(i_t \neq i_{t-1})}_{\text{scaling penalty}}, \tag{2}$$

where $c_t, m_t \in [0, 1]$ denote the normalized CPU and memory utilization at time t. The variable i_t is the current number of replicas and $i_{\max}$ is the maximum allowable number of replicas. The coefficients α, β, γ control the relative importance of efficiency, cost-awareness and stability, respectively. Lastly, $\mathbb{I}$ is an indicator function that evaluates to 1 if a scaling action occurs and 0 otherwise. The first term encourages high resource utilization, the second penalizes unnecessary resource usage when utilization is low and the third introduces a small cost for any scaling operation to reduce system churn and promote stability.

d) Transition Probability: Although the agent operates under a model-free RL framework, the environment implicitly defines a stochastic transition function $P(s_{t+1} \mid s_t, a_t)$, governed by the system's internal dynamics and autoscaling mechanisms. Here, $s_t \in \mathcal{S}$ denotes the system state at time t and $a_t \in \mathcal{A}$ represents the selected scaling action. The subsequent state s_{t+1} reflects the impact of the action on system behavior, including updated resource usage, VPA recommendations and node-level metrics. While the agent does not model this transition explicitly, it learns an optimal policy by interacting with the environment and observing these state transitions over time.

e) Discount Factor: The discount factor $\gamma \in [0, 1]$ governs the agent's preference for immediate versus future rewards. A value of $\gamma = 0.95$ is used to account for the delayed effects of autoscaling decisions in the environment.

C. Action Selection

The RL agent selects an action a_t based on the current state observation $\mathbf{s}_t$ and the learned policy, which estimates the action-value function $Q(\mathbf{s}, a; \theta)$. The selected action is determined as:

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(\mathbf{s}_t, a; \theta), \tag{3}$$

where $\mathbf{s}_t$ denotes the current system state, a_t is the chosen scaling action and θ are the parameters of the policy network.

To balance exploration and exploitation during training, the agent follows an ϵ-greedy policy. With probability ϵ, it selects a random action to explore alternative scaling strategies. With probability $1 - \epsilon$, it selects the action that maximizes the estimated Q-value. The exploration rate ϵ is initialized to a high value and decays gradually toward a predefined minimum, enabling extensive exploration in the early stages of training and promoting exploitation as learning converges.

This action selection mechanism enables the agent to adaptively determine optimal scaling decisions under dynamic workloads. By continuously refining its policy based on observed system states and rewards, the agent avoids premature convergence to suboptimal behaviors and improves its ability to generalize across different resource allocation scenarios.

$$a_t = \begin{cases} \text{random action from } \mathcal{A}, & \text{with probability } \epsilon, \\ \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(\mathbf{s}_t, a; \theta), & \text{with probability } 1 - \epsilon, \end{cases} \tag{4}$$

where a_t is the action taken at time t, $\mathcal{A}$ is the action space, ϵ is the exploration rate, $\mathbf{s}_t$ is the current state and $Q(\mathbf{s}_t, a; \theta)$ is the estimated action-value of taking action a in state $\mathbf{s}_t$, given parameters θ.

D. Streams

Dueling DDQN decomposes the Q-value function into a state-value function $V^\pi(s)$ and an advantage function $A^\pi(s, a)$ as follows:

$$V^\pi(s) = \mathbb{E}[R_t \mid s_t = s, \pi], \tag{5}$$

where R_t is the expected value of the reward from time t and $\mathbb{E}$ represents the expected value given the policy π. Thus, the advantage function is defined as:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s), \tag{6}$$

where $Q^\pi(s, a)$ is the action-value function representing the expected return of taking action a in state s and thereafter following policy π.

In the context of Dueling DDQN, these two functions are represented by separate streams within the neural network architecture. At the final layer, the streams are combined to compute the Q-values for each action in the action space. The mathematical formulation for this combining process is:

$$\begin{aligned} Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \\ \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right), \end{aligned} \tag{7}$$

here θ, α and β represent the parameters of the shared layers in the neural network, the advantage stream and the value stream, respectively. Finally, $|\mathcal{A}|$ denotes the size of the action space and the sum over $\alpha' \in \mathcal{A}$ is used to average the advantage estimates [7].

E. Learning Mechanism

The reward function $\mathcal{R}(s, a, s')$ maps a state–action–next-state tuple to a scalar reward $r \in \mathbb{R}$, balancing resource efficiency, over-provisioning cost and system stability. The agent stores transition tuples (s, a, r, s') in a replay buffer and samples mini-batches to update its policy network using the following learning target [7]:

$$y = r + \gamma\, Q_{\text{target}}\left(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-\right), \tag{8}$$

where r is the observed reward for executing action a in state s and transitioning to s', γ is the discount factor and θ^- denotes the parameters of the target network.

The action that maximizes the Q-value is selected using the online (policy) network with parameters θ and its value is evaluated using the target network. This separation reduces

overestimation bias and improves stability. The target network parameters θ^- are updated via soft updates from the online network during training. The full training procedure is summarized in Algorithm 1.

F. Confidence Mechanism

To enhance the reliability and stability of the RL agent's decisions, an entropy-based confidence mechanism is employed. Throughout each episode, the agent produces a sequence of predicted actions for every service instance. At the end of the episode, the empirical distribution of these actions is constructed and the associated uncertainty is quantified using Shannon entropy [3]. A dynamic entropy threshold $\tau(\epsilon)$ is used to assess the confidence of the agent's action predictions. This threshold is proportional to the current exploration rate ϵ, permitting higher decision variability during early training. As exploration decays, the threshold becomes stricter, thereby enforcing greater consistency in the agent's actions. This adaptive filtering mechanism suppresses unstable or exploratory decisions from being applied to the environment. If the entropy of the predicted action distribution exceeds $\tau(\epsilon)$, indicating high variability and low confidence, the action is discarded. Conversely, if the entropy falls below the threshold, the action is deemed confident and executed.

This mechanism serves as a safeguard against erratic or exploratory behavior being applied in production, improving both decision quality and system stability. Formally, let $\mathcal{A}_p = \{a_1, a_2, \ldots, a_T\}$ denote the action history for pod p across T decision steps. Let $\pi_p(a)$ denote the empirical probability mass function over actions. The entropy is computed as:

$$H(\pi_p) = -\sum_{a \in \mathcal{A}} \pi_p(a) \ln \pi_p(a), \tag{9}$$

where $\mathcal{A}$ is the discrete action set. A dynamic entropy threshold $\tau(\epsilon)$, parameterized by the agent's current exploration rate ϵ, is then applied to determine confidence. Only actions for which $H(\pi_p) < \tau(\epsilon)$ are considered confident and executed. This mechanism acts as a confidence filter, preventing the application of uncertain or highly variable decisions during unstable learning phases. The threshold's dynamic adjustment is defined as:

$$\tau(\epsilon) = \max(\tau_{\text{min}}, k \cdot \epsilon), \tag{10}$$

where τ_{min} is the minimum entropy threshold and k is a tunable coefficient that scales with the agent's current exploration rate ϵ. This formulation aligns the agent's decision confidence with its exploration phase so during early training greater variability is allowed, while, in later stages, stricter thresholds ensure that only consistent, low-entropy actions are executed.

Since the maximum entropy of a uniform distribution over K discrete actions is $\ln K$, the entropy of an action distribution, $H(\pi_p)$, is bounded within the interval $[0, \ln K]$. The dynamic threshold scales with the agent's current exploration rate ϵ, such that only actions satisfying

$$H(\pi_p) < \tau(\epsilon), \quad \forall\, \epsilon \in [0, 1], \tag{11}$$

are executed. This filtering mechanism allows greater variability during early training and enforces stricter confidence requirements as exploration decays, effectively suppressing unstable or high-uncertainty decisions.

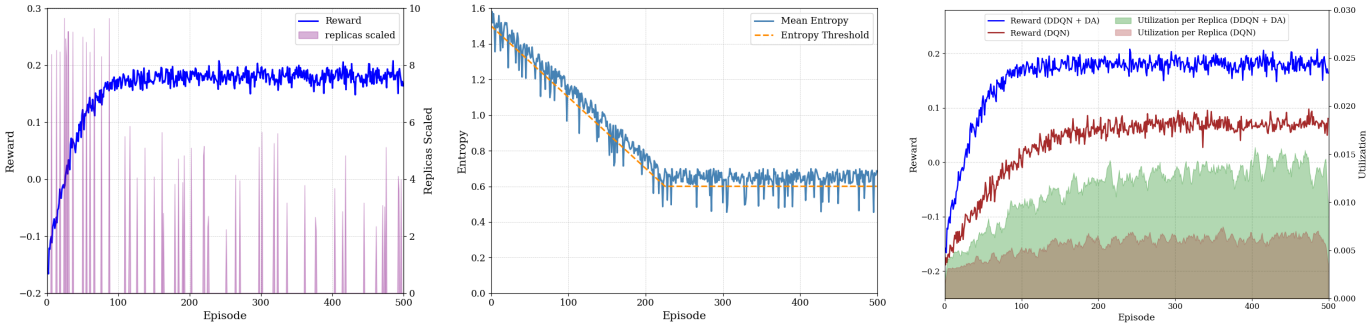
Algorithm 1 Dueling DDQN with Confidence Filtering

```
Initialize replay buffer M, Q-network Q, target network Q̂, epsilon ε, batch size B
for each episode do
    for each step do
        Observe s_t, select a_t via ε-greedy policy from Q
        if H(π_p) < τ(ε) then
            Execute a_t, observe r_t, s_{t+1}
            store (s_t, a_t, r_t, s_{t+1}) in M
        end if
        if |M| > B then
            Sample batch from M
            for each (s, a, r, s') in batch do
                Set the target to the reward r_i plus the
                discounted future reward with action
                selection using the online network
                and action evaluation using the target network
                Update Q via gradient descent
            end for
        end if
        Soft update Q̂
    end for
    Decay ε
end for
```

IV. Performance Experimentation

Performance evaluations were conducted within a multi-node cloud-edge environment under dynamic traffic patterns, reflective of the diverse B5G service demands. The RL agent was implemented using PyTorch [8], facilitating modular implementation of the Dueling DDQN architecture. Real-time resource metrics were collected from each node to form the agent's observation space, where each node received requests from virtual users following a stochastic arrival pattern. A static baseline with a fixed replica count of three and a horizontal pod autoscaler (HPA) baseline were used to assess SLA adherence under non-adaptive and rule-based conditions.

The progression of total reward per episode alongside the frequency of scaling actions is depicted in Fig 3a. Initially, high variance in both reward and scaling frequency is observed, attributable to the agent's exploratory behavior under a high exploration rate ϵ. As training progresses and the policy converges, the entropy-based confidence mechanism filters high-uncertainty actions, resulting in more stable and deliberate scaling. The reduction in replica fluctuation, coupled with an upward trend in cumulative reward, indicates that the agent is progressively refining its policy toward stability and efficiency. To further quantify the agent's decision confidence over time, Fig. 3b illustrates the mean action entropy per episode alongside the dynamic entropy threshold $\tau(\epsilon)$, showcasing that the confidence mechanism is effectively filtering exploratory noise while allowing deliberate actions to propagate. Subsequently, a comparison between the proposed Dueling DDQN agent and the standard DQN baseline, in terms of normalized CPU utilization per active replica, is



(a) Reward to replicas scaled (b) Mean entropy per episode (c) Reward to utilization

Fig. 3: Agent reward compared to replicas scaled (a), mean entropy per episode (b) and CPU utilization per replica (c).

depicted in Fig. 3c. A positive correlation is observed between reward accumulation and resource efficiency, demonstrating the agent's ability to learn policies that optimize CPU usage while minimizing unnecessary replica overhead. Over time, the utilization per replica begins to stabilize, suggesting convergence toward a consistent operating regime that balances workload demands with efficient resource allocation. This plateau reflects the agent's identification of an energy-conscious provisioning strategy in which system performance is sustained with minimal excess capacity. The observed stabilization further suggests that the agent internalizes a longer-term notion of sustainable, zero-touch orchestration derived from its state observations, consistently outperforming the DQN baseline in terms of resource efficiency. Moreover, Fig. 3c further presents a convergence comparison between the proposed Dueling DDQN approach and the standard DQN. This comparison highlights the effectiveness of the dueling architecture in accelerating learning and enhancing policy robustness in complex, resource-constrained environments. Finally, Fig. 4 presents the success rate and 95th percentile latency under increasing load. The static baseline degrades significantly under stress, while the HPA baseline achieves low latency under steady conditions but exhibits a notable drop rate due to its reactive nature and reliance on threshold-based triggers. The DQN agent performs moderately better and the proposed approach maintains high success rates and low latency by responsively adapting replica counts, ensuring more reliable SLA compliance under contention.

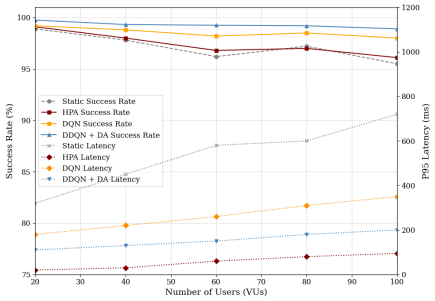


Fig. 4: VUs to success rate and 95th percentile latency

V. Conclusion

In summary, this work presents a Dueling DDQN-based autoscaling framework for zero-touch management of dynamic edge environments, enhanced with entropy-based confidence filtering, which acts as a meta-cognitive mechanism, reinforcing the agent's awareness of its operating context. The framework functions as a robust mechanism that leverages multi-level resource signals to learn holistic, system-level policies, addressing the limitations of reactive or threshold-based methods. Experimental results demonstrate that the integration of Dueling DDQN with entropy-based confidence filtering enables efficient, energy-aware and autonomous scaling, paving the way for zero-touch management in dynamic, resource-constrained edge environments.

Acknowledgment

This work was supported by the European Commission research project ADROIT6G with Grant agreement ID: 101095363.

References

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[2] C. Cicconetti, M. Conti, A. Passarella, and D. Sabella, "Toward distributed computing environments with serverless solutions in edge systems," *IEEE Communications Magazine*, vol. 58, no. 3, pp. 40–46, 2020.

[3] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[4] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–5.

[5] S. Zhang, T. Wu, M. Pan, C. Zhang, and Y. Yu, "A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning," in *2020 IEEE International Conference on Web Services (ICWS)*, 2020, pp. 489–497.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, 2015.

[7] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, 2016, p. 1995–2003.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.