Scaling-Ops: SLA-Compliant Resource Management via Offline Reinforcement Learning Pipeline

Ioannis Kyriakopoulos*, Dimitrios Selis§*, Kostas Ramantas†, Christos Verikoukis‡*
*Industrial Systems Institute, Patras, Greece
§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: ikyriakopoulos@isi.gr, dimitrios.selis@upc.edu, kramantas@iquadrat.com, cveri@ceid.upatras.gr

***Abstract*—Contemporary service architectures have become complex and challenging to maintain. The continuous demand for high performance, low-latency services in today's intricate technological landscape necessitates robust, scalable and resource-efficient infrastructures. Acknowledging the challenges that arise, service providers aim to meet the growing demand within existing infrastructure by employing advanced cloud techniques. These approaches have emerged at the forefront, offering streamlined solutions for organizing service architectures while contending with ongoing maintenance requirements and the need for continuous improvement. The focus of service providers has shifted toward the integration of artificial intelligence (AI), both as a service and as a resource orchestrator. AI models have become popular, due to their ability to identify complex patterns and propose optimal solutions at the service level, particularly with regard to system resource utilization. The primary objective of this paper is to tackle the auto-scaling problem, so as to achieve optimal resource utilization in cloud-edge environments, using reinforcement learning (RL). We propose two complementary system models: an online Kubernetes-based auto-scaling model for real-time resource utilization and a novel offline RLOps pipeline for training RL models, designed to minimize system overhead during training and accelerate experimentation with multiple model candidates.**

***Index Terms*—Resource management, auto-scaling, reinforcement learning (RL), Kubernetes, RLOps**

I. Introduction

Modern cloud computing has altered how applications are deployed and how the infrastructure is managed. By taking advantage of the distributed nature of cloud environments, contemporary deployments can now be hosted across a multitude of machines, rather than being confined to a single physical server. This shift, along with the continuous and rapid network infrastructure improvements, has introduced novel techniques and technologies, that improve service deployment in cloud environments. Central to this transformation are cloud-native technologies. Docker [1] serves as a tool for organizing applications into units known as containers. Containers are useful in production, offering lightweight and efficient packaging of services. Instead of deploying a monolithic block of code that contains the entire application's logic, Docker enables the organization of applications into smaller units. Containers are also portable and can be transferred across different machines without facing kernel restrictions, unlike classical virtual machines. While organizing an application into multiple containers can be advantageous, the problem of container orchestration arises. Kubernetes [2] addresses this by functioning as a container orchestrator, managing containers at scale in a distributed system model. It streamlines application development by offering advanced functionalities, such as service discovery, storage orchestration, self-healing, scaling and more. Altogether, these technologies not only accelerate development, but also possess highly scalable and resilient operational capabilities, a key feature when addressing the workload demands frequently associated with modern infrastructure.

Within the sophisticated Kubernetes ecosystem, resource management is a critical feature and is managed through two mechanisms that operate at the level of pods. A pod is the smallest deployable unit in the Kubernetes object model, consisting of one or multiple containers that share the same network, storage and computational units. Each of these two mechanisms is designed to align computational resources with the fluctuating demands of the applications they support. The Horizontal Pod Autoscaler (HPA) adjusts the number of pod instances in response to observed data by adjusting the number of containers based on predefined resource limits, most commonly CPU utilization. The Vertical Pod Autoscaler (VPA) dynamically alters the CPU and memory limits for existing pods, aiming to optimize resource availability for individual running instances. Despite their widespread adoption and utility, these tools rely on threshold-based logic. It is often proved insufficient and prone to significant limitations when confronted with rapidly changing load patterns. As a result, they are prone to computational resource over-utilization, which adds operational cost or trigger

service level agreement (SLA) violations, impacting both user experience and business credibility.

To effectively address these limitations and in general, to enhance the responsiveness and efficiency of cloud infrastructure, there is a growing interest in integrating artificial intelligence (AI) techniques into cloud-native systems. Reinforcement learning (RL) has demonstrated considerable promise. Its unique strength lies in its ability to learn from highly adaptive and optimal scaling strategies through continuous interaction with the environment. However, the operationalization of RL for real-time auto-scaling within production environments currently presents a formidable obstacle: the direct training of complex RL models within live Kubernetes clusters. The training process of such models adds considerable overhead to the system state and is significantly resource-intensive, frequently stretching over periods of days. Such training cycles make RL models inflexible for the rapid agility required in dynamic production environments, where workloads can shift suddenly and unpredictably. This paper introduces a novel approach to address the extensive training time required by RLOps pipelines, effectively compressing it from a time-consuming process spanning days to just minutes. Our approach strategically blends a lightweight linear regression layer with two custom environments, specifically leveraging Gymnasium's robust capabilities for simulation. The first environment is built for intensive offline training, harnessing vast quantities of historical resource scaling data to accurately simulate realistic system states and corresponding workload patterns. The second environment, which is optimized for model evaluation, operates with efficacy solely on incoming request data, thus efficiently eliminating the need for full system metrics. Furthermore, a lightweight linear regression is utilized to calculate the coefficients of key performance indicators, specifically SLA, CPU utilization and Requests Per Second (RPS), providing crucial feedback directly to the RL agent. By integrating both the training and evaluation phases into a robust, isolated offline environment, our system gains the ability to generate and test multiple RL models in a significantly reduced amount of time. This operational efficiency profoundly empowers operators to swiftly identify and choose the most effective scaling strategy based on empirical performance indicators, such as the precise range of pod counts utilized for optimal resource allocation and performance. In doing so, this innovative pipeline brings to the foreground truly intelligent, RL-driven auto-scaling within practical reach for real-world Kubernetes-based environments, laying the foundation for faster, more efficient and smarter infrastructure management.

II. Related Work

Kubernetes auto-scaling has attracted considerable attention, especially as cloud-native applications grow more resource-intensive and workload patterns less predictable. While HPA scalers handle basic scaling using standard resource metrics, they are often rendered unable to cope with rapidly changing workloads. These scalers exhibit delayed responsiveness and demonstrate limited adaptability to fluctuations in workloads, potentially resulting in suboptimal resource usage and SLA violations. To address these limitations, researchers have explored the application of RL in order to enable more intelligent scaling decisions, with respect to SLAs. DScaler [3] introduces a deep reinforcement learning (DRL) [4] methodology to identify optimal scaling patterns as the workload evolves over time.By formulating the auto-scaling problem as a Markov Decision Process (MDP), DScaler allows an agent to observe system metrics and learn scaling policies that optimize future performance and cost. The authors compare DScaler with HPA, focusing primarily on CPU utilization and conclude that DScaler adapts dynamically to changes in workload behavior and avoids scaling actions that could lead to inefficient resource utilization. A-SARSA [5] proposes a hybrid model that combines RL with a time-series forecasting technique. The idea behind A-SARSA is that workload fluctuations exhibit patterns that can be identified and predicted using forecasting models. By incorporating predictions of future load into the MDP, the RL agent can take proactive scaling actions rather than reactive ones, enabling better resource utilization and reduced response time violations. However, a significant limitation remains in the training time required.

All of these methods rely heavily on online training within a live Kubernetes cluster. According to our observations and the system models described in prior literature, each scaling action introduces considerable delays, often exceeding standard operational response times by more than 50% to 200%, thereby leading to prolonged training cycles that hinder experimentation and limit the ability to adapt models to evolving applications or dynamic traffic patterns. This limitation prevents service providers from experimenting quickly and adapting models to new applications or traffic patterns. It also adds unnecessary load to production systems during training. Our work addresses the research gap concerning the system overhead introduced by Kubernetes when training RL models. Instead of proposing a new scaling policy, we introduce a novel RLOps pipeline designed to accelerate training for RL-based auto-scaling. This pipeline is deployed entirely offline and can generate and evaluate multiple RL agent instances in under ten minutes. To the best of our knowledge, no existing work offers an offline training pipeline for Kubernetes auto-scaling. All prior methods assume that training occurs within the live system, rendering their deployment and iteration cycle slow and often impractical.

By removing this constraint, the proposed pipeline

significantly improves the feasibility of applying RL-based auto-scaling in real-world Kubernetes environments. In summary, the contribution of this work lies in introducing a more efficient approach to training and evaluating RL agents. The pipeline enables the offline comparison of different strategies and facilitates the deployment of only the best-performing models, thereby advancing RL-based auto-scaling toward practical applicability.

III. Problem Definition and System Model

A. Problem Definition

This section addresses the challenges of auto-scaling using RL agents, as well as the time-consuming nature of training such agents. One of the key features addressed by Kubernetes is container scaling, a resource provisioning mechanism that adjusts container instances according to collected metrics (e.g. CPU, Memory, RPS). Although classic HPA offers a basic solution, it is often insufficient for complex scenarios.

To address this, we propose a DRL approach, which leverages neural networks (NNs) to estimate Q-values [4], which correspond to the expected cumulative reward an agent can obtain given an action taken in a specific state following a policy π. Unlike the classic Q-learning algorithm, which maintains a discrete Q-table of size $n \times m$ (where n represents the size of the action space and m the size of the state space), the DRL model generalizes across large and continuous state-action spaces. This results in a more robust and adaptable model that is easier to train and customize for specific system requirements.

Another challenge observed during model training is the system overhead introduced by Kubernetes. Our calculations indicate that each scaling cycle takes approximately 45-120 seconds (excluding scaling failures). When the duration is multiplied by the minimum number of timesteps required to produce a deployable model, it becomes evident that a single training epoch may be completed over a period of several days. This time delay significantly increases the complexity of using RL models, making them less practical for tasks such as system auto-scaling and scheduling.

To address this limitation, we propose a training pipeline specifically designed to reduce Kubernetes-related overhead. The pipeline leverages historical data collected by Prometheus and incorporates theoretical assumptions about the system's future state, thereby optimizing the training process.

B. System Model

Figure 1 illustrates the architecture of the proposed system model. The Ingress component serves as the entry point for user requests, routing traffic from the external environment to the application, *Service A*, which corresponds to the Bookinfo service [6]. Performance and usage data are drawn using Prometheus, aided by CAdvisor for system level metrics and Istiod for network telemetry. These data are then stored in Prometheus Time Series Database (TSDB). At the core of the system is the BiScaler container, which hosts the RL model within a Kubernetes Cluster. The BiScaler fetches the current state of the cluster and the RL model determines an appropriate scaling action. This action is published via gRPC server, which is then consumed by the KEDA pod [7] as a gRPC client. KEDA applies the scaling decision to the cluster accordingly.

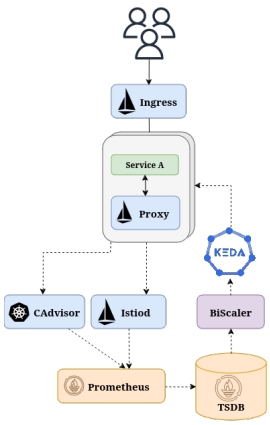

Fig. 1: System Model

C. Container Architecure

The principals used are significantly different from those in DScaler [3]. There are notable similarities in both the reward function and query structures, with only minor modifications. Specifically, the reward function is defined as:

$$\text{reward} = \text{CPU} + b_i - 0.3 \cdot \text{Replicas}, \tag{1}$$

where b_i represents the delay-related component. Specifically,

$$b_i = \begin{cases} \dfrac{1}{1 + \dfrac{Del_{\text{obs}}}{\text{SLA}}}, & \text{if } Del_{\text{obs}} < 250\,\text{ms}, \\ -2, & \text{if } Del_{\text{obs}} \geq 250\,\text{ms}, \end{cases} \tag{2}$$

where Del_{obs} denotes the observed service delay. The reward function is a combination of three key metrics designed to optimize container performance. The CPU metric incentivizes the system to maximize resource utilization, rewarding actions that lead to higher CPU usage. At the same time, the system is penalized for increasing the Replica count, as indicated by the term $-0.3 \cdot Replicas$, encouraging it to be efficient with the current resources rather than simply scaling up. The most critical component, of this reward function, is the penalty term b_i, which is correlated to the service delay Del_{obs} and prevents SLA violations. This term applies a penalty as the delay approaches 250

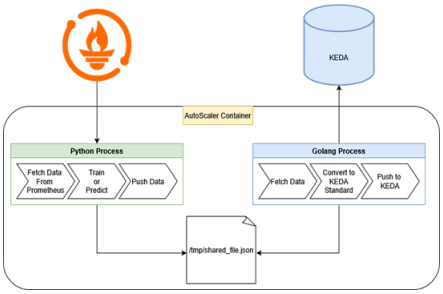

Fig. 2: RL Container Architecture

(milliseconds) and a relatively severe penalty of -2 as the delay exceeds the threshold of 250(milliseconds). In general, our goal is for the reward function to incentivize actions, that maximize resource utilization without causing any SLA violations. [1]

Figure 2 presents BiScaler's internal architecture. The system is built around two primary processes, managed by the `supervisord` process manager.

The Python process implements the core logic of the RL model. Its workflow is as follows:

- It fetches the current state of the target pod (i.e., the pod to be scaled) from Prometheus.
- Based on the exploration-exploitation strategy, the agent either performs a random action (exploration) or chooses the optimal action (exploitation).
- The new replica count is calculated as follows:

$$\text{Replica\_Count}_{t+1} = \text{Replica\_Count}_t + \text{action}$$

at time t, and is written to a shared file.

Subsequently, the Golang process takes action:

- It initializes a gRPC server.
- KEDA reads the desired number of replicas from the shared file.
- The data is converted into gRPC format and posted to the server.
- KEDA, acting as a gRPC client, retrieves the desired replica count and performs the scaling operation.

The containerized architecture encapsulates the DRL scaler's logic. This process repeats at each timestep, or whenever the agent is trained—either integrated within the system or solely functioning as an inference scaler.

D. Pipeline

Figure 3 illustrates our proposed RLOps pipeline, which acts as an offline trainer, designed to accelerate Deep Q-Networks (DQN) training by reducing system overhead. A core aspect of this pipeline is the mathematical formulation of the Kubernetes system's next state.

[1]The constant 0.3 is a system-specific parameter defined for this experiment. It may not contribute meaningfully to the reward function in other experimental environments.

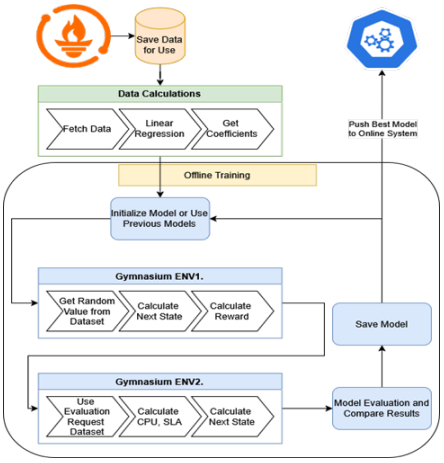

Fig. 3: RL trainer pipeline

1) Regressor: The first step in training DQN models is to acquire and process data from Prometheus. The SLA serves as a critical metric for evaluating the DQN model's performance. Defining the correlation among SLA, CPU utilization and RPS contributes to subsequent calculations.

$$\text{SLA} = a_1 \cdot \text{CPU} + a_2 \cdot \text{Reqs} \tag{3}$$

Consequently, a linear regression model, selected for its interpretability, is utilized to derive the coefficients, as defined in Equation 3. These coefficients are applied in two distinct scenarios: in Gymnasium ENV1, they facilitate the prediction of the next SLA state after an action is taken and in Gymnasium ENV2, they enable real-time calculation of both current and prospective SLA values.

2) Gymnasium ENV1: The agent's training occurs within Gymnasium ENV1, where it performs random sampling across the previously used dataset(for calculating coefficients). Based on the fetched state, the agent selects an action. This action is evaluated using the equations below:

$$CPU_{new} = CPU \cdot \frac{currentPods}{desiredPods} \tag{4}$$

$$SLA = \gamma \cdot (a_1 \cdot CPU_{new} + a_2 \cdot Reqs_{new}) \;\; \forall \gamma \;\in\; [0, 1] \tag{5}$$

Equation 4 calculates the next CPU utilization based on the ratio of the current number of pods to the agent's desired number of pods. Equation 5 calculates the next SLA state. It incorporates a smoothing factor γ to reduce the value fluctuations and align predicted values closer to actual observations.

3) Gymnasium ENV2: This pipeline segment evaluates the trained models, using only RPS data, within Gymnasium ENV2. Gymnasium ENV2 is responsible for calculating both the current state of the environment (CPU, RPS, pods) and next states. The next

value of the SLA calculation remains consistent, as demonstrated in Equation 5.

$$N = Requests \cdot \frac{\overline{Replicas}}{\overline{Requests}} \tag{6}$$

$$CPU_{new} = \frac{\overline{CPU}}{\overline{Replicas}} \cdot \frac{N}{Pods_{des}} \tag{7}$$

$$Reqs_{new} = min\{Reqs, \frac{\overline{Reqs}}{\overline{Pods}} \cdot Pods_{des}\} \tag{8}$$

Here Equation 6 determines the number of active pods by multiplying the number of requests by the ratio of mean replicas over mean requests. Equation 7 calculates the next value of CPU, it processes the current and desired (by the RL agent) pod counts, multiplying them by the mean CPU values per replica. Finally, Equation 8 determines the next RPS, leveraging the mean values of requests and pods to calculate a constant, which is then multiplied by the desired number of pods. Algorithm 1 depicts the algorithmic process that takes place. The evaluation process commences after the agent has completed a testing period. This evaluation is simplified by solely considering the average number of pods metric, which is deemed sufficient and necessary to prevent overfitting (e.g., the agent exclusively selecting actions at pod boundaries). The evaluation dataset used in this study is illustrated in Figure 4. It depicts the pattern of user requests within the deployed service, characterized by recurring slopes over time. This data serves as the input for the evaluation phase, where the trained models are assessed. During this process, the agents compute the system state based on the request dynamics, providing the foundation for evaluating model performance.

Upon completion of the training, testing and evaluation phases, multiple RL models are produced. The most suitable models are then uploaded to Kubernetes, based on user preference.

IV. Results

This section evaluates the models generated by the RL pipeline, which is designed to act as a DQN scaler for Kubernetes. The pipeline iteratively produces models, with each successive model aiming to improve the initial model.

The models were assessed using the online system which is depicted in Figure 1. Within this system, a model is fitted and used to select actions based on the current state provided by Prometheus. The chosen action is then applied to the Kubernetes cluster, utilizing KEDA, thereby triggering a corresponding scaling event. Evaluation was based on three key metrics derived from the MDP formulation:

- Number of Replicas: To assess the scaling behavior and resource allocation.
- CPU Utilization: To gauge resource management efficiency.

Algorithm 1 Integrated SLA Prediction and DQN Autoscaler Pipeline

Input: $\mathbf{X}_{\text{train}} = [\text{CPU}, \text{Req}]$, $\mathbf{y}_{\text{train}} = \text{SLA}$, Traffic $\mathcal{T}$
Consts: $\alpha = 0.4$, $T_{\text{SLA}} = 250$, γ
Phase 1: Coefficient Calculation
$\boldsymbol{\beta}_{\text{full}} \leftarrow (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}_{\text{train}}$
$\boldsymbol{\beta} = [\beta_0, \beta_1] = [\boldsymbol{\beta}_{\text{full}}[1], \boldsymbol{\beta}_{\text{full}}[2]]$
Phase 2: DQN Training (OfflineEnv)
Init $Q(\mathbf{s}, a; \theta)$, $Q_{\text{tgt}}(\mathbf{s}, a; \theta')$
$\mathbf{s} \leftarrow \text{ENV.reset}()$
while training a model **do**
  $a \leftarrow \text{SelectAction}(\mathbf{s})$
  $\text{Rep}_{\text{new}} \leftarrow \max(1, \text{Rep}_{\text{old}} + a)$
  $\text{CPU}_{\text{new}} \leftarrow \text{CPU}_{\text{old}} \cdot \frac{\text{Rep}_{\text{old}}}{\text{Rep}_{\text{new}}}$
  $\mathbf{s}' \leftarrow [\text{CPU}_{\text{new}}, \text{Req}_{\text{old}}, \text{Rep}_{\text{new}}]$
  $\text{RTT} \leftarrow \alpha(\beta_0 \text{CPU}_{\text{new}} + \beta_1 \text{Req}_{\text{old}})$
  Calculate Reward Equation 1 and 2
  $y \leftarrow reward + \gamma \cdot \max_{a'} Q_{\text{tgt}}(\mathbf{s}', a'; \theta')$
  Gradient descent Step: $\nabla_\theta (Q(\mathbf{s}, a; \theta) - y)^2$
  $\mathbf{s} \leftarrow \mathbf{s}'$
end while
Phase 3: Evaluation (OfflineEnv2)
$\mathbf{s} \leftarrow \text{ENV}_{\text{eval}}\text{.reset}()$
Load $Q(s, a; \theta)$
for $\text{Net}_{\text{act}} \in \mathcal{T}$ **do**
  $a \leftarrow \arg\max_{a'} Q(\mathbf{s}, a')$
  $\text{Rep}_{\text{new}} \leftarrow \text{Rep}_{\text{old}} + a$
  Compute CPU_{new} via Eq. (7).
  Compute Reqs_{new} via Eq. (8).
  Compute SLA :
  $\text{RTT}_{\text{eval}} \leftarrow \alpha(\beta_0 \text{CPU}_{\text{new}} + \beta_1 \text{Reqs}_{\text{new}})$
  $Log_{Preds} \rightarrow (State, Reward, NextState)$
end for
Output: $Models, Logs$

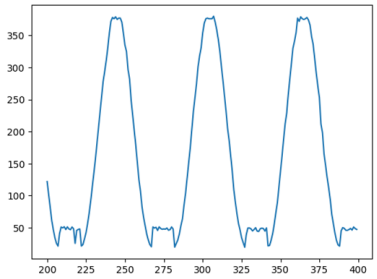

Fig. 4: Evaluation Data

- RPS: To measure the system's ability to handle incoming traffic.
- Request Delay: Defined as an SLA variable, representing the quality of user experience.

For comparative analysis, three scaling mechanisms were evaluated: a traditional HPA serving as a control, Model 1 (the initial model) and Model 2 (a subsequent model derived from Model 1). HPA was configured with 70% CPU utilization target.

Figure 5 illustrates the performance across these metrics. Figure 5, representing the number of replicas, clearly shows that Model 1 consistently utilizes fewer replicas compared to the HPA. While in Figure 6, Model 1 consequently exhibits higher CPU utilization, these values are rendered acceptable, because with more replicas, CPU utilization per replica generally

decreases. The HPA, despite allocating more replicas, results in lower actual CPU utilization per replica, suggesting over-provisioning, which can be observed in Figure 6.

Model 1 demonstrates a tendency for SLA violations, as depicted in Figures 7 and 8. This behavior indicates that Model 1 is efficiently leveraging container resources to their fullest, but becomes prone to user request delays and subsequent SLA violations during sudden increases in network load. This highlights a critical trade-off between aggressive resource utilization and maintaining service quality under fluctuating loads. Finally, Table 1 presents a comparative analysis of key metrics, including SLA, number of replicas and CPU utilization, to evaluate the performance of HPA, Model 1 and Model 2.

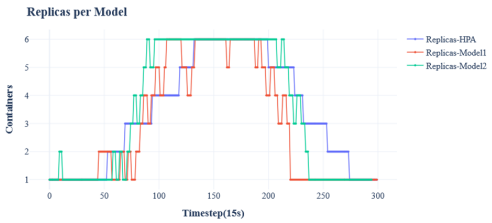

Fig. 5: Number of Replicas

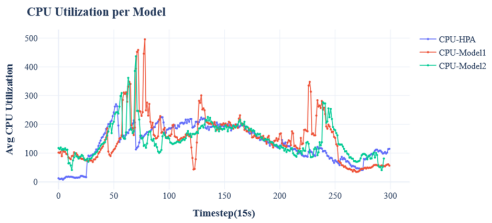

Fig. 6: CPU Utilization

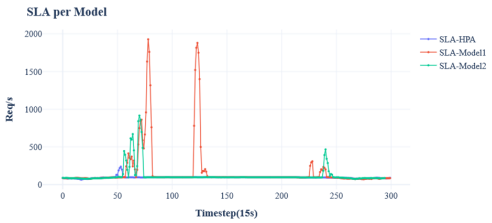

Fig. 7: SLA Score

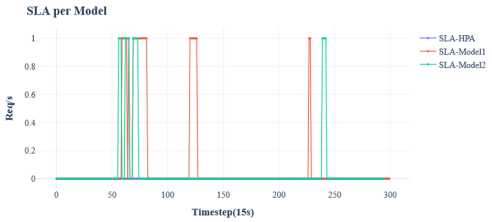

Fig. 8: SLA Violation

Metric	HPA	Model 1	Model 2
SLA (Req/s)	94	173	120
Replicas	3.39	2.95	3.47
CPU Utilization (mCores)	135	152	146

TABLE I: Mean-Comparison of HPA with Model 1 and 2

V. Conclusions

This paper introduced a novel RLOps pipeline designed to significantly improve the training time of RL driven auto-scaling in dynamic cloud environments. By addressing the bottleneck of time-consuming RL model training, the proposed system successfully reduces training duration from days to minutes, an essential improvement for real-world applications.

Our methodology leverages two distinct Gymnasium environments and a linear regression model to effectively define the relationship between SLA, CPU and RPS. The first environment hosts efficient offline training using Prometheus data, while the second enables state calculation through system data mathematical formulation. A key innovation of this pipeline is the ability to generate multiple, ready-to-deploy, RL models within a short period of time. By evaluating these models based on metrics like the average of pod counts, the pipeline gives the ability of model selection based on optimal auto-scaling behavior, tailored to specific operational demands.

This work represents a significant step toward enabling intelligent auto-scaling solutions that are more robust, accessible and practical for dynamic Kubernetes deployments, effectively bridging the gap between theoretical RL and efficient implementation in production-ready cloud environments.

Acknowledgment

This work was supported by the European Commission research projects ADROIT6G (ID: 101095363) and FLECON6G (ID: 101192462).

References

[1] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

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

[3] Z. Xiao and S. Hu, "Dscaler: A horizontal autoscaler of microservice based on deep reinforcement learning," in *2022 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2022, pp. 1–6.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[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] Istio, "Bookinfo application," https://istio.io/latest/docs/examples/bookinfo/.

[7] KEDA, "Kubernetes event-driven autoscaling (keda)," https://keda.sh.