Dynamic and Forecast-based Containers Autoscaling for Kubernetes with Reinforcement Learning

Alfredo Lipari[1], Gabriele Proietti Mattia[1], and Roberto Beraldi[1]

[1]Department of Computer, Control and Management Engineering Antonio Ruberti, Sapienza University of Rome, Rome, Italy

***Abstract*—Efficient resource management in Kubernetes is crucial for optimizing performance and cost in cloud computing environments. Traditional autoscaling methods react to workload changes but often fail to predict them, leading to underutilization of resources or performance degradation. This paper introduces a model-free Reinforcement Learning (RL) based autoscaler that leverages a deep Q-Network (DQN) alongside a Long Short-Term Memory (LSTM) network for workload forecasting. By predicting future request rates, our autoscaler proactively adjusts the number of container replicas, ensuring compliance to Service Level Objectives (SLOs) while minimizing resource usage. Experimental results demonstrate that our approach outperforms standard Kubernetes auto-scaling strategies, achieving a resource consumption reduction of up to 10% and improving performance levels up to 30% compared to the default auto-scaling algorithm.**

I. Introduction

Cloud computing has revolutionized the IT industry by enabling scalable, flexible, and efficient resource management. At the heart of this transformation lies containerization, a lightweight virtualization technology that isolates applications and their dependencies in portable, self-contained units. Microservice architectures enhance cloud-native applications' flexibility by decomposing applications into independently deployable services, allowing fine-grained resource allocation. Kubernetes automates the deployment and scaling of these containerized applications, featuring the Horizontal Pod Autoscaler (HPA) [1] to adjust container replicas based on resource utilization. While both horizontal scaling (adjusting replica count) and vertical scaling (adjusting resource allocation per replica) are possible in Kubernetes, horizontal scaling offers superior flexibility for stateless microservices and better fault tolerance through redundancy, which is why we focus on it in this work. Despite its widespread use, the HPA has limitations. Relying solely on reactive scaling based on single-metric triggers, it often fails to address microservices' nuanced demands. High CPU utilization doesn't always correlate with degraded performance [1], resulting in inefficient scaling decisions. Furthermore, HPA's inability to anticipate workload changes causes either resource overprovisioning or performance degradation, highlighting the need for smarter autoscaling mechanisms. This paper presents a novel Reinforcement Learning (RL) based autoscaler that dynamically adjusts computational resources based on real-time workload variations. Our proposed autoscaler integrates a Double Deep Q-Network (DQN) with a Long Short-Term Memory (LSTM) network for workload prediction, transitioning from reactive to predictive scaling for enhanced resource efficiency and SLA compliance. The contributions of this paper can be summarized as follows:

- A novel RL-based autoscaler that integrates DQN and LSTM to predict workload patterns and optimize scaling decisions without relying on predefined thresholds.
- A unified architecture for both simulation and deployment, facilitating seamless transitions from controlled environments to production clusters in GKE.
- Experimental results showing that the proposed autoscaler reduces resource consumption by up to 10

The paper is structured as follows: Section II discusses related works, Section III presents the system model, and Section IV details the proposed autoscaling framework. Section V provides the experimental results, and Section VI concludes the paper.

II. Related Work

Rossi et al. [4] introduced dynamic, multi-metric threshold-based scaling policies using RL to autonomously adjust CPU and memory utilization thresholds. Their multi-agent Model-Based RL and Deep Q-Learning (DQL) approaches demonstrated improved deployment objectives and reduced execution times. However, their reliance on model-based algorithm may not be feasible in complete new scenarios, where having a complete map of the interactions between services is not possible. Our model-free agents learns optimal scaling actions on complete new workload profiles and patterns and acts directly on the number of replicas instead on the threshold, enabling more adaptive and responsive scaling.

Wang et al. [5] proposed a container scaling strategy using model-based RL to address inefficiencies in Kubernetes' HPA. By formulating the scaling problem as a Markov Decision Process (MDP), they developed an RL algorithm that improved resource utilization and service quality. However, model-based RL approaches are sensitive to modeling errors, reducing adaptability to unforeseen workload patterns. In contrast, our

[1]Kubernetes Documentation, 'Horizontal Pod Autoscaler,' accessed December 14, 2024. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

model-free approach does not require an explicit environmental model if not only known parameters, that we will describe later in details, enhancing robustness and adaptability to unpredictable workloads.

Predictive autoscaling in Kubernetes typically involves tools like KEDA with PredictKube [2], which analyze historical traffic data, enabling proactive scaling for known events such as traffic surges during election cycles or holiday promotions. For instance, a news website might preemptively scale replicas ahead of election-day traffic surges, ensuring system responsiveness. While effective for predictable workload patterns, such methods rely on historical trends and predefined events. Our approach diverges by employing an LSTM-based predictor to forecast workload fluctuations dynamically, even for workloads without clear temporal patterns. By learning directly from interactions, our RL agent provides more robust and adaptive scaling, maintaining performance in unpredictable conditions.

While several advanced autoscaling approaches exist in the literature, our experimental evaluation focuses on comparing our predictive approach with both the industry-standard HPA and a non-predictive DQN agent. This non-predictive DQN agent effectively represents current RL-based autoscalers in the literature, particularly similar to Rossi et al.'s [4] approach. Due to time restriction and implementation complexity, we couldn't directly implement and compare with all alternative autoscaling algorithms, focusing instead on demonstrating the specific benefits of our prediction-enhanced approach.

In summary, while prior studies have advanced autoscaling through ML and RL, many depend on threshold-based policies or model-based RL, which can limit adaptability to highly dynamic workloads. Our approach addresses these limitations by:

1) Employing a Model-Free DQN Agent: Learning optimal scaling actions directly from environmental interactions eliminates the need for predefined thresholds, enhancing adaptability.
2) Integrating LSTM-Based Workload Prediction: Forecasting future request rates enables proactive scaling decisions, improving responsiveness to workload changes.
3) Create a realistic simulation environment: The agents must be trained in a environment that takes in consideration, when possible, the realistic factors of a real K8S cluster to be effectively production ready.

III. Designing the Reinforcement Learning-Based Autoscaler

In this section, we detail the core design of our Reinforcement Learning (RL)-based autoscaler. We first describe the Double Deep Q-Network (DQN) and we then introduce the Long Short-Term Memory (LSTM)-based workload predictor.

[2]Dysnix, "PredictKube," 2024. [Online]. Available: https://dysnix.com/predictkube.

A. Overview of the DQN Autoscaler

The DQN agent employs a neural network to approximate the Q-value function, mapping state-action pairs to expected rewards. The architecture includes an input layer accepting a five-dimensional state vector, two fully connected layers (64 and 32 neurons) with ReLU activation, and an output producing Q-values corresponding to scaling actions. Network weights use Xavier uniform initialization for stable convergence.

To improve training stability, the agent adopts Double DQN with a separate target network [9], separating action selection from evaluation. The agent also uses prioritized experience replay [8], storing past experiences with their temporal-difference errors and focusing on higher-error experiences for more efficient learning.
Experiences with higher TD errors are considered more important because they provide the agent with better learning opportunities. This approach also reduces the risk of overfitting to recent experiences by breaking the sequence of correlated samples, leading to faster and more stable learning of effective scaling policies.

1) Q-Value Update Equation: During training, the *main network* selects the action that maximizes the Q-value in the next state, while the *target network* evaluates this action to compute a more stable Q-value target. The target value is calculated as:

$$y_t = r_t + \gamma \cdot Q_{\text{target}}\left(s_{t+1}, \arg\max_a Q_{\text{main}}(s_{t+1}, a)\right),$$

where r_t is the immediate reward, γ is the discount factor that controls how much future rewards are considered, and Q_{main} and Q_{target} are the Q-values from the main and target networks, respectively.

The agent minimizes the difference between this target y_t and its current Q-value estimate $Q_{\text{main}}(s_t, a_t)$ using the mean squared error loss. This process adjusts the weights of the main network to bring its predictions closer to the target values.

To maintain stability during training, The target network weights are periodically updated using a soft update mechanism. This means the target network's parameters are slowly blended with those of the main network over time, ensuring that the target network evolves smoothly rather than abruptly.

2) Action Space: The action space $\mathcal{A}$ consists of three discrete actions:

- Scale Down (-1): Decrease the number of replicas by one, respecting the minimum limit.
- No Change (0): Maintain the current number of replicas.
- Scale Up (+1): Increase the number of replicas by one, respecting the maximum limit.

This discrete action space prevents abrupt changes in resource allocation, promoting system stability while accommodating workload fluctuations.

B. State and Reward Computation

1) State Representation: The DQN agent operates on a state space that captures the system's key operational metrics, providing a comprehensive view of both current performance and predicted future demand. Specifically, the state representation is a 5-dimensional vector:

1) Normalized CPU Utilization [0,1] (U): The ratio of current CPU usage to the maximum available CPU.
2) Normalized Number of Replicas [0,1] (N): The ratio of current replicas to the maximum allowable replicas.
3) Normalized Request Rate [0,1] (R): The current request rate normalized against the system's capacity.
4) Normalized Predicted Request Rate [0,1] (PR): The predicted future request rate generated by the LSTM workload predictor.
5) Rate Change [-1,1] (RR): The ratio between the predicted request rate and the previous request rate, reflecting workload trends.

While CPU utilization and replica counts are standard metrics used in traditional autoscalers, our approach introduces the predicted request rate and its change over time as innovations. These additions enable the agent to transition from reactive to proactive scaling, allowing it to anticipate changes in workload and adjust replicas preemptively.

2) Reward Calculation: The reward function directs the agent toward maintaining performance targets while minimizing resource usage. It consists of three main components, two of which are based on prior studies [4] and have been refined to accommodate better modeling after extensive testing:

$$R = -\left(C_{\text{perf}} + C_{\text{res}}\right) + C_{\text{align}}$$

Detailed breakdown:

a) Performance Cost (C_{perf}): The performance cost increases sharply when response times exceed $T_{\max}$. The parameter α allows operators to control how aggressively high latencies are penalized:

$$C_{\text{perf}} = \begin{cases} \frac{RT - T_{\min}}{T_{\max} - T_{\min}}, & \text{if } RT \leq T_{\max} \\ \alpha + \frac{RT - T_{\min}}{T_{\max} - T_{\min}}, & \text{if } RT > T_{\max} \end{cases}$$

b) Resource Cost (C_{res}): The resource cost grows nonlinearly with the number of replicas to discourage over provisioning. By tuning the parameter γ, the system penalizes high resource usage, while the parameter β rewards efficient usage when the number of replicas is within $r_{\min}$. This additional component was added after considerable testing to ensure that the agent prioritizes using the minimum number of replicas required to handle the workload. The resource cost is defined as

$$C_{\text{res}} = C_{\text{base}} + C_{\text{exp}} = \gamma \frac{r - r_{\min}}{N_{\max} - r_{\min}} + 0.1\left(\beta^{(r-1)} - 1\right)$$

c) Cost Alignment (C_{align}): The cost alignment is introduced to encourage stable actions by the agent, penalizing unnecessary changes to the system state. This penalty incentivizes the agent to maintain the current state when it is optimal, reducing oscillations.

$$C_{\text{align}} = -0.6 \text{if action } != 0$$

By incorporating normalized performance measures and nonlinear cost functions, over multiple training episodes, the agent learns to balance response time targets and resource efficiency, scaling up under heavy load to prevent latency penalties, and scaling down when the load decreases to reduce costs. The table below I summarizes all the parameters introduced with their values populated in the simulation:

Symbol	Description	Value
$T_{\max}$	Maximum response time	0.7 seconds
$T_{\min}$	Minimum response time	0.2 seconds
$N_{\max}$	Maximum number of replicas	13 replicas
μ	Service rate per replica	0.8 requests/second
γ	Resource cost scaling parameter	2.5
β	Exponential penalty parameter	1.3
C_{align}	Action alignment penalty	-0.6 if action $\neq 0$, otherwise 0
α	Penalty factor for exceeding $T_{\max}$	2.3
$r_{\min}$	Minimum number of replicas	1
$r_{\max}$	Maximum number of replicas	13

TABLE I
PARAMETERS AND THEIR VALUES USED IN THE DQN AUTOSCALER AND SIMULATION ENVIRONMENT.

C. LSTM Workload Predictor

To enable the agent to anticipate future changes in request rates, we integrate a Long Short-Term Memory (LSTM)-based neural network predictor. LSTMs are a class of recurrent neural networks (RNNs) designed to capture both short-term and long-term dependencies in sequential data, making them highly suitable for time-series forecasting [11]. In the context of our system, the LSTM predictor leverages sequences of historical request rates to forecast the next request rate, enabling The architecture consists of two LSTM layers with 64 hidden units each and a dropout rate of 0.1 to prevent overfitting. These are followed by a fully connected linear layer that predicts the change (Δ) in the request rate. The prediction for the next request rate is computed as:

$$\hat{r}_{t+1} = r_t + \Delta r_t,$$

where r_t is the most recent observed request rate, and Δr_t is the predicted change output by the model.
The model output is blended with a simple linear trend calculated from recent observations, at a fixed ratio (e.g., 70% model prediction and 30% trend). This fixed-weight blending provides a balance between learned patterns and short-term directional changes. The predictor is trained incrementally using Mean Squared Error (MSE) between predicted and observed values:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{r}_{t+1}^{(i)} - r_{t+1}^{(i)}\right)^2,$$

where N is the batch size, and $\hat{r}_{t+1}^{(i)}$ and $r_{t+1}^{(i)}$ are the predicted and actual request rates for the i-th sequence in the batch.

IV. Architecture

Having detailed the DQN and LSTM components, we now outline the integration of these modules within our overall system. We present a unified architecture that applies to both our simulation and the real-world Kubernetes deployment, emphasizing their distinct characteristics.

A. System Architecture

Figure 4 illustrates the main workflow that drives both the simulation and deployment setups. At each discrete time step :

1) Input Request Rate: The input traffic rate is fed into both the Simulated Service and the LSTM Predictor.
2) Simulated Service: Processes the request rate and generates key performance metrics, such as CPU utilization, response time, and the current number of replicas.
3) LSTM Predictor: Analyzes historical request rates to forecast the next request rate, helping the system anticipate workload changes.
4) DQN Agent: Consumes both the performance metrics from the Simulated Service and the predicted request rate from the LSTM Predictor to determine the optimal scaling action.
5) Scaling Action: Executes the scaling decision by adjusting the number of replicas in the Simulated Service and he process iterates until the predefined workload patterns are completed.

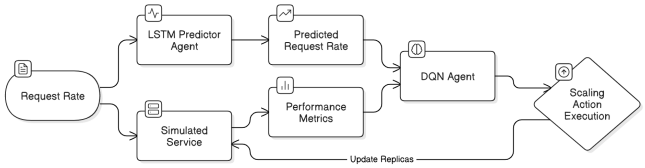

Fig. 1. The system architecture illustrating the process at each step, from input request rate processing to decision-making by the DQN agent.

The following sections provide a detailed explanation of each component and its role in the workflow.

B. Simulation Architecture

The simulation environment is designed to mimic the behavior of a microservice under varying workloads. It is used to train and benchmark three different autoscaling approaches and is responsible for generating the workload for every step, creating sinusoidal patterns, and simulates service metrics.

1) Workload Simulation: The workload generator creates realistic and dynamic request patterns, emulating production environments. The total arrival rate $\lambda(t)$ (in requests per minute) is modeled as a combination of multiple components to capture both periodic and irregular traffic fluctuations.
The main wave component is a sinusoidal function [7] that captures the periodic nature of traffic variations. It is defined as

$$\lambda_{\text{main}}(t) = \lambda_{\text{base}} + A_p \sin(\omega_b t + \phi_b),$$

Where λ_{base} represents the baseline arrival rate, A_p is the amplitude of the main wave, $\omega_b = \frac{2\pi}{T_c}$ is the angular frequency determined by the cycle period T_c and ϕ_b is a random phase shift added to introduce variability. In addition to the main wave, a secondary waves are included to simulate asymmetry and irregularities in traffic. These are summed together with the main wave, resulting in a more dynamic workload pattern:

$$\lambda_{\text{secondary}}(t) = A_s \sin(\omega_s t + \phi_s)$$

The final arrival rate $\lambda(t)$ is computed by combining the two components:

$$\lambda_(t) = \lambda_{\text{main}}(t) + \lambda_{\text{secondary}}(t)$$

to maintain the constraints configured, the arrival rate is then clipped between a minimum and maximum known bound:

$$\lambda_(t) = max\{\lambda_m, min\{\lambda(t), \lambda_M\}\}$$

It follows a summary of the variable definitions used in our simulation:

Symbol	Description	Value
λ_{base}	Baseline arrival rate	80 requests/minute
A_p	Amplitude of the main wave	160 requests/minute
T_c	Period of the main wave	1200 seconds (20 minutes)
ϕ_b	Phase shift of the main wave	Uniform random $[-0.2, 0.2]$ radians
$A_{s,i}$	Amplitude of the i-th secondary wave	Uniform random $[3.0, 9.0]$ requests/minute
f_i	Frequency of the i-th secondary wave	Uniform random $[0.1, 0.3]$ Hz
$\phi_{s,i}$	Phase of the i-th secondary wave	Uniform random $[0, 2\pi]$ radians
λ_{min}	Minimum arrival rate	6 requests/minute
λ_{max}	Maximum arrival rate	300 requests/minute

TABLE II
SUMMARY OF THE KEY PARAMETERS AND VALUES USED IN THE WORKLOAD SIMULATION MODEL.

2) Response Time Model: The response time T models the microservice's performance under load, derived from the principles of the $M/M/c$ queueing system [6]. In this context, the conditional waiting time in the $M/M/c$ queue with unlimited buffer is approximated using an Erlang distribution [10], which allows for a probabilistic calculation of waiting times based on the system's configuration.

In this model, the Erlang distribution characterizes response time using two key parameters: the rate parameter, $n\mu$, which represents the total service rate where n is the number of servers and μ is the service rate per server; and the shape parameter, k, which corresponds to the effective number of stages in the system, representing the requests contributing to the waiting time. Using these parameters, the mean response time is computed as $E[W] = \frac{k}{n\mu}$, while the variance is given by $\text{Var}(W) = \frac{k}{(n\mu)^2}$. Table III summarizes the parameter values used.

Symbol	Description	Value
$T_{\min}$	Minimum achievable response time	0.2 seconds
T	Average response time	Computed via the model
c	Number of active replicas	Determined by scaling policies
μ	Service rate of a single replica	0.8 requests/second
ρ	Utilization factor	$\lambda/(c\mu)$
$c\mu - \lambda$	Remaining system capacity	Positive when system is stable

TABLE III
OVERVIEW OF THE KEY PARAMETERS AND THEIR CORRESPONDING VALUES USED IN THE RESPONSE TIME MODEL BASED ON THE $M/M/c$ QUEUEING THEORY.

C. Deployment Architecture

After completing the training of the DQN and LSTM models, we deploy them in a real-world environment using Google Kubernetes Engine (GKE). Figure 2 presents the architecture of the production-level cluster, showcasing the interaction between the custom autoscaler and various components:

1) The workload generator (a Kubernetes Job) calculates the current request rate, following the same behavior as in the simulation environment. It then sends requests to the deployed service through a Load Balancer.
2) The service processes the requests and updates its metrics. These metrics—such as request rate, CPU utilization, and response time—are exposed to the Prometheus add-on integrated into the GCP project.
3) The pretrained autoscaler retrieves the metrics every 15 seconds from the service and Prometheus. If a scaling action (e.g., increasing or decreasing replicas) is necessary, the autoscaler communicates with the Kubernetes Control Plane, which subsequently executes the scaling operation by adjusting the number of replicas across the worker nodes.

1) Setting Up the Cloud Environment: The deployment begins with setting up a dedicated GCP project and provisioning a GKE Autopilot cluster within a unique namespace. The GKE Autopilot mode abstracts away node-level management, simplifying the setup. However, it introduces restrictions on capacity provisioning, particularly when allocating new nodes [3]. The cluster consists of a maximum of 10 virtual machine instances (e2-medium) provided by Google Cloud Compute Engine, each provisioned with 1 vCPU and 256 MiB of RAM.

2) Deploying the Microservice and pod monitoring: A CPU-intensive microservice was developed using FastAPI, performing computationally expensive tasks with adjustable workload. Metrics essential for autoscaling were exposed through a Prometheus endpoint to facilitate monitoring and decision-making.

3) Mimicking the Synthetic Workloads: To bridge the gap between simulation and deployment, synthetic workloads mimicking those used in training were reproduced. A Python-based workload generator script replicated the workload patterns from the simulation environment. To avoid network latency associated with running the script externally, it was packaged as a Kubernetes Job and deployed within the same cluster as the microservice. Each Job ran approximately three hours, covering six different workload patterns, and providing outputs such as SLO violation percentages and average response times.

4) Integrating the Autoscaler: With the microservice and workload generator operational, the autoscaler was deployed to manage scaling dynamically. The workflow of the autoscaler is as follows:

1) Every 15 seconds, the autoscaler queries the Kubernetes Metrics API and Prometheus endpoints to retrieve the CPU usage across all the replicas, the request rate and the latency.
2) The LSTM Predictor forecasts the request rate for the next 15 seconds.
3) The trained DQN Agent consumes the current metrics and predicted request rate to decide one of the following actions:
   - Scale Up (increase replicas),
   - Scale Down (decrease replicas),
   - No Change.
4) If a scaling action is needed, the autoscaler contacts the Kubernetes Control Plane to update the number of replicas in the Deployment.

In contrast to the simulation environment, where scaling changes take effect immediately, the real-world deployment introduces additional latency. Provisioning a new Pod, or adding new nodes, can take several minutes, adding complexity to maintaining performance targets and resource efficiency.

To mitigate these challenges, we adopted several operational best practices recommended by GKE [4]. This included setting appropriate liveness probes, utilizing graceful termination periods to ensure stable transitions during scaling events, and leveraging other configuration optimizations to reduce the impact of node spin-up delays and transient performance fluctuations.

5) Monitoring and Dashboard: To analyze the autoscaler's behavior in real-time, GKE's native monitoring tools were utilized. By enabling the Prometheus add-on, custom metrics from the microservice were scraped and integrated into GCP's Monitoring and Logging services. A custom dashboard was created to visualize key metrics, including:

1) Request Rate Over Time: A line chart showing the incoming request rate throughout the experiment. This allowed us to observe workload intensity and compare it with scaling decisions.
2) Number of Replicas Over Time: A chart tracking the number of active replicas serving the workload. By superimposing this data on the request rate, we could

[3]GKE Autopilot Documentation, accessed December 20, 2024. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview.

[4]Kubernetes Documentation, accessed January 10, 2025. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/.

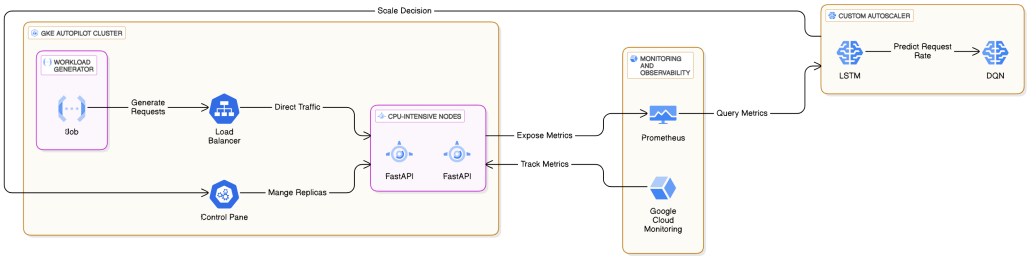

Fig. 2. Google Cloud architecture diagram illustrating the deployment of the custom autoscaler.

determine if the autoscaler reacted effectively to spikes and drops.

3) Response Time Over Time: A latency-focused plot illustrating how response times evolved in relation to shifting workloads and scaling actions. While the plot was built using a PromQL query and lacked precise granularity, it clearly showed the response time trends.
4) Summary Widget: A widget summarizing key performance indicators (KPIs) over the selected time range, including:
   - Average response latency,
   - Total SLO violations.

Since the GCP dashboard cannot be directly exported as a PDF, the metrics were extracted as CSV files, and similar visualizations were recreated using Matplotlib for reporting purposes.

V. Experimental Results

We evaluated three autoscaling approaches under controlled conditions in both a simulated environment and a real-world GKE setup:

1) DQN without Predictor: A pretrained DQN agent without workload forecasting capability (representing the current literature as previously noted).
2) DQN with Predictor: A pretrained DQN agent using an LSTM-based predictor to forecast the next 15 seconds of request rates.
3) HPA (Horizontal Pod Autoscaler): The standard Kubernetes HPA algorithm configured with two CPU utilization thresholds, 60% and 65%. These thresholds were selected to make the HPA more reactive, given the dynamic workload in the environment.

The results from both the simulation and deployment environments are presented below.

A. Simulation Results

The agents were trained using the workload simulator, using the same seed across 500 unique patterns, each representing several days of simulated traffic. Each pattern consisted of 166 steps, with each step representing a 15-second interval. Effectively, this means each sinusoidal pattern spans approximately 40 minutes. Training the agents took approximately thirty minutes, enabling them to learn optimal scaling policies within the simulated environment.

To assess performance, the agents were evaluated on 1,000 new workload patterns. The evaluation measured response times, Service Level Objective (SLO) compliance, and resource efficiency. The results for the DQN with Predictor agent are illustrated in Figure 3, and a summary table comparing all approaches is provided below.

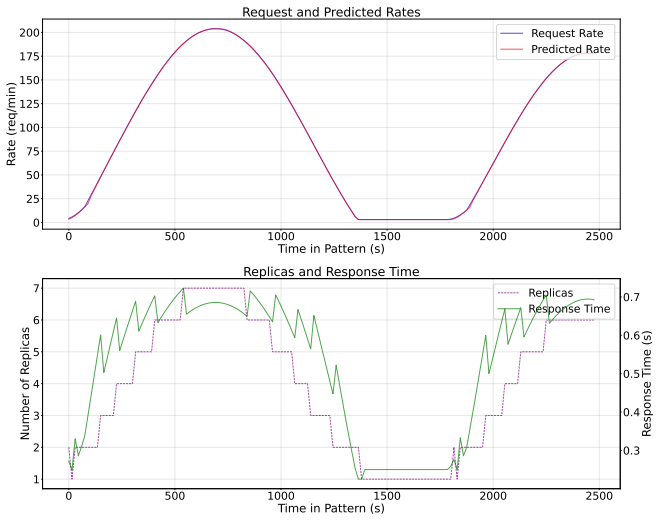

Fig. 3. Simulation Plot: The top plot (magenta) shows the sinusoidal request rate over time. In the bottom plot, the stepped purple line indicates the autoscaler's replica count, while the green line tracks the average response time, demonstrating how proactive scaling maintains low latency.

Architecture	Avg Response Time (s)	SLO Violations (%)	Avg Replicas
DQN with Predictor	0.501	0.1	4.32
DQN without Predictor	0.518	0.2	4.26
HPA (60% Threshold)	0.525	0.5	4.26
HPA (65% Threshold)	0.550	3.2	3.96

TABLE IV
Performance comparison of different autoscaling approaches in simulation environment. Showing the average response time, SLO violations, and resource usage.

a) Key Observations:

1) The DQN with Predictor achieved the best overall results, achieving the lowest average response time (0.501

seconds) and the fewest SLO violations (0.1% with only 70 violations), while using an average of 4.34 replicas, a count close to the other approaches. From the evaluation plot in Figure 3, different from the others (which can be consulted in the project repository on github [5], the number of replicas deployed (bottom plot) anticipates the request rate (top plot), so when it reaches its highest point, the number of replicas is already deployed to serve the response under the Tmax.

2) The DQN without Predictor is in second place, with a total of 289 violations with fewer replicas, this means that followed the request trends well but couldn't anticipate well all other workloads.
3) The HPA with 60% threshold showed acceptable response times (0.525 seconds) but a higher SLO violation rate of 0.5%, matching the average number of replicas used by the DQN without Predictor, showing its limit in being static and understanding our requirements. By increasing the threshold (to 65%) we naturally get fewer replicas but an exponential increase in SLO violations, indicating that in a real environment, a team should spend time in understanding which is the best threshold for their service, assuming that service demand remains constant over time.

B. Deployment Results

To validate these findings in a real-world environment, we tested the autoscaling approaches over a three-hour period on GKE. The workload included six distinct patterns, with at least one exceeding 4 requests per second. The results are summarized in Table V.

Architecture	Avg Response Time (s)	SLO Violations (%)	Avg Replicas
DQN with Predictor	0.32	0.13	3.72
DQN without Predictor	0.35	0.30	3.71
HPA (65% Threshold)	0.34	0.36	4.17

TABLE V
PERFORMANCE COMPARISON OF DIFFERENT AUTOSCALING APPROACHES IN THE GKE CLUSTER. SHOWING THE AVERAGE RESPONSE TIME, SLO VIOLATIONS, AND RESOURCE USAGE.

The results pictured in the table closely resemble those we obtained in the simulation. Overall, the predictive agent outperformed the others, achieving the lowest SLO violations. The predictive agent achieved this result using nearly the same number of replicas as the non-predictive agent, which incurred twice as many violations. Testing the HPA at a 65% threshold was sufficient to observe its performance. as we can see it has a greater number of violations using more replicas, so lowering or increasing the threshold would not be beneficial.
Using GCP monitoring tools, a dashboard was prepared of the three options:

The agent with the predictor does not perfectly follow the sudden changes of the patterns (in blue), instead, it adopts a more conservative approach by leveraging learned trends. In this way, the agent is sure to always allocating the correct number of replicas to handle the current and the next step.

[5]Alfredo Lipari, "Project Custom Autoscaler Repository," Available: https://github.com/alfredoLipari/CustomAutoscaler-k8s.

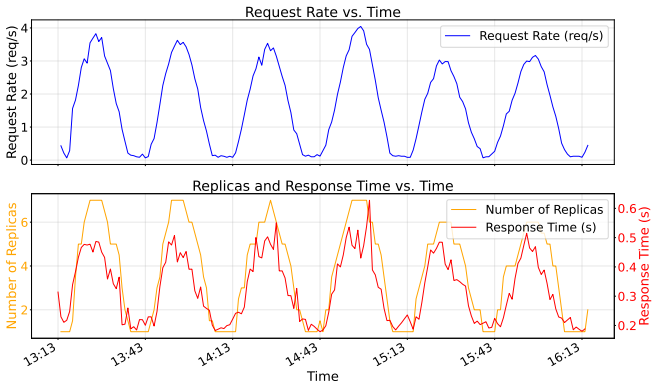

Fig. 4. DQN with predictor plot: The top plot illustrates the request rate (in blue) over time. The bottom plot displays the number of replicas (in orange) on the left y-axis and the response time (in red) on the right y-axis.

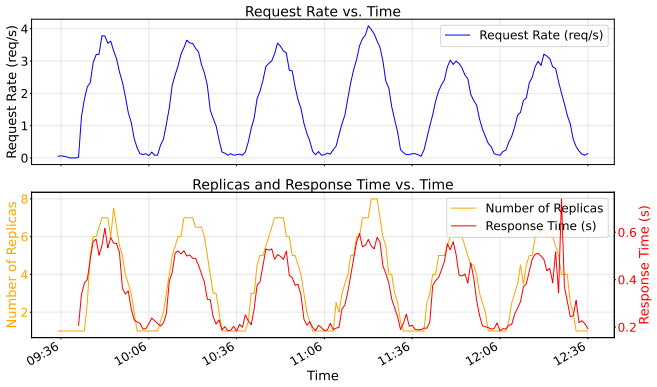

Fig. 5. DQN without predictor plot: The top plot illustrates the request rate (in blue) over time. The bottom plot displays the number of replicas (in orange) on the left y-axis and the response time (in red) on the right y-axis.

In contrast, the agent without prediction (Figure 5) attempts to follow request rate patterns directly. The sudden replica changes sometimes worsened performance, particularly for higher request rates where it couldn't anticipate secondary increases.

The HPA (Figure 6) performed worst despite using more replicas due to its slow response. This latency is evident as the replica count plot forms a rectangular shape rather than aligning with request rate patterns.

C. Design Choices and Their Impact

Several key design decisions influenced the autoscaler's performance characteristics. Our experimental evaluation demonstrated the efficacy of these choices:

- Constrained Action Space: The autoscaler scales by only one replica at a time to provide fine-grained control and prevent resource oscillation. This approach enhanced

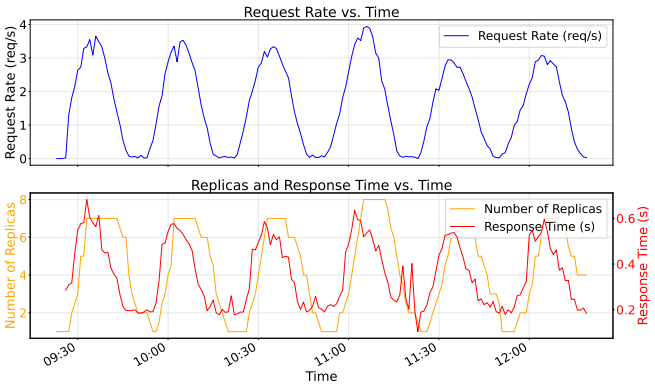

Fig. 6. HPA 65% plot: The top plot illustrates the request rate (in blue) over time. The bottom plot displays the number of replicas (in orange) on the left y-axis and the response time (in red) on the right y-axis.

training stability and performance compared to more aggressive scaling alternatives in our testing, while still effectively handling workload changes through prediction-driven proactive scaling.

- 15-Second Metrics Interval: This interval balances responsiveness and stability, matches Kubernetes' default HPA interval, aligns with typical pod startup latencies, and provides sufficient time for metrics to stabilize. Our experiments confirmed this as an optimal choice compared to shorter intervals (excessive scaling from transient spikes) and longer ones (delayed responses).

D. Limitations and Considerations

While our approach demonstrates advantages over traditional autoscaling, several limitations should be noted:

- Simulation vs. Reality: Our simulation doesn't fully capture real-world factors like network latency, container startup times, and hardware variability. Though our unified architecture minimizes simulation-reality gaps, some discrepancies may affect performance.
- Prediction Reliability: When the LSTM predictor struggles with unpredictable traffic patterns, a blending mechanism (70% model prediction, 30% trend extrapolation) should provide some stability.
- Operational Overhead: Our autoscaler operates as an external service that interacts with the Kubernetes API, introducing minimal overhead to the cluster. The main cost comes with the time to train the autoscaler
- Service Independence: The current system scales services independently without considering interdependencies, a limitation in complex microservice architectures that could be addressed through multi-agent reinforcement learning.

These considerations inform our future research directions, which include enhancing prediction robustness for more variable workloads and developing cross-service coordination mechanisms.

VI. Conclusion

In this work, a new autoscaling approach was presented that utilizes Reinforcement Learning combined with an LSTM-based predictor to surpass the limitations of traditional reactive Kubernetes autoscaler. The study found that despite certain limitations, a simulation environment can effectively train the agent on various workload patterns. The extensive experiments show that the proposed solution in a real cluster, effectively predicts the workload in the near future and understands how to anticipate traffic surges instead of simply reacting to them, which improved performance and optimize resource usage. As a consequence, our autoscaler contributes to sustainability goals by reducing resource overprovisioning, lowering energy consumption.

Future work will focus on enhancing prediction accuracy and assessing the benefits of forecasting further into the future. Future research could expand this work by deploying the custom autoscaler in large-scale applications with multiple interactive services and diverse performance constraints, further analyzing its behavior.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU and partially by project no. 202277 WMAE CUP B53D23012820006, "EdgeVision against Varroa (EV2): Edge computing in defence of bees" funded by the Italian's MUR PRIN 2022 (ERC PE6) research program and by the European Union – Next Generation EU.

References

[1] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-Effective Scaling for Microservice Applications in the Cloud With an Online Learning Approach," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1100–1116, Apr. 2022, doi: 10.1109/TCC.2020.2985352.

[2] A. Abdel Khaleq and I. Ra, "Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications," *IEEE Access*, vol. 9, pp. 35463–35476, 2021.

[3] O. P. Egbuna, "Machine Learning Applications in Kubernetes for Autonomous Container Management," *Journal of Cloud Computing*, vol. 10, no. 1, p. 12, 2021.

[4] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Dynamic Multi-Metric Thresholds for Scaling Applications Using Reinforcement Learning," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1807–1821, Apr.–Jun. 2023, doi: 10.1109/TCC.2022.3163357.

[5] Z. Wang, "A Model-Based RL Approach to Container Scaling in Kubernetes," *Proc. of the ACM Symposium on Cloud Computing*, 2023.

[6] P. Harrison and N. M. Patel, *Performance Modelling of Communication Networks and Computer Architectures*. Addison–Wesley, 1992, p. 173.

[7] A. V. Oppenheim and A. Schafer, *Discrete-Time Signal Processing*. Prentice-Hall, 1999.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[9] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2016.

[10] W.-C. Chan and Y.-B. Lin, "Waiting time distribution for the M/M/m queue," *IEE Proceedings - Communications*, vol. 150, no. 3, pp. 163-167, 2003

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.