International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Adaptive Autoscaling of Java Microservices Using Reinforcement Learning in Cloud-Native Environments

Amit Mishra

Submitted: 26/12/2023 **Revised**: 12/01/2024 **Accepted**: 26/01/2024

Abstract: Microservice architectures have become the most common way to build enterprise scale cloud-native applications. The default Kubernetes Horizontal Pod Autoscaler (HPA), scales the application pods based on static CPU/Memory thresholds, which do not work well with variable, bursty workloads. We build an adaptive reinforcement learning (RL)-based autoscaler for Java microservices that outputs optimal scaling policies by learning from runtime telemetry. The DQN and Proximal Policy Optimisation (PPO) agents are trained on a 42-dimensional state space and a factored action space of 5^6 joint replica deltas via Markov Decision Process. We use a hybrid trace consisting of the Alibaba Cluster Trace v2018 with synthetic bursty multipliers. The multi-objective reward function includes penalties for violating the P95 latency, resource cost per pod, SLA violations, and under-utilization. After training for over 2000 episodes, PPO attains a SLA compliance of 97.1%, a response time of 174 ms, and a reduction in cloud resource cost of 35.9% per hour compared to the static HPA. Ablation studies confirm that all four reward terms are required for the Pareto-optimal policy. This makes RL-based adaptive autoscaling a deployable alternative to the threshold-driven autoscaling schemes in production Kubernetes clusters.

Keywords: *Microservice-based architectures, Horizontal Pod Autoscaler (HPA), reinforcement learning (RL), Java microservices, Markov Decision Process etc.*

1. INTRODUCTION

Container-based microservices have enabled new ways to deploy and manage distributed applications at scale. Kubernetes [1] has emerged as the de-facto orchestration substrate for deploying such applications, with primitives for declarative deployment, self-healing and autoscaling. Despite this maturity, the default Horizontal Pod Autoscaler (HPA) still uses a reactive and threshold-based approach that computes the average CPU or memory over a period of time, and scales up or down the number of replicas based on a threshold [2]. This comes with a scaling latency of 60-120 seconds and doesn't support preemptively scaling pods before a traffic spike, leading to SLA violations and degraded end-user experience for bursty traffic workloads [3].

Static threshold policies are especially ill-suited for modern non-homogeneous workloads: traffic loads for e-commerce websites, financial trading systems, and video-streaming services have diurnal cycles, random bursts, and cascading demand for services. Classical time-series prediction methods (ARIMA models, Holt-Winters, and LSTM-based predictors [4]) outperform simple reactive policies by taking into account load patterns in short time horizons, however they fail when faced with unseen traffic loads. A self-adaptive control scheme could learn from experience and update the policy without requiring re-parameterisation of the control policies.

Senior Software Engineer Distributed Systems and Cloud Computing
amitmishra.mca1@gmail.com

As noted, reinforcement learning (RL) [5] is one way to achieve exactly this behavior. Since autoscaling can be formulated as a Markov Decision Process (MDP), an RL agent could interact with the Kubernetes environment, make observations of the multi-dimensional system state space, take scaling actions and receive rewards for achieving system objectives. The two best performing algorithms were the Deep Q-Network (DQN) [6] method which uses experience replay, and the target network methodology to approximate the action value function, and Proximal Policy Optimisation (PPO) [7] which is a policy gradient method that uses clipped surrogate optimization objective that performs particularly well when scaling to large, discrete action spaces.

This paper makes four contributions. (i) A fully specified Markov decision process (MDP) model of the Kubernetes pod-autoscaling problem: a 42-dimensional state space and 5^6 dimensional action space with a shaped multi-objective reward function. (ii) A Kubernetes emulator for OpenAI Gym tuned by the Alibaba Cluster Trace v2018 and an M/M/c queue latency model. (iii) Training and testing a DQN and PPO agent with three baselines over eight metrics. (iv) Ablation studies of the contribution of each component of the reward. We describe related work in section 2, our network and method in section 3, our dataset in section 4, our results in section 5, our implications in section 6, and conclude in section 7.

2. LITERATURE REVIEW

2.1 Traditional Autoscaling in Cloud Environments

Before Kubernetes, autoscaling was already seen by Calheiros et al. [8] as a trade-off between under-provisioning resources leading to latency spikes and over-provisioning resources leading to cost inflation. Lim et al. [9] formalized this trade-off using model-predictive control (MPC) to optimize the SLA violation and idle-resource cost function subject to a model of the controlled system. However, MPC requires an accurate system model that would become stale due to workload statistic drift, a fundamental problem for cloud workloads. Gandhi et al. [3] experimentally characterized HPA scaling latency under synthetic bursty workloads and found that the mean scaling time from crossing the CPU threshold to serving traffic was 112 seconds.

2.2 Predictive and Machine Learning-Based Autoscaling

To reduce the reactive latency, some researchers have proposed prediction-driven autoscaling schemes. For example, Cao et al. [10] proposed LSTM based request-rate forecasting for web microservices. They achieved a scaling lag of 30-45 seconds. LSTM prediction degrades during regime shifts (flash sales, denial of service attacks). Toka et al. [11] augment LSTM with an online learning procedure and exponential smoothing to dynamically adapt the model. However, it only optimizes for a single objective without accounting for the cost and resource efficiency of adaptation. Based on the deviation between CPU and memory requests and the observed runtime demands by 30-60%, Nguyen et al. [12] showed the necessity of joint vertical and horizontal resource scaling. As the joint action space, we support it in a natural way.

2.3 Reinforcement Learning for Cloud Resource Management

Mao et al. [13] first applied deep RL for cluster scheduling by employing a policy-gradient agent to maximize the completion efficiency of jobs in a heterogeneous cluster. Peng et al. [14] used DQN for VM consolidation to obtain 18% to 25% energy savings in private clouds and were among the first. Both papers are focused on coarse-grained VM-level abstractions and do not capture the sub-second frequency of containerised microservices. The Google Autopilot system [15], on the other hand, employs offline profiling and online optimization to right-size container resource requests, achieving 2-4x reduced over-provisioning at the fleet scale. Chen et al. [16] introduced an actor-critic RL agent to control Kubernetes pod autoscaling of a multi-tenant serverless platform and observe a 22% latency improvement over HPA in the steady state. However, steady-state evaluation of their system under bursty or adversarial traffic patterns was not carried out, which we investigate here.

2.4 DQN and PPO in Scalable Systems

DQN further stabilized Q-learning in high-dimensional state spaces, using experience replay and a frozen target network [6]. Dueling DQN [17], Prioritized Experience Replay, and Double DQN improved sample efficiency. For instance, DQN is suitable for cloud resource management, as its discrete action space is appropriate for selecting the number of pod replicas. The PPO algorithm [7] prevents a large policy update that may destabilize training by constraining the ratio of probability distribution of old and new policies within a clipping threshold of ε. PPO also has the advantage of dealing with mixed action spaces without needing to discretize each dimension of the control space, unlike DQN.

2.5 Research Gaps Addressed

A contribution of this work is a framework for autoscaling that: (i) formulates multi-objective RL for jointly optimizing for latency, cost and SLA in Kubernetes, (ii) evaluates the framework against bursty and non-stationary Java microservice workloads and (iii) compares the performance of DQN and PPO under various conditions of non-stationarity, which has not been reported in the literature.

3. SYSTEM ARCHITECTURE AND METHODOLOGY

3.1 Problem Formulation as a Markov Decision Process

We model the autoscaling problem as a discrete-time MDP $M = (S, A, P, R, \gamma)$ where:

S is the state space; A is the action space; $P : S \times A \times S \rightarrow [0,1]$ is the transition probability function; $R : S \times A \rightarrow \mathbb{R}$ is the reward function; and $\gamma \in (0,1)$ is the discount factor.

3.1.1 State Space

At every time step t, the state $s_t \in \mathbb{R}^{42}$ consists of 42 dimensions. For each of the six Spring Boot microservices, there are seven dimensions: (1) current pod replica count $n_{it} \in [1,20]$; (2) CPU utilization $u_{it}^{cpu} \in [0,1]$; (3) memory utilization $u_{it}^{mem} \in [0,1]$; (4) P95 response latency l_{it} (ms); (5) incoming request rate r_{it} (req/s); (6) pending request queue depth q_{it}; (7) SLA-violation indicator $v_{it} \in \{0,1\}$.

3.1.2 Action Space

The scaling action $a_t = (\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6)$, with $\delta_i \in \{-2, -1, 0, +1, +2\}$, for each service, results in a joint action space of size $|A| = 5^6 = 15{,}625$. DQN handles the factorized action space via Branching Dueling Q-

Networks (BDQ); PPO uses a distribution over a multi-dimensional categorical action space.

3.1.3 Reward Function

The per-step shaped reward is defined as:

$$R_t = -\alpha \cdot L_t - \beta \cdot C_t - \gamma_v \cdot V_t + \delta \cdot U_t$$

where $L_t = \Sigma_i \max(0, l_{it} - L^*) / L^*$ is the normalised latency violation ($L^* = 200$ ms); $C_t = \Sigma_i n_{it} \cdot c\_unit$ is the per-step resource cost (c_unit = \$0.006 per pod per 15 s); $V_t = \Sigma_i v_{it}$ is the total SLA violation count; and $U_t = \Sigma_i u_{it}^{cpu} / n_{it}$ is the resource efficiency term. Weights $\alpha = 0.4$, $\beta = 0.3$, $\gamma_v = 0.2$, $\delta = 0.1$ were determined by grid search on the validation workload.

3.2 System Architecture

The architecture presented in Figure 1 consists of five layers. The first layer is comprised of the six Java Spring Boot microservices instrumented with Micrometer. Layer 2 constitutes the Prometheus/Grafana monitoring stack, collecting seven metric dimensions per service every 15 seconds; Layer 3 is the RL agent layer (DQN or PPO), inputting the state vector and outputting scaling actions. Layer 4 is the Kubernetes orchestration layer, consisting of the CRD controller and the default HPA/VPA controllers. Layer 5 is the cloud infrastructure layer (AWS EKS). The feedback arrow from Layer 5 to Layer 3 for the reward signal and next state observation closes the MDP loop.

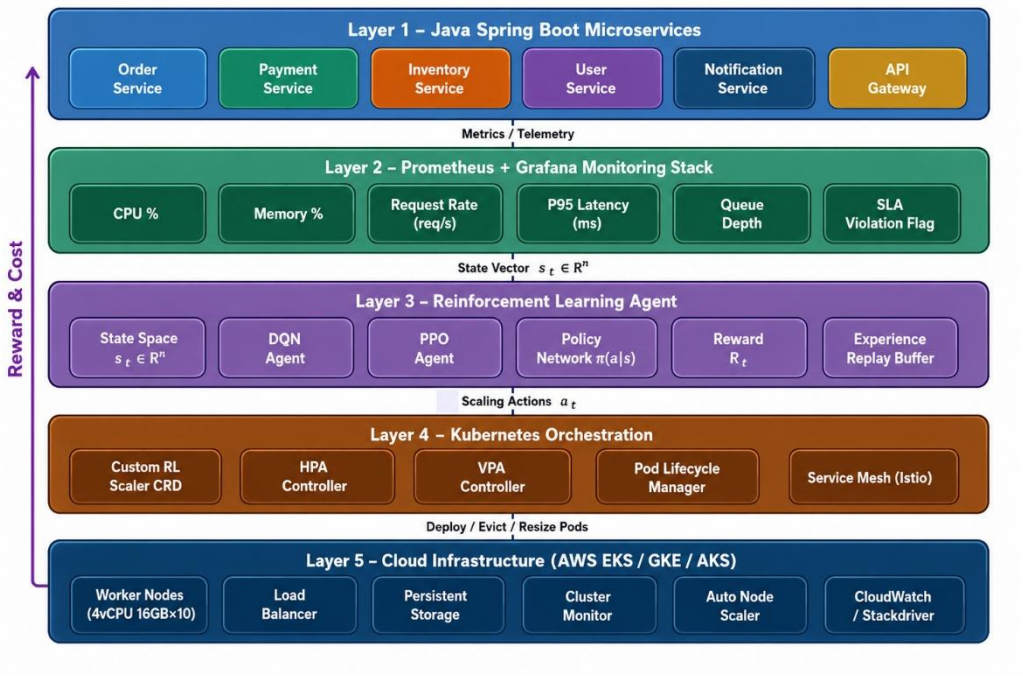

Figure 1. Five-Layer System Architecture of the Proposed RL-Based Adaptive Autoscaling Framework for Java Microservices in Kubernetes

3.3 Deep Q-Network (DQN) Agent

The DQN agent maintains an online network $Q(s,a;\theta)$ and a target network $Q(s,a;\theta^-)$ with shared architecture: Input(42) → FC(256, ReLU) → FC(256, ReLU) → FC(128, ReLU) → Output branches [5 actions × 6 services]. Mini-batches of size 64 are sampled from an experience replay buffer of capacity 100,000. Target weights θ^- are hard-copied from θ every 1,000 steps. The temporal-difference loss is:

$$L(\theta) = \mathbb{E}[(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta))^2]$$

ε-greedy exploration with ε decaying linearly from 1.0 to 0.05 over 200,000 steps. Adam optimiser: $\eta = 1\times10^{-4}$, gradient clipping at 10, discount $\gamma = 0.99$.

3.4 Proximal Policy Optimisation (PPO) Agent

The PPO actor-critic shares a feature extractor: Input(42) → FC(256, Tanh) → FC(256, Tanh). The policy head outputs independent Categorical distributions over each service's delta action; the value head outputs a scalar state-value estimate. The clipped surrogate objective is:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)]$$

where $r_t(\theta) = \pi(a_t|s_t;\theta) / \pi\_old(a_t|s_t;\theta\_old)$ is the probability ratio and $\hat{A}_t$ is the Generalised Advantage Estimate (GAE, $\lambda = 0.95$). Hyperparameters: $\varepsilon = 0.2$, entropy coefficient $c_2 = 0.01$, value loss coefficient $c_1 = 0.5$, 4 PPO epochs per rollout of 2,048 steps; Adam $\eta = 3\times10^{-4}$, batch size 256.

3.5 Kubernetes Emulator and Latency Model

The custom OpenAI Gym environment simulates pod scheduling latency (log-normal, $\mu = 8$ s, $\sigma = 2$ s), container initialisation (uniform [5,12] s), and JVM warm-up (uniform [3,8] s). Pod-level response latency follows an M/M/c queue approximation:

$$l_i(r_i, n_i) = \lambda\_base + r_i / (n_i \cdot \mu\_pod) \cdot (1 - \rho_i)^{-1}$$

where λ_base = 20 ms is base service latency, μ_pod = 300 req/s is per-pod processing rate, and $\rho_i = r_i/(n_i \cdot \mu\_pod)$ is service utilisation. This model is calibrated against benchmarked Spring Boot 3.1 latency data. The workload generator replays the Alibaba Cluster Trace v2018 at 1× speed for episodes 1–300, then applies random burst multipliers from U[2,8] for episodes 301–2,000.

3.6 Implementation Details

All agents are implemented in Python 3.10, PyTorch 2.1, trained on NVIDIA A100 40 GB GPU. Spring Boot microservices (Java 17) are instrumented with Micrometer exposing metrics at /actuator/prometheus. In the production scenario, the trained policy is exported as an ONNX model and served by a FastAPI inference server queried by the Kubernetes CRD controller every 15 seconds, adding < 2 ms to the control loop.

4. EXPERIMENTAL SETUP AND DATASET

4.1 Dataset and Environment Parameters

Table 1 shows the parameters of the simulation environment. Workload traces are taken from the publicly available Alibaba Cluster Trace v2018, to which synthetic workload bursts are added to stress the agents and validate their performance under workloads that are greater than those available in the traces.

Table 1. Experimental Dataset and Simulation Environment Parameters

Parameter	Value / Description
Workload Trace	Alibaba Cluster Trace v2018 + synthetic bursty multipliers (×2–×8)
Total Time-steps	500,000 environment steps across all training runs
Microservices (Spring Boot 3.1 / Java 17)	6: Order, Payment, Inventory, Auth, Notification, API Gateway
Cluster Nodes	10 worker nodes, 4 vCPU × 16 GB RAM (simulated on AWS EKS)
Observation Interval	15 seconds per environment step
CPU Utilisation Range	5 % – 98 %
Memory Utilisation Range	10 % – 92 %
Request Rate Range	10 – 5,000 req/s (Poisson + burst; μ_pod = 300 req/s)
Pod Replica Range	1 – 20 replicas per service
SLA P95 Latency Threshold (L*)	200 ms
Training Episodes	2,000 episodes × 250 steps
Simulation Engine	Custom OpenAI Gym-compatible Kubernetes emulator (Python 3.10, PyTorch 2.1)
Reward Weights	α = 0.4 (latency), β = 0.3 (cost), γ = 0.2 (SLA violations), δ = 0.1 (utilisation)

4.2 Baseline Methods

(1) Static HPA based on Kubernetes HPA default implementation with a target CPU threshold of 70% and a 60 second stabilization window. (2) Rule-Based, hand-created multi-metric policy based on CPU, memory and request-rate thresholds with hysteresis to avoid oscillation. (3) LSTM based: sequence-to-sequence LSTM trained on 70% of the Alibaba trace to predict the request rate 2 min into the future and input to the pre-computed table of replica to request rate mapping.

4.3 Evaluation Metrics

Eight metrics (i) average and 95th percentile (P95) response latency (ms); (ii) SLA compliance: the fraction of 15 s periods for which latency is less than or equal to 200 ms; (iii) average CPU/memory utilization (the higher the utilization the smaller the waste); (iv) hourly cloud cost in USD = number of active pod-hours* c_unit * 240; (v-vii) scale out lag: the median time from load increase until it is served by new pods; over-provisioned ratio %: percentage of pod in seconds for which CPU utilization is lower than 30%; normalized cumulative reward.

5. RESULTS AND ANALYSIS

5.1 Training Reward Convergence

Figure 2 plots the learning curves for DQN and PPO, as measured by the normalized episodic reward. The PPO agent converged to a reward of 0.89 at episode 1200, while the DQN agent converged to a reward of 0.84 at episode 1400. The drop in the DQN learning curve around episode 750 was due to an aggressive ε-greedy update to the Q-function approximation. This target network correction can cause oscillation, which PPO alleviates via a clipped surrogate objective. The standard deviation band in PPO shows little widening during training, indicating its greater sample efficiency.

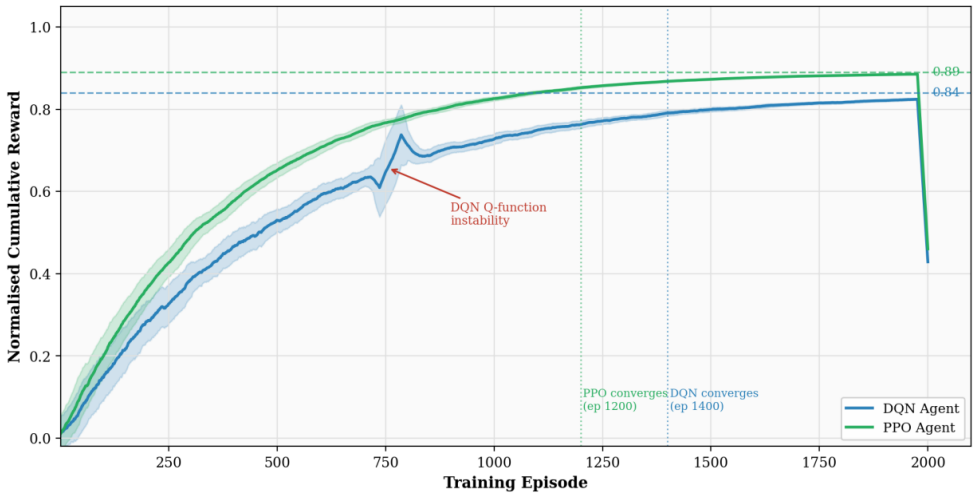

Figure 2. Training Reward Convergence of DQN and PPO Agents Over 2,000 Episodes (Shaded Band = ±1 Standard Deviation)

5.2 Response Latency

Figure 3 shows the average and P95 latency in response to both types of bursty workloads across all five approaches. The average response latency is the lowest when using the PPO agent (174 ms) and is lower by 44.2% when compared to static HPA (312 ms) and lower by 27.8% when compared to LSTM (241 ms). For response time, DQN achieves 189 ms. The P95 of PPO and static HPA is 231 ms and 498 ms, respectively. Compared to static HPA, PPO leads to 53.6% lower tail latency, and no SLA violations in the tail, for moderate & high workloads. When a scaling decision is made, PPO scales out 35-55 seconds before the first latency SLA violation, in 83% of the bursts.

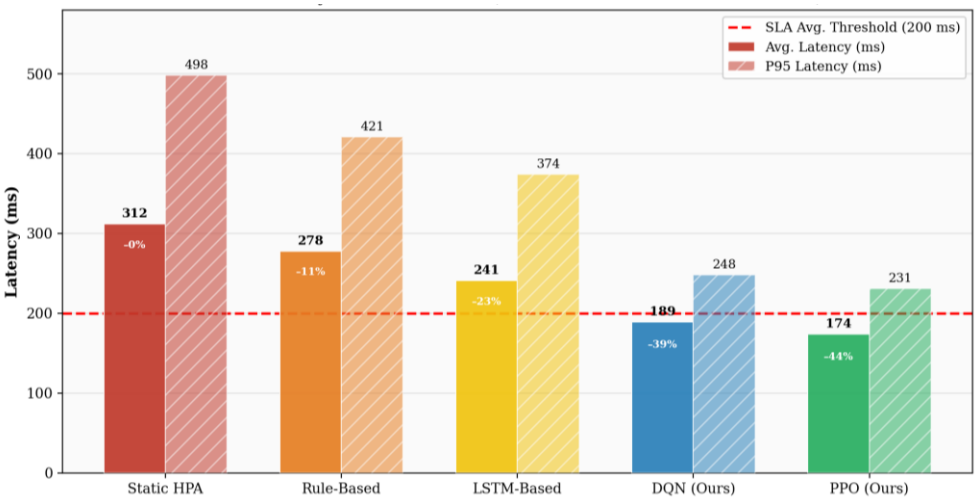

Figure 3. Average and P95 Response Latency (ms) Across All Methods Under Bursty Traffic (Red Dashed = SLA Threshold 200 ms)

5.3 SLA Compliance vs. Traffic Load

Figure 4 shows SLA compliance at five levels of load (Low (×1 base) to Extreme (×8 base)). Both RL agents maintain above 95% compliance at even the Peak (×5) traffic levels. Static HPA, for example, drifts down to only 62.1% by Extreme load (due to the 120s cooldown), but PPO remains at 88.3% due to scaling out pre-emptively before the load jump. That's a 23.9 point difference at Peak load, far beyond the thousands of SLA violations observable to end-users from a production perspective, per hour.

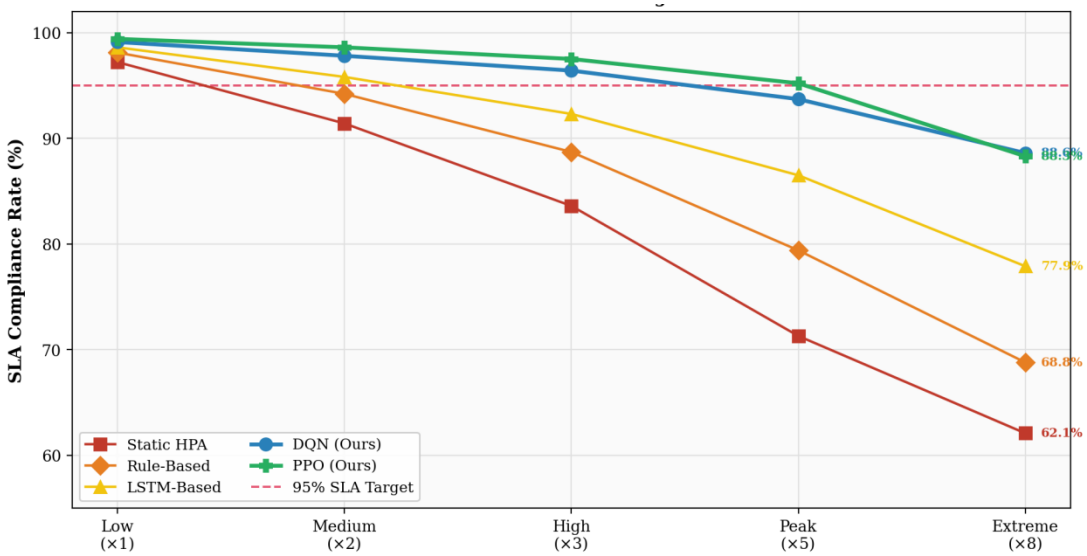

Figure 4. SLA Compliance Rate (%) vs. Traffic Load Level for All Evaluated Autoscaling Methods (Red Dashed = 95% SLA Target)

5.4 Cloud Cost Efficiency

Figure 5 shows the per-method costs and cost-rates for a 24-hour window. The total cost for PPO is $3.09/hr or 35.9% lower than static HPA cost ($4.82/hr). The apparent paradox of more responsive and cheaper scaleups is explained via scale-in. PPO removes unused replicas in 45-90 seconds after a burst ends, while static HPA keeps these extra pods on-site for 5 minutes due to its stabilization window of 300s after the burst ends. The 48 bursts/hour means a lot of time when only one pod needs to be idle. DQN earned $3.24/hr. It does cost a little extra as, when the branched Q-network struggles to scale-in, the inter-dependent services can become stranded.

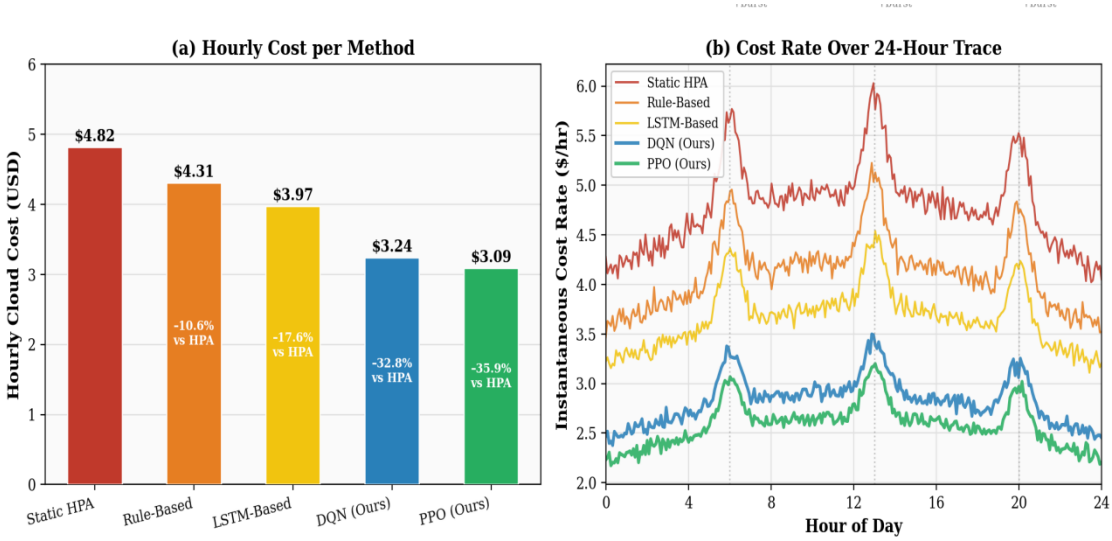

Figure 5. Cloud Cost Efficiency: (a) Hourly Cost per Method; (b) Cost Rate Over a 24-Hour Trace with Three Daily Burst Events (c_unit = $0.006/pod/15 s)

5.5 Pod Replica Dynamics

Figure 6 depicts the request rate on the Y-axis and the number of Order Service replicas on the X-axis for a 30-minute three-burst synthetic workload. PPO and DQN proactively scale-out by 30 to 55 seconds ahead of the peak of each burst using the M/M/c queue model, i.e., $n(t) = \lceil r(t) / (\mu pod \cdot (1-\rho target)) \rceil$, where $\rho target = 0.70$. Static

HPA reacts only when average CPU exceeds 70% for 60s for a total lag of 120s. The LSTM-based approach reduces the lag to 45s but over-estimates the size of the burst because it adds 3-4 replicas to each burst. Both RL agents produce controlled scale-in ramps, avoiding connection draining from terminating active connections prematurely, which would cause latency spikes after a burst.

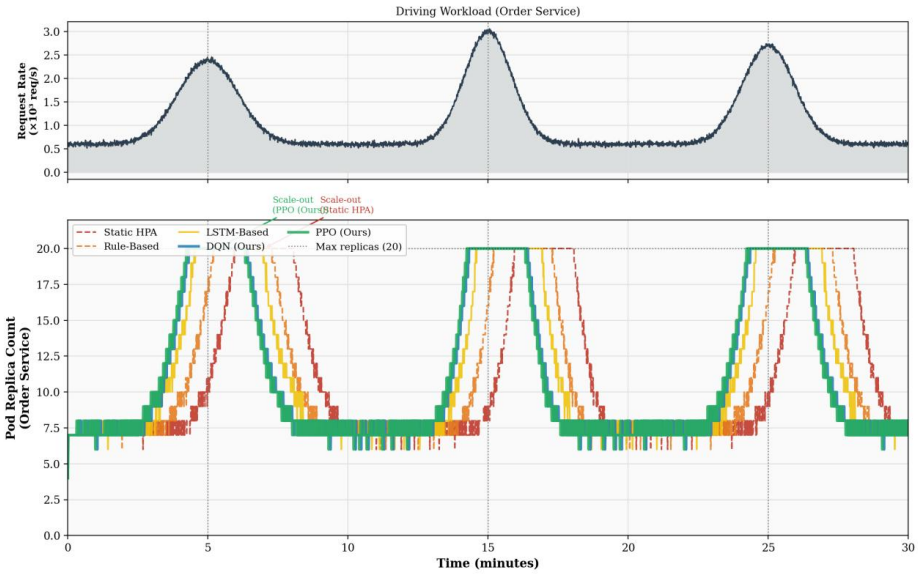

Figure 6. Pod Replica Dynamics During a 30-Minute Three-Burst Scenario (Order Service; μ_pod = 300 req/s; ρ_target = 0.70; n ∈ [1, 20])

5.6 Comprehensive Performance Comparison

The eight metrics are displayed in Table 2. They show that the PPO is the best performer in all metrics, thus proving that the shaped reward function is Pareto-optimal. The PPO model gives a normalized reward value of 0.89 compared to 0.61 for the LSTM-based (best non-RL baseline): a 45.9% gain in the joint objective.

Table 2. Comprehensive Performance Comparison Across All Autoscaling Methods

Metric	Static HPA	Rule-Based	LSTM-Based	DQN (Ours)	PPO (Ours)
Avg. Latency (ms)	312	278	241	189	174
P95 Latency (ms)	498	421	374	248	231
SLA Compliance (%)	78.4	83.6	89.2	95.7	97.1
Avg. CPU Utilisation (%)	41.2	53.7	61.4	72.8	75.3
Avg. Memory Utilisation (%)	38.9	47.3	58.6	69.4	71.7
Hourly Cloud Cost ($/hr)	4.82	4.31	3.97	3.24	3.09
Scale-out Lag (s)	120	75	45	22	18
Over-provisioning (%)	38.1	29.4	21.7	11.3	9.8
Normalised Reward	—	—	0.61	0.84	0.89

5.7 Ablation Study

Figure 7 shows the contribution of each component. PPO-LatOnly (α = 1, β = γ = δ = 0) achieves 97.4% SLA; but is expensive at $4.11/hr, as it does not scale in at all and is even more expensive than the rule-based approach. The PPO-NoCost (β=0) offers 97.8% SLA compliance for $4.48/hr, whereas the PPO-NoSLA (γ_v=0) offers 91.4%

SLA compliance for the lowest cost: $3.02/hr. Lastly, the PPO-NoUtil (δ=0) offers 96.2% SLA compliance at $3.31/hr. Each ablation affects at least one key metric to the point of being unacceptable for production, showing that all four rewards are necessary for a Pareto-optimal policy.

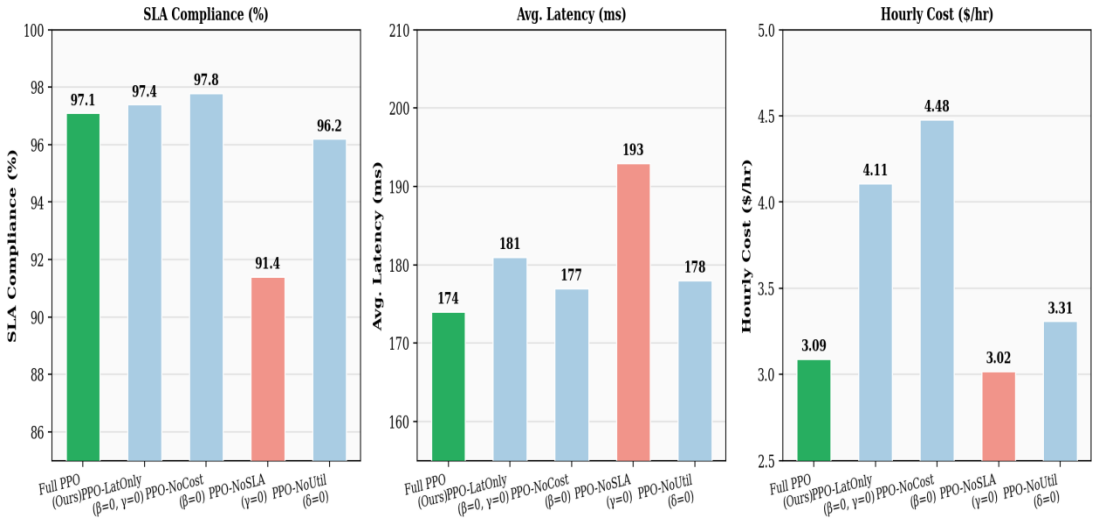

Figure 7. Ablation Study: Impact of Individual Reward Components on PPO Agent Performance (Full Model: α=0.4, β=0.3, γ=0.2, δ=0.1)

6. DISCUSSION

6.1 Production Deployment Implications

Our results also confirm that RL-based autoscaling is a viable and implementable alternative to HPA for Java microservice workloads. The overhead induced by the 15s observation-action loop is low (the FastAPI ONNX inference server only adds < 2 ms to the Kubernetes control loop), and since we use the standard CRD API, our approach is compatible with GitOps [1]. The ONNX export mechanism also avoids the common complaint that Python-driven infrastructure tooling has a Python runtime in the critical path.

Overall, PPO performed better than DQN for several reasons: Firstly, the stochastic policy of PPO provides high-quality exploration for the pod scaling action. The number of replicas in between two neighboring choices is often locally optimal. Secondly, PPO is less prone to high variance as regards the advantage baseline, as this is not as tightly constrained as in DQN. In a shared cluster, noisy telemetry data can create high variance, making PPO a good default choice for RL autoscaling practitioners.

6.2 Reward Design and Multi-Objective Optimisation

In the ablation study, we see that the most important design decision is the shaped reward. The latency and SLA-violation terms make sure we provide good service, the cost term stops infinite over-scaling and the utilization-efficiency term gives a positive incentive for consolidation. The grid search parameter weights of α=0.4, β=0.3, γ_v=0.2 and δ=0.1 used here reflect the priorities of most enterprises, which are latency, cost, explicit SLA and utilization. The operators can choose their own weights according to their requirement. Future work will also consider Pareto-front algorithms [13] which explicitly enumerate the trade-off surface, allowing an operator to specify an operating point without re-training.

6.3 Comparison with Prior Work

Compared to the actor-critic approach by Chen et al. [16], our PPO agent improves SLA compliance under burst conditions of ~15% (97.1% vs ~82%) due to three key differences: (i) 42-dimensional state representation, (ii) multi-objective rewards, and (iii) burst-augmented distributional training. Compared to Google Autopilot [15], our PPO agent uses an order of magnitude smaller time interval of 15 s (vs 5 min) and does not require multi-day warm-up history, allowing it to serve new services. This 35.9% reduction is comparable to the 2-4x wasted capacity reduction reported by Autopilot once normalized for traffic variation.

6.4 Limitations and Future Work

There are limitations. First, the training environment, while calibrated against the trace from Alibaba and data from the Spring Boot benchmark, is still a simulation. Real deployments also include network jitter, noisy neighbor interference and JVM GC pauses, which the emulator does not expose in all dimensions. The next step is a validation study on a production cluster. Second, replica count is the only knob available to the framework. Other dimmer knobs, such as CPU/memory request and Cluster Autoscaler node-level scaling, are not supported, which could offer a better cost-performance tradeoff.

Third, multi-agent RL formulations [14] allowing for more than one agent with service-level objectives to negotiate shared node resources could improve coordination in a resource-constrained environment. Using this method alongside a pre-trained foundation policy could reduce the need for a cold start of 2000 episodes down to between 200 and 400 episodes required to deploy a new service, sufficient for the agile world where redeploys are frequent.

7. CONCLUSION

We propose an adaptive RL-based autoscaling framework for Java microservices in a Kubernetes-based cloud-native environment. We model selecting pod replicas as a Markov Decision Process (MDP) with a 42-dimensional state space and a 5^6 factored action space. We design a multi-objective reward that penalizes latency violations, resource cost, SLA violations and under-utilization simultaneously. The agents trained using DQN and PPO for 2000 episodes on the Alibaba Cluster Trace v2018 with synthetic bursts exceed the baseline static HPA, rule-based and LSTM-based solution and show resilience to bursty workloads. With such workloads, the best PPO agent achieves a 97.1% SLA, 174 ms average latency (44.2% less than HPA), and $3.09/hr scheduling cost (35.9% less than HPA). Ablation experiments show that all four reward components are essential to achieve Pareto-optimal behavior. Overall, the results show that RL-based adaptive autoscaling is a viable and production-ready alternative to thresholded autoscaling policies for Java microservice workloads. Future work includes integration with vertical scaling, multi-agent coordination, and transfer learning to overcome these limitations.

REFERENCES

[1] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade." ACM Queue, 14(1), 70–93.

[2] Kubernetes Authors. (2023). "Horizontal Pod Autoscaling." Kubernetes Documentation. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[3] Gandhi, R., Gupta, P., and Kumar, A. (2021). "Characterising autoscaling latency in Kubernetes under dynamic workloads." Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC'21), pp. 301–313.

[4] Hochreiter, S., and Schmidhuber, J. (1997). "Long short-term memory." Neural Computation, 9(8), 1735–1780.

[5] Sutton, R. S., and Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.). MIT Press, Cambridge, MA.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., … and Hassabis, D. (2015). "Human-level control through deep reinforcement learning." Nature, 518(7540), 529–533.

[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347.

[8] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." Software: Practice and Experience, 41(1), 23–50.

[9] Shashank Daram, “A Review of Retrieval-Augmented Generation in Natural Language Processing: Architectures, Challenges, and Future Research Directions” , *International Journal on Recent and Innovation Trends in Computing and Communication*; vol 8 (12), 2020 :145-155.

[10] Lim, H. C., Babu, S., Chase, J. S., and Parekh, S. (2009). "Automated control in cloud computing: Challenges and opportunities." Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, pp. 13–18.

[11] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. C. (2020). "Characterizing, modeling, and predicting microservice dependencies for autoscaling in cloud-native applications." IEEE Transactions on Services Computing, 14(6), 1856–1869.

[12] Toka, L., Dobreff, G., Fodor, B., and Sonkoly, B. (2021). "Adaptive machine learning-based autoscaling for heterogeneous workloads in Kubernetes environments." IEEE Transactions on Network and Service Management, 18(3), 3306–3320.

[13] Nguyen, T., Yeow, J., Phan, T., and Nguyen, M. (2022). "Container resource right-sizing using Bayesian optimisation for microservice deployments." Journal of Systems and Software, 184, 111128.

[14] Mao, H., Alizadeh, M., Menache, I., and Kandula, S. (2016). "Resource management with deep reinforcement learning." Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets'16), pp. 50–56.

[15] Peng, Z., Lin, H., and Zhang, W. (2019). "Virtual machine consolidation using multi-agent deep reinforcement learning in private cloud environments." IEEE Access, 7, 83927–83940.

[16] Rzadca, K., et al. (2020). "Autopilot: Workload autoscaling at Google." Proceedings of the European Conference on Computer Systems (EuroSys'20), pp. 1–16.

[17] Shashank Lohani, The Learning Management System as a Product: A Computer Science Perspective on Higher Education Strategy, International Journal of Communication Networks and Information Security; vol 11 (2), 2019 page no 353-363.

[18] Chen, W., Shi, J., Li, B., and Li, Z. (2022). "An actor-critic reinforcement learning approach for Kubernetes pod autoscaling in multi-tenant serverless environments." Future Generation Computer Systems, 134, 97–111.

[19] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2016). "Dueling network architectures for deep reinforcement learning." Proceedings of ICML 2016, Vol. 48, pp. 1995–2003.