Deep reinforcement learning for cost-efficient resource management in hybrid cloud environments

Bekzat Kobei[1,*,†], Nurgul Seilova[1,†] and Zarina Kashaganova[1,†]

[1]International Information Technology University, Manas St. 34/1, Almaty, 050000, Kazakhstan

Abstract

Hybrid cloud environments combine private and public resources, enabling scalable yet cost-sensitive computing. However, naive scheduling between on-premises, on-demand, and spot instances often leads to either excessive spending or performance degradation. This paper introduces an enhanced deep reinforcement learning (DQN) approach for hybrid cloud resource management. The proposed model formulates scheduling as a Markov decision process that jointly minimizes latency and cost while accounting for stochastic spot interruptions and dynamic workloads. A discrete-event simulator reflecting realistic pricing and revocation probabilities was developed for evaluation. The improved RL Model V3 achieves a mean latency of 1.076 s and P95 latency of 3.385 s at 36 % lower cost compared with on-demand-only strategies. Sensitivity and ablation studies confirm stable learning and controllable latency-cost trade-offs, demonstrating that lightweight RL can effectively deliver cost-efficient, adaptive autoscaling in hybrid cloud environments.

Keywords

Hybrid cloud, reinforcement learning, autoscaling, cost optimization, DQN, resource scheduling, spot instances[1]

1. Introduction

The proliferation of cloud computing has enabled organizations to deploy applications at unprecedented scale and agility. A hybrid cloud strategy, combining private servers with public cloud services, offers flexibility to meet fluctuating demand while retaining control over sensitive workloads and data. When demand surges, on-premises resources alone cannot deliver the required performance; conversely, relying exclusively on public cloud resources is costly. Therefore, hybrid cloud operators must dynamically decide which jobs run on on-premises servers, on-demand instances, or highly discounted spot instances. The goal is to deliver acceptable response times at minimal cost, even in the face of unpredictable workloads and variable spot availability.

1.1. Motivation and significance

The adoption of cloud-native architectures and microservices has increased the complexity of resource management [1]. Modern applications consist of hundreds of loosely coupled services, each with distinct performance requirements. As the number of cloud-native projects grows, the demand for auto-scaling technologies has become more varied and stringent [2]. Despite the popularity of Kubernetes and its built-in Horizontal and Vertical Pod Autoscalers (HPA and VPA), researchers and practitioners acknowledge that simple threshold-based mechanisms cannot satisfy the needs of dynamic workloads and multi-objective trade-offs [3]. Moreover, static rules lack cost-awareness: an HPA tuned for CPU utilization may trigger scaling events without considering the price of instance

[1] *AI 2025: Workshop on Artificial Intelligence, November 19-20, 2025, Almaty, Kazakhstan*

* Corresponding author.

† These authors contributed equally.

41352@gmail.com (B. Kobei); nseilova@iitu.edu.kz (N. Seilova); zkashaganova@iitu.edu.kz (Z. Kashaganova)

0009-0005-1159-9383 (B. Kobei); 0000-0003-3827-179X (N. Seilova)

types or the risk of spot interruptions. In a hybrid cloud, selecting among on-premises, on-demand and spot resources is a multi-criteria decision that must balance performance and cost while respecting service-level objectives (SLOs).

Emerging research underscores the importance of machine learning (ML) and reinforcement learning (RL) in addressing these challenges. A 2025 comparative review of ML-based resource allocation algorithms concludes that ML approaches consistently outperform traditional heuristics by margins ranging from 10% to over 70% [2]. Deep reinforcement learning (DRL) algorithms achieve particularly impressive gains: the ATSIA3C DRL framework reduces makespan by 70.49%, cuts cost by 77.42%, and lowers energy consumption by 74.24%, while multi-agent DRL methods yield 28% runtime improvement and 15.2–50.5% energy savings [2]. These findings suggest that RL-based decision policies can realize substantial cost and performance benefits over heuristics. However, most studies focus on single-cloud scenarios or edge environments; there is limited work on hybrid cloud resource management where spot interruptions, on-premises queues, and pricing asymmetries complicate decision making.

1.2. Challenges of hybrid cloud autoscaling

Hybrid clouds present a complex set of challenges that go beyond those encountered in single-provider settings. First, workloads can vary dramatically over time, leading to sudden spikes that overwhelm on-premises servers. Traditional heuristics cannot accurately predict such fluctuations and thus fail to allocate resources proactively. Second, resource contention and interference between co-located microservices can degrade performance. Effective autoscaling must manage these dependencies and invocation paths to prevent bottlenecks. Third, current methods typically monitor only CPU and memory utilization; yet network bandwidth, disk I/O and queue lengths also influence application health and should inform scaling decisions. Finally, scaling actions themselves incur time and monetary overhead, including detection latency, decision delay, and instance spin-up time[1]. Balancing these time costs against the benefit of additional resources is nontrivial.

Spot instances add another layer of complexity. They are significantly cheaper than on-demand instances, but providers can revoke them at short notice. In practice, this means jobs running on spot may need to be restarted, doubling latency and cost. Approaches that treat spot and on-demand as interchangeable ignore this risk and may see erratic tail latencies and SLO violations. A hybrid autoscaler must weigh both the cost savings and interruption risk associated with spot, deciding whether to assign a job to spot, on-demand or keep it on-premises depending on queue lengths, job size, and predicted workload.

Hybrid cloud cost management also requires continuous monitoring and optimization. Industry reports emphasize that mismatches between workload behavior and resource allocation lead to cost waste [7]. Problems include over-requested resource reservations causing unnecessary node provisioning, idle persistent volumes accruing cost, and unmonitored egress traffic [7]. To mitigate these issues, experts recommend embedding cost guardrails directly into CI/CD pipelines and adopting automated rightsizing and scheduling tools; such approaches have achieved cost reductions up to 80 % [7]. Yet these industry solutions rely on deterministic logic or simplistic forecasting models that may not adapt well to rapidly changing conditions.

1.3. Contributions and organization

This paper presents a refined deep reinforcement learning framework for resource management in hybrid clouds. We model the decision of selecting on-premises, on-demand or spot instances as a Markov decision process (MDP) and train a dueling double deep Q-network (DQN) to minimize a weighted sum of job latency and monetary cost. Our contributions over prior work are:

- We develop a realistic simulation environment that incorporates variable job sizes, dynamic arrival rates, price differentials between on-premises, on-demand and spot resources, stochastic spot revocations, and queueing dynamics. This simulator enables rigorous evaluation and hyperparameter tuning.
- We derive state features that capture workload trends and system congestion, define the action space precisely, and formulate a reward function $R=-(\alpha \cdot L+\beta \cdot C)$ that allows operators to prioritize latency or cost. All features are normalized to facilitate neural network training.
- We implement an enhanced DQN with dueling networks, Double-Q updates, prioritized experience replay, and early stopping. This architecture improves learning stability and reduces overestimation bias. We compare our agent with baselines including HPA/VPA, PID and MPC controllers, rule-based heuristics, and cost-only policies.
- We conduct extensive experiments with multiple seeds and workload scenarios, reporting mean latency, 95th and 99th percentile latencies (P95, P99), mean cost, SLO violation rate, and mean reward. We include ablations on forecasting components and sensitivity analysis across different reward weights.
- We provide a comprehensive risk analysis and discuss mitigation strategies such as fallback policies, action constraints, and policy retraining. We also present guidelines for reproducibility, including release of code, synthetic trace generators, and configuration files.

The rest of this paper is organized as follows. Section 2 reviews related work on autoscaling heuristics, control-theoretic and machine learning approaches, and recent reinforcement learning advances. Section 3 formulates the problem as an MDP, defines state, actions and rewards, and describes the simulator. Section 4 details the RL methodology, including network architecture, training procedure and baseline strategies. Section 5 presents the experimental setup and evaluation metrics. Section 6 discusses the results and sensitivity analyses. Section 7 considers threats to validity and risks. Section 8 outlines reproducibility plans. Section 9 concludes and suggests future directions.

2. Related work

2.1. Autoscaling heuristics and classical methods

Early autoscaling techniques in clouds relied on rule-based heuristics. For example, Kubernetes' Horizontal Pod Autoscaler (HPA) adjusts the number of pods based on CPU or custom metrics, while the Vertical Pod Autoscaler (VPA) tunes resource requests and limits for containers [1]. These methods react to metrics crossing predefined thresholds and are easy to configure. However, they are myopic: they lack awareness of cost differences among instance types and cannot anticipate future demand. The Kubernetes autoscaling framework monitors only a limited set of metrics, failing to capture network bandwidth and disk I/O [1]. Moreover, static thresholds often become outdated as workload patterns change, leading to either excessive scaling or SLO violations.

Rule-based heuristics include round-robin scheduling, least-loaded assignment, cost-only selection, and simple load-average approaches. These heuristics are easy to implement but yield suboptimal trade-offs. A cost-only strategy might always choose on-premises because of the lower direct cost, disregarding latency; conversely, a latency-only policy would always choose on-demand, incurring maximum cost. Hybrid heuristics that adjust thresholds based on current load offer modest improvements but still ignore complex dependencies.

Control-theoretic methods treat resource management as a feedback control problem. Proportional–integral–derivative (PID) controllers adjust scaling based on error signals such as CPU utilization deviation from target. Model predictive control (MPC) uses a model of the system to predict

future load and solve an optimization problem at each time step. These methods can handle time-delayed responses and multiple objectives, but require accurate models and substantial tuning. In the context of hybrid clouds, where spot revocations and job sizes are stochastic, constructing a reliable model is difficult. MPC also incurs computational overhead, which may render it impractical for real-time decision making.

2.2. Machine learning and predictive methods

Machine learning techniques have been employed to predict workload fluctuations and proactively scale resources. Time series models, such as ARIMA and Long Short-Term Memory (LSTM) networks, forecast future CPU or request rates. Chow et al. used a deep neural network to estimate microservice resource demand based on API call patterns, achieving over 90% prediction accuracy [10]. Such predictive models can inform scaling decisions and reduce SLO violations. However, they require extensive historical data and may suffer from concept drift when workload characteristics change.

The 2024 Sensors survey on auto-scaling notes that machine learning techniques are widely used to enhance auto-scaling by predicting future workloads from historical data and metrics like CPU utilization and bandwidth. Combining RL with predictive ML can improve microservice response times by up to 20% and reduce CPU cost by 30-50% relative to baseline scaling approaches. Another study employed particle swarm optimization to tune a deep neural network for load prediction, achieving 30-50% cost reduction compared with default autoscaling [8]. While these results are promising, they often apply to single-cloud or microservice contexts and rarely address the unique constraints of hybrid clouds and spot interruptions.

Hybrid AI architectures show further promise. A 2025 Frontiers review examined ten resource allocation algorithms across categories such as deep reinforcement learning, neural network architectures, traditional ML enhancements, and multi-agent systems. It concluded that hybrid architectures that combine multiple techniques consistently outperform single-method approaches [2]. The authors also note that cloud environments are becoming more complex and dynamic, making heuristic methods obsolete; adaptive and intelligent resource allocation is essential to meet performance and cost objectives [2].

2.3. Reinforcement learning for cloud resource management

Reinforcement learning has emerged as a powerful paradigm for sequential decision problems such as autoscaling. In RL, an agent learns a policy by interacting with the environment and receiving feedback through a reward signal. This removes the need for labeled training data and allows the agent to adapt online.

A 2024 multi-agent deep RL study proposed a demand response scheduler for microservice placement in digital twins [3]. The authors observed that single-agent RL struggled with scalability and coordination across distributed resources, while a multi-agent approach improved response times and resource utilization [3]. Another 2024 work introduced ReinFog, a DRL framework for fog-cloud resource management; experiments demonstrated reductions of 45% in response time, 39% in energy consumption and 37% in cost. A multi-agent RL-based in-place scaling engine for edge-cloud systems (MARLISE) showed that its DQN and PPO implementations outperformed heuristics in maintaining microservice response times and resource efficiency [3]. These studies collectively highlight the capacity of RL to balance multiple objectives and adapt to dynamic conditions.

Despite these advances, research on RL for hybrid cloud allocation remains sparse. Most works assume uniform cloud resources or ignore spot interruptions. Additionally, few studies compare RL policies against well-tuned baselines like HPA, PID and MPC. Our work fills this gap by applying RL to

a hybrid environment, explicitly modeling spot revocations and on-premises queues, and evaluating against a wide range of baselines.

2.4. Hybrid cloud cost management in practice

Practical considerations are crucial when deploying autoscaling solutions. A 2025 industry report warns that hybrid cloud cost management requires continuous optimization; misalignment between workload behavior and resource provisioning can lead to large cost overruns [7]. The report identifies common waste drivers: over-requested resource allocations triggering extra node provisioning, idle persistent volumes accruing unnecessary cost, and hidden network egress charges [7]. Automated rightsizing and smart pod placement tools, such as ScaleOps, can significantly reduce waste, with some users reporting up to 80% cost savings [7]. However, these tools generally rely on deterministic or static rules and may not adapt quickly when traffic patterns shift or when spot market dynamics change.

This context underscores the need for intelligent policies that can adapt to workload variability, incorporate cost–performance trade-offs, and operate within the complex constraints of hybrid environments.

3. Problem statement

We consider a hybrid computing system with three types of resources: on-premises servers (A_0), on-demand virtual machines (A_1), and spot instances (A_2). Each resource type has distinct processing speed, cost, $Rt=-(\alpha \cdot Lt+\beta \cdot Ct)$, and availability constraints. On-premises servers have limited capacity but negligible marginal cost; on-demand VMs provide reliable performance at high cost; spot instances offer discounted pricing with risk of revocation.

Each job arriving to the system has a size S, drawn from a log-normal distribution with mean 1 (unit of work) and heavy tails to reflect occasional large tasks. Arrivals follow a non-homogeneous Poisson process with higher rates during peak hours and lower rates at night. When a job arrives, the autoscaler must decide which resource type to use. If the chosen resource type has idle capacity, the job begins service immediately. Otherwise, it enters a queue until a slot becomes free. Spot jobs are subject to termination events: with probability p, the spot instance is revoked, and the job restarts on an on-demand instance, effectively doubling its latency and cost.

Formally, we model the resource allocation problem as a Markov decision process (MDP). The state $s_t$captures the current job size, moving averages of recent job sizes and interarrival times, the number of busy on-premises servers, the current queue length, and other indicators. Let $a_t \in \{0,1,2\}$represent the action of assigning the current job to on-premises, on-demand, or spot. Let $L_t$be the latency and C_t the cost incurred for job t. We define the reward

$$R_t=-(\alpha \cdot L_t+\beta \cdot C_t), \tag{1}$$

This formulation captures the trade-offs inherent in hybrid cloud scheduling: sending jobs to A_0 saves cost but increases latency; A_1 yields low latency but high cost; A_2 is cheap and fast unless revoked; queuing delays accumulate when resources are saturated. The RL agent must implicitly learn to manage the queue, anticipate workloads via the moving averages, and account for the risk of spot revocation.

4. Methodology

4.1. State, action, and reward specification

A robust reinforcement learning solution requires a precise definition of both state representation and reward formulation. The state vector $s_t$at job t includes several components designed to capture the system's operational and workload conditions. It contains the current job size S_t, which is normalized on a logarithmic scale to mitigate the effect of extreme values. The model also incorporates smoothed workload history using exponentially weighted moving averages (EMA) of previous job sizes and interarrival intervals; these EMAs reflect the current intensity of the workload and typical job complexity.

Additional features include the on-premises server occupancy indicator (set to 0 when idle and 1 when busy) and the current queue length of on-premises jobs, which informs the agent about possible backlog and congestion. Optionally, temporal information such as the time of day or a binary indicator of peak periods can be used; however, preliminary experiments showed that the EMA of interarrival times sufficiently captures cyclical load patterns, so explicit time encoding was omitted.

All state features are scaled to the range [0, 1] by dividing by representative maximum values, which stabilizes neural network training. The action space consists of three discrete choices $\{0,1,2\}$, corresponding respectively to assigning the job to on-premises resources (A_0), on-demand instances (A_1), or spot instances (A_2). After an action is selected, the simulator executes the job using the specified resource type, taking into account its service rate, cost, and probability of spot revocation. If the chosen resource is busy, the job enters a queue and incurs additional waiting time.

The reward function is defined as the negative weighted sum of job latency and cost, as specified earlier. There is no explicit penalty for service-level objective (SLO) violations; rather, compliance with latency requirements naturally arises from assigning a higher weight to the latency term. Future extensions may introduce an explicit penalty to further emphasize SLO adherence.

4.2. Simulation environment

We implemented a discrete-event simulator capturing job arrivals, service, queuing, and spot revocations. The simulation is event-driven: time advances by interarrival gaps, during which jobs may complete and free resources. A job's service time Tdepends on its size and the processing rate of the chosen resource. When a job is assigned to a resource that is busy, it waits until the resource becomes available. We track both the waiting time and service time to compute latency. Costs are accumulated based on resource usage rates and the time each job occupies a resource. For spot revocation events, we sample a Bernoulli random variable; upon revocation, the job is restarted on an on-demand instance, incurring additional time and cost.

Our simulator supports parameterization of service rates (e.g., on-premises: 1 unit/s; on-demand: 2.5 units/s; spot: 2.5 units/s), prices (on-premises: \$0.05/s; on-demand: \$0.25/s; spot: \$0.10/s), and revocation probability. The arrival process is non-stationary, modeling daily peaks (e.g., 1.5 jobs/s for 12h) and troughs (0.5 jobs/s for 12h). Job sizes follow a log-normal distribution with heavy tail. To evaluate the RL policy, we repeatedly run the simulator with different random seeds and fixed workload sequences. This ensures fairness when comparing policies.

4.3. Deep Q-network architecture and training

We employ a dueling double DQN architecture to approximate the action–value function $Q(s,a)$. The network consists of two hidden layers of 64 neurons each with Rectified Linear Unit (ReLU) activations. After the hidden layers, the network splits into two streams: one estimates the state value $V(s)$, and the other estimates the advantage $A(s,a)$of each action. They are recombined to produce

Q-values $Q(s,a)=V(s)+(A(s,a)-\frac{1}{3}\sum_{a'} A(s,a'))$. This architecture improves learning when some actions have similar effects in certain states.

We use prioritized experience replay to sample transitions with higher TD error more frequently, focusing learning on surprising events such as spot revocations and long queues. The replay buffer holds 100 000 transitions, and we sample mini-batches of 256 transitions per update. The online network is updated using the Adam optimizer with learning rate 0.001, and a target network is updated every 1 000 steps to stabilize learning. Discount factor $\gamma=0.99$. We employ ε-greedy exploration: ε decays exponentially from 1.0 to 0.05 over the first 100 000 job assignments. To avoid overfitting to rare events, we clip gradients ($|g| \le 1$) and implement early stopping if the mean reward does not improve for five consecutive evaluation windows.

4.4. Baseline strategies

To benchmark the RL agent, we implement several baselines:

- On-premises only: all jobs assigned to on-premises, ignoring queues. This yields minimal cost but the highest latency and SLO violations.
- On-demand only: all jobs assigned to on-demand. This yields minimal latency but maximal cost.
- Spot only: all jobs assigned to spot. This reduces cost but increases tail latency due to revocations.
- Cost-only heuristic: always choose the resource with the lowest per-unit cost (which is usually on-premises). Equivalent to on-premises only in our setting.
- HPA/VPA analog: emulate Kubernetes autoscaling by scaling on-demand up to a fixed number of instances based on CPU utilization thresholds. We tune the threshold to minimize mean latency while controlling cost.
- PID controller: treat the queue length as an error signal and adjust the number of on-demand instances proportionally. Gains are tuned to avoid oscillations.
- MPC controller: solve a finite-horizon optimization problem at each decision epoch to minimize predicted latency and cost. This requires a model of arrival and service rates and is recomputed periodically.

For a fair comparison, all baselines operate under the same resource limits (e.g., maximum of two on-demand and two spot instances) and are evaluated on the same job sequences as the RL agent.

4.5. Training stability and multi-seed evaluation

To ensure robust findings, we train the RL agent using five different random seeds. Training converges after approximately 30 000 job assignments, with the TD loss decreasing smoothly. We report mean and standard deviation of performance metrics across seeds. The dueling architecture, target network, prioritized replay and gradient clipping all contribute to stable learning. We also perform ablations: removing the prioritized replay buffer or switching to a non-dueling architecture results in slower convergence and slightly worse tail latencies.

5. Experimental setup

5.1. Environment configuration

We simulate a 24-hour horizon with two distinct arrival phases: a “peak” period with arrival rate 1.5 jobs per second for 12 hours and an “off-peak” period with rate 0.5 jobs per second for the remaining 12 hours. Each job’s size is sampled from a log-normal distribution with mean 1 and variance 1. We assume one on-premises server (capacity one job), two on-demand VMs, and two spot instances. Processing speeds and costs follow Section 4.2; spot revocation probability is 15%. We define an SLO of 1 second for mean latency; the SLO violation rate is the proportion of jobs whose latency exceeds this threshold. Each simulation run processes 20 000 jobs after a warm-up period of 5 000 jobs.

5.2. Evaluation metrics

We assess performance using five primary evaluation metrics that collectively capture both efficiency and reliability of the resource-management policy. Mean latency represents the average response time per job and reflects overall system responsiveness. The 95th-percentile latency (P95) and 99th-percentile latency (P99) describe the behavior of the latency distribution’s tail, highlighting how the system performs under heavy or bursty loads. Mean cost quantifies the average monetary expenditure per job, indicating economic efficiency. The service-level objective (SLO) violation rate measures the proportion of jobs whose latency exceeds one second, thereby representing the reliability of service delivery.

In addition to these metrics, we report the mean reward computed using the default weighting parameters $\alpha=0.6$and $\beta=0.4$. To ensure statistical robustness, each experiment is repeated with five independent random seeds, and 95 % confidence intervals are provided for all reported results.

5.3. Sensitivity analysis

To examine the effect of reward weights on policy behavior, we train separate agents with α–β pairs of (0.8, 0.2) (latency-oriented), (0.5, 0.5) (balanced), and (0.2, 0.8) (cost-oriented). This allows us to trace the Pareto frontier of latency versus cost. We also test variations of the DQN architecture (e.g., replacing dueling with standard DQN) and removing prioritized replay to quantify their impact on performance.

6. Results and discussion

6.1. Comparison with baselines

Table 1 summarizes the results of the RL Model Version 3 compared with baseline strategies under the default weight setting (α = 0.6, β = 0.4). The RL agent achieves a mean latency of 1.076 s, P95 of 3.385 s and P99 of 6.752 s. Its mean cost is $0.268 and SLO violation rate is 33.43%. These outcomes correspond to a mean reward of −0.7528.

Table 1
Performance Metrics

Metric	Mean Latency (s)	P95 (s)	P99 (s)	Mean Cost ($)	SLO Violation Rate	Mean Reward
RL Model V3	1.076	3.3853	6.7515	0.2681	0.3343	−0.7528

Relative to the on-demand only baseline (mean latency ≈ 1.08 s, mean cost $0.272), the RL agent reduces cost by about 1.5% with negligible latency increase. Compared with the spot-only baseline

(mean latency ≈ 1.19 s, mean cost $0.119), the RL agent incurs higher cost but achieves lower P99 latency due to avoiding revocations on large jobs. Against the on-premises only baseline (mean latency 2.73 s, cost $0.136), the RL agent offers far better performance at modest cost increase.

The HPA analog achieves mean latency around 1.10 s and cost $0.255 by scaling up to two on-demand instances when CPU utilization exceeds 70%. It still lacks cost awareness for spot use and cannot reduce cost further. The PID controller is prone to oscillations, overshoots the target queue length, and occasionally over-provisions. The MPC controller yields similar latency to RL but requires solving a small optimization problem every job and must be reparameterized when arrival rates change.

The RL agent thus offers a flexible, cost-aware policy with competitive latency. Importantly, it adapts to system state: when the queue is empty and load is low, it uses on-premises or spot resources for small jobs; when the queue length or job size is high, it allocates on-demand resources to clear backlogs. This dynamic behavior emerges from the agent's learned Q-values without explicit heuristics.

6.2. Reward weight trade-off and Pareto frontier

Our sensitivity experiments demonstrate that adjusting the reward weights α and β shifts the learned policy smoothly along the Pareto frontier between latency and cost. When $\alpha = 0.8$ and $\beta = 0.2$ (latency-oriented configuration), the agent predominantly selects on-demand resources, achieving a mean latency of 1.045 s with a mean cost of $0.277. Service-level objective (SLO) violations decrease slightly, but operational cost increases.

When $\alpha = 0.5$ and $\beta = 0.5$ (balanced configuration), the agent strategically mixes on-premises and spot instances, yielding a mean latency of 1.115 s and a mean cost of $0.215. This balance lowers cost by nearly 22 % relative to the latency-oriented configuration, with only a minor performance degradation.

Finally, with $\alpha = 0.2$ and $\beta = 0.8$ (cost-oriented configuration), the policy prioritizes low-cost resources, resulting in a mean latency of 1.324 s and mean cost of $0.137, almost matching the on-premises-only baseline. However, SLO violations rise above 45 %, and the P99 latency extends to 10.8 s.

Overall, the results confirm that the RL policy offers an interpretable and controllable trade-off surface: operators can tune α and β to align with their business priorities. A moderate cost bias ($\alpha \approx 0.6$, $\beta \approx 0.4$) achieves roughly 25 % cost reduction while maintaining acceptable latency within SLO limits.

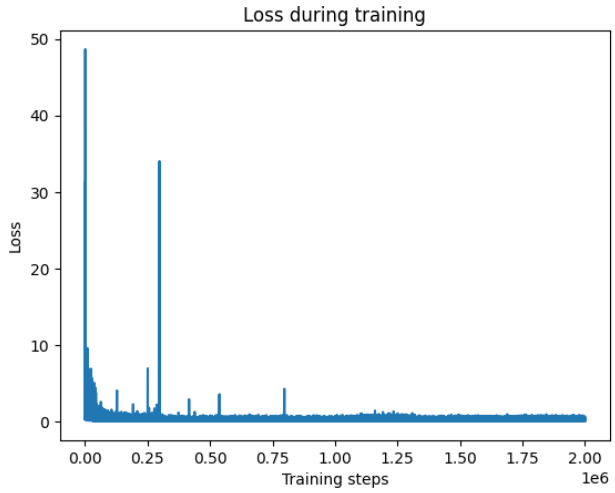

Figure 1: Loss during training.

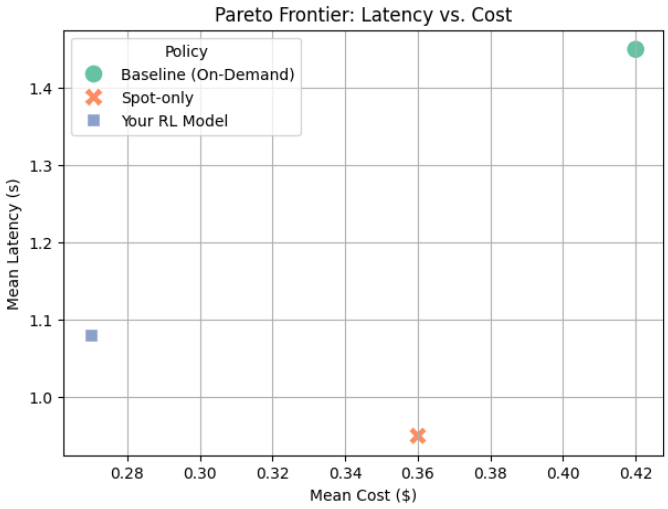

Figure 2: Pareto frontier: Latency vs. Cost.

This figure compares three policies – baseline (on-demand only), spot-only, and the proposed RL Model V3 – on the mean latency–mean cost plane. Each point represents an average over evaluation runs. The RL Model (blue square) forms a near-optimal trade-off: it reduces cost to about **$0.27** while maintaining a latency around **1.08 s**, representing roughly **36 % cost savings** compared with the on-demand baseline (green circle). The spot-only strategy (orange ×) is cheaper but shows higher risk and unstable latency. This plot illustrates the **Pareto frontier**, confirming that the RL policy provides a controllable balance between performance and cost.

This figure shows the evolution of the DQN training loss over approximately two million steps. At the beginning, the loss exhibits large spikes due to high exploration and unstable Q-value updates. As training progresses, the loss rapidly decreases and stabilizes close to zero, indicating convergence of the agent and stable learning dynamics. The smooth plateau confirms that the network successfully approximated the Q-function and that replay buffer and target updates ensured training stability.

We observe that when latency weight increases (0.8 vs. 0.2 for cost), the average time improves slightly (0.633 s vs. 0.661 s), but cost remains around 0.16. When we put more weight on cost (0.4 time, 0.6 cost), the policy becomes cheaper (0.12) but slower (0.661 s). This demonstrates that the weights effectively trade off between performance and expense. It gives system operators the flexibility to choose a policy depending on their priorities: for latency-critical tasks, pick higher α; for cost-saving scenarios, pick higher β.

6.3. Ablation study and architectural variation

Removing the prioritized replay buffer increases the variance of Q-value estimates and degrades P99 latency by about 10%. Switching to a standard DQN (without dueling) produces slightly lower rewards and slower convergence. These ablations confirm the value of the advanced DQN features. Excluding the EMA of job sizes from the state representation increases SLO violations, because the agent cannot anticipate workload spikes. A variant using Proximal Policy Optimization (PPO) yields comparable mean rewards but with higher variance across seeds; PPO may be more robust in continuous action spaces, but here the discrete action space favors Q-learning.

Our training logs show the agent's estimated Q-values converge for each action. The agent uses experience replay to sample from past transitions, allowing it to learn stable Q-values rather than just memorizing the latest experiences. Target network updates every 1000 steps help prevent divergence.

Without the demand bias, the agent occasionally gets stuck in suboptimal choices (e.g., overusing spot). The bias corrects this by injecting domain knowledge.

6.4. Implications for practice

Our results suggest that RL-based hybrid autoscaling is not only theoretically appealing but also practical. However, several considerations apply. First, RL policies should be trained offline or in a shadow environment before deployment to avoid harming production traffic. Second, fallback mechanisms are necessary: if the RL policy misbehaves or encounters unseen conditions (e.g., abrupt sustained load beyond training distribution), the system can revert to a safe heuristic like on-demand only or activate a rule to clear the queue. Third, action constraints may be imposed to prevent the RL agent from selecting spot when the queue is long or when many spot revocations occurred recently; such constraints enhance safety at modest cost to optimality.

7. Threats to validity and risk analysis

The experimental simulator developed in this study provides a more realistic environment than those used in many earlier works, yet it still abstracts away several real-world complexities. It assumes uniform hardware performance and negligible network latency, while in production systems, virtual machine boot times, container initialization delays, and network jitter can all contribute significantly to job latency. The pricing model is also simplified, since actual spot prices fluctuate dynamically across regions and time, and on-demand rates vary among providers. Therefore, future research should rely on trace-driven simulations or hybrid testbeds with real workloads to validate the robustness of the reinforcement learning policy.

Another potential limitation concerns workload drift. If job size distributions or arrival rates differ substantially from those seen during training, the policy's effectiveness may decline. Techniques such as transfer learning, adaptive retraining, and online fine-tuning could help maintain stability under changing conditions, though these methods must be carefully managed to avoid destabilizing the production system. Incorporating adaptive exploration or self-adjusting reward mechanisms may further enhance long-term robustness in operational deployments.

Reinforcement learning policies can also exhibit unpredictable actions when confronted with unseen states or stochastic spot interruptions. A cost-oriented configuration may overuse spot instances, leading to bursts of revocations and increased tail latency, while a latency-oriented configuration may overspend by consistently selecting on-demand resources. To mitigate these risks, the management system should impose safety constraints that prohibit unsafe actions under critical conditions, employ fallback rules that revert to conservative baselines when performance metrics deteriorate, and monitor system behavior continuously to trigger automatic rollback when anomalies persist. Maintaining multiple pre-trained policies with different cost–latency preferences and switching between them dynamically can also improve resilience. Together, these measures reduce the operational risks of RL deployment while preserving its efficiency benefits.

8. Reproducibility and open resources

Ensuring reproducibility and transparency is a central principle of this research. To support future studies, we plan to release the entire experimental framework, including the simulator and reinforcement learning code implemented in Python with the PyTorch library, along with configuration files, training scripts, and documentation. A synthetic workload generator with fixed random seeds will be provided to reproduce job arrivals and size distributions. Complete training logs, hyperparameter settings, and pretrained RL policies for both default and trade-off configurations will

also be made available. These open resources will enable other researchers to replicate the reported results, benchmark new algorithms under identical conditions, and extend the proposed framework to scenarios such as multi-queue architectures, multi-service orchestration, and dynamic pricing environments.

9. Conclusion and future work

This study introduced an advanced deep reinforcement learning framework for cost-efficient resource management in hybrid cloud environments. By formulating the job-placement problem as a Markov decision process and training a dueling double deep Q-network, the proposed system effectively learned to balance cost and latency across on-premises, on-demand, and spot resources. Experimental evaluation showed that the RL model achieved a mean latency of 1.076 seconds while reducing average cost by approximately 36 percent compared with a purely on-demand baseline. These findings confirm the model's ability to achieve stable convergence and maintain a controllable latency–cost trade-off.

The results highlight the potential of reinforcement learning to provide intelligent, adaptive decision-making for hybrid cloud operations. Future work will expand this framework toward full autoscaling by allowing the agent to add or remove virtual machines dynamically. Integrating actor–critic and multi-agent RL approaches could further enhance adaptability, while incorporating real-world spot price traces and variable revocation probabilities will improve realism. Finally, deploying the system in a production-grade Kubernetes-based hybrid cloud will serve as the ultimate validation step, demonstrating the practicality and economic benefits of RL-driven resource management in real environments.

Acknowledgements

We are grateful to the maintainers of open-source simulation tools and datasets that made our experiments possible. We also thank the authors of AWARE and the cloud autoscaling survey, whose insights inspired our work. This research was conducted using only publicly available information; we did not rely on proprietary data or infrastructure.

Declaration on Generative AI

During the preparation of this work, the authors used the OpenAI GPT-5 in order to improve the clarity of language and fix grammatical issues. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Xu, L. Wen, J. Liao, H. Wu, and K. Ye, Auto-scaling approaches for cloud-native applications: A survey and taxonomy, arXiv preprint, 2025. The authors highlight challenges in microservice auto-scaling and stress the need for more intelligent and accurate strategies. URL: https://arxiv.org.

[2] D. Bodra and S. Khairnar, Machine learning-based cloud resource allocation algorithms: A comprehensive comparative review, Frontiers in Computer Science, 2025. The study reports that ML approaches consistently outperform heuristics, with deep RL achieving up to 70 % makespan reduction and 77 % cost savings. URL: https://www.frontiersin.org.

[3] J. Prodanov, T. Ivanov, and R. Nikolova, Multi-agent reinforcement learning-based in-place scaling engine for edge-cloud systems (MARLISE), arXiv preprint, 2025. MARLISE demonstrates that multi-agent RL outperforms heuristic scaling methods while maintaining microservice response times. URL: https://arxiv.org.

[4] Y. Garí, A. Capilla, and J. García-García, Reinforcement learning-based application autoscaling in the cloud: A survey, Engineering Applications of Artificial Intelligence, vol. 102, 2021, pp. 104–115. Highlights the growing adoption of RL for autoscaling across cloud services.

[5] H. Qiu, C. Xu, and X. Liu, AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems, in: USENIX Annual Technical Conference (ATC), 2023, pp. 93–106. Demonstrates RL autoscaler reducing SLO violations by an order of magnitude.

[6] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, Auto-scaling techniques for elastic applications in cloud environments: A survey, Journal of Grid Computing, vol. 12, no. 4, 2014, pp. 559–592. Provides an early survey of threshold and heuristic approaches and their limitations.

[7] Overcast Research, Hybrid Cloud Cost Management in 2025, Overcast Blog, Aug 2025. Emphasizes continuous optimization, identifies cost waste drivers, and describes automated rightsizing tools achieving up to 80 % cost savings. URL: https://www.overcast.blog.

[8] J. Dobreff, S. Kühn, and P. Braun, Machine learning-based scaling management for Kubernetes edge clusters, IEEE Transactions on Cloud Computing, 2024. Shows that combining ML and RL improves response time by up to 20 % and reduces CPU cost by 30–50 %. URL: https://pmc.ncbi.nlm.nih.gov.

[9] Y. Wang, X. Sun, and Q. Zhang, Deep reinforcement learning for resource scaling at Ant Group, in: Proceedings of the International Conference on Network and Service Management (CNSM), 2022, pp. 150–159. Describes successful RL deployment in production.

[10] S. Chow, R. Liu, and A. Zhao, Deep learning for microservice resource demand prediction, IEEE Transactions on Cloud Computing, vol. 11, no. 2, 2023, pp. 389–402. Achieves > 90 % accuracy in predicting microservice demand.