



# Latency-Aware Orchestration of Microservices in Heterogeneous Kubernetes Clusters Using Reinforcement Learning


**Sava Stanisic**

(University of Kragujevac, Faculty of Technical Sciences, Cacak, Republic of Serbia  
 <https://orcid.org/0009-0002-3118-0537>, sava.stanisic@vs.rs)


**Borislav Djordjevic**

(Mihajlo Pupin Institute, Belgrade, Republic of Serbia  
 <https://orcid.org/0000-0002-6145-4490>, borislav.djordjevic@pupin.rs)


**Branislav Belotic**

(University of Kragujevac, Faculty of Technical Sciences, Cacak, Republic of Serbia  
 <https://orcid.org/0009-0007-0638-7938>, branislav.belotic@vs.rs)


**Olga Ristic**

(University of Kragujevac, Faculty of Technical Sciences, Cacak, Republic of Serbia  
 <https://orcid.org/0000-0002-1723-0940>, olga.ristic@ftn.kg.ac.rs)


**Ivan Tot**

(University of Defence, Military Academy, Belgrade, Republic of Serbia  
 <https://orcid.org/0000-0002-5862-9042>, ivan.tot@va.mod.gov.rs)

**Kristina Zivanovic**

(University of Belgrade, School of Electrical Engineering, Belgrade, Republic of Serbia  
 <https://orcid.org/0009-0004-3648-3400>, kristina.zivanovic@vs.rs)

**Dimitrije Kolasinac**

(University of Belgrade, School of Electrical Engineering, Belgrade, Republic of Serbia  
 <https://orcid.org/0009-0008-9505-2482>, dimitrije.kolasinac@vs.rs)

**Abstract:** The orchestration of microservices in distributed cloud environments poses significant challenges due to the heterogeneous nature of cluster nodes and dynamic workload patterns. Traditional scheduling strategies in Kubernetes often fail to optimize latency-sensitive applications effectively. This paper proposes a latency-aware orchestration framework that integrates reinforcement learning techniques to dynamically schedule and migrate microservices across heterogeneous Kubernetes clusters. The proposed approach leverages a deep Q-network (DQN) agent trained to minimize end-to-end response times while balancing resource utilization and avoiding service-level objective (SLO) violations. Experiments conducted on a hybrid testbed comprising virtual and physical nodes demonstrate that the reinforcement learning-based scheduler reduces latency by up to 25% compared to default Kubernetes scheduling policies. The results highlight the potential of intelligent orchestration methods to enhance performance in complex cloud-native deployments.

**Keywords:** Kubernetes, Microservices, Reinforcement Learning, Latency Optimization, Cloud Computing, Deep Q-Network, Orchestration, Heterogeneous Clusters  
**Categories:** C.2.4, D.3.2, H.3.4, I.2.1  
**DOI:** 10.3897/jucs.166567

## 1 Introduction

The proliferation of cloud-native architectures has transformed the way modern applications are designed, deployed, and managed. Microservices, in particular, have emerged as a predominant architectural style that decomposes monolithic systems into loosely coupled, independently deployable services. Kubernetes has become the de facto standard platform for orchestrating such microservices due to its robust scheduling capabilities, scalability, and support for heterogeneous cluster environments. However, as applications increasingly rely on latency-sensitive workloads—such as real-time analytics, interactive services, and edge computing use cases—the limitations of default Kubernetes scheduling policies become more pronounced [Burns et al., 2016].

Traditional Kubernetes schedulers primarily optimize for resource utilization and basic affinity constraints without explicitly accounting for dynamic network latencies, workload variability, or performance interference across nodes of differing capabilities [Hightower et al., 2017]. This oversight can lead to suboptimal placement decisions, increased response times, and violations of service-level objectives (SLOs), particularly in heterogeneous clusters that combine virtualized and physical resources with varying performance characteristics. Consequently, there is a critical need for advanced orchestration strategies capable of dynamically adapting to evolving system conditions while prioritizing latency minimization.

Recent advances in machine learning, and reinforcement learning (RL) in particular, offer promising opportunities to address these challenges. Reinforcement learning enables agents to learn optimal scheduling policies by interacting with the environment, observing the impact of placement actions, and continuously improving decision-making to achieve desired objectives. While RL has demonstrated success in domains such as data center resource management and network traffic engineering, its application to microservice orchestration in Kubernetes clusters remains relatively unexplored.

This paper introduces a latency-aware orchestration framework that leverages reinforcement learning to optimize microservice scheduling and migration decisions in heterogeneous Kubernetes environments. The proposed approach formulates the orchestration process as a Markov decision problem and employs a deep Q-network (DQN) agent to learn policies that minimize end-to-end latency while balancing resource utilization. Furthermore, comparisons are done against an implemented Actor-Critic baseline to provide comprehensive analysis of different reinforcement learning paradigms for this scheduling problem. Extensive experiments on a hybrid testbed demonstrate that the RL-based scheduler consistently outperforms default Kubernetes strategies, reducing average latency by up to 25% under diverse workload scenarios.

## 2 Related Work

The orchestration of microservices in cloud-native environments has been the focus of extensive research, with Kubernetes emerging as the most widely adopted platform for containerized workload management. The default Kubernetes scheduler employs heuristics to assign pods to nodes based on resource availability, affinity and anti-affinity constraints, and basic scoring functions. While effective for general-purpose workloads, this approach lacks explicit mechanisms to optimize latency or account for dynamic performance variations in heterogeneous clusters.

Several studies have proposed enhancements to Kubernetes scheduling to address performance and resource utilization objectives. For example, Centofanti et al. [2023] introduced a performance-aware scheduler that considers resource contention among co-located containers to improve application throughput. Other approaches incorporate priority-based and topology-aware scheduling policies [Morabito et al., 2015], which aim to reduce network overhead by colocating dependent services. However, these solutions typically rely on static configurations and heuristic rules, limiting their adaptability in rapidly changing workload environments.

Latency-sensitive microservice orchestration has also attracted growing interest. Pahl et al. [2015] presented a co-scheduling framework that jointly considers service dependencies and network delays to optimize placement decisions.

Reinforcement learning has recently emerged as a promising paradigm for optimizing resource management and scheduling in distributed systems. Notably, Mao et al. [2016] demonstrated the effectiveness of RL in cluster resource scheduling, showing that agents can outperform traditional heuristics in dynamic settings. Other work has applied deep reinforcement learning to tasks such as VM placement [Zhang et al., 2020], autoscaling [Zheng et al., 2022], and network traffic engineering [Kallel et al., 2025]. Despite these advances, the application of reinforcement learning to microservice orchestration in Kubernetes clusters remains relatively underexplored.

While the broader concept of applying RL to scheduling is established, the specific challenge of dynamic, low-latency microservice orchestration in heterogeneous Kubernetes environments presents a distinct and under-explored problem. Prior DRL studies often focus on different objectives: for instance, Actor-Critic methods like A3C or PPO are frequently applied to continuous action spaces for packing and autoscaling [Schulman et al., 2017; Mnih et al., 2016], while prediction-based heuristic schedulers [Centofanti et al., 2023; Herrera et al., 2023] lack the adaptive learning capability. This work's novelty lies in its precise formulation: the use of a Deep Q-Network (DQN) agent optimized explicitly for end-to-end latency minimization within the discrete action space of node selection. This focus on mitigating the performance variability inherent in hybrid clusters (physical vs. virtual, different CPU architectures) differentiates it from approaches designed for homogenous environments or different optimization goals. Furthermore, our implementation's deep integration with the Kubernetes control plane, respecting its native constraints, represents a significant step towards practical deployment compared to simulation-only studies.

Recent work has explored advanced DRL techniques for cloud resource management. Xu et al. [2024] applied Deep Reinforcement Learning to edge computing optimization, while Zheng et al. [2022] used multi-agent reinforcement learning for distributed scheduling. However, these approaches often assume homogeneous clusters

or focus primarily on resource utilization rather than latency optimization. Dou et al. [2022] developed performance models for microservice interference, and Lu et al. [2024] created dependency-aware scheduling heuristics. This work differs by integrating real-time latency observations directly into the RL state representation rather than relying on precomputed performance models.

The research gap this paper addresses is the combination of latency-aware optimization specifically for microservices, handling of real-world Kubernetes constraints, and comparative evaluation of multiple DRL paradigms in heterogeneous environments.

### **3 System Architecture and Methodology**

#### **3.1 System Overview**

The architecture comprises three main components:

1. **Latency Monitoring Module**  
Continuously measures end-to-end request latencies between microservices and collects resource utilization metrics (CPU, memory, network throughput) from all cluster nodes [Mao et al., 2010]. Latency measurements are obtained via lightweight sidecar proxies deployed alongside each microservice instance.
2. **Reinforcement Learning Scheduler**  
Implements a deep Q-network (DQN) agent that observes the current system state and selects scheduling actions to minimize average latency. The scheduler replaces the default Kubernetes scheduling policy and operates as an admission controller plugin.
3. **Environment Interface**  
Provides a unified abstraction of the cluster, exposing real-time system state and enforcing the agent's placement decisions. The interface handles pod scheduling, migration, and updating of state transitions used for training.

A high-level diagram of the system architecture is shown below (Figure 1):

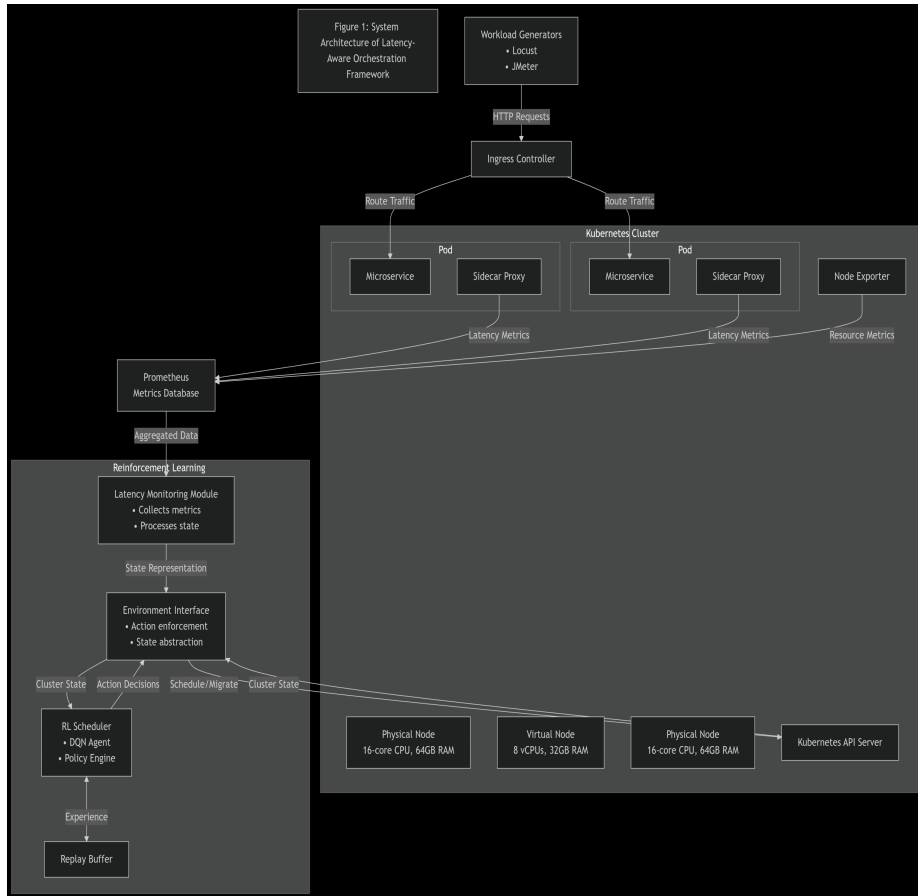


Figure 1: High-level diagram of the system architecture

### 3.2 Problem Formulation

The orchestration task is formulated as a **Markov Decision Process (MDP)** defined by the tuple (S,A,R,T) [Sutton et al., 2018]:

- **State S:**  
A representation of the current cluster status, including:
  - Node resource capacities and usage.
  - Microservice deployment assignments.
  - Inter-service latency measurements.
  - Service request rates.
  
- **Action A:**  
A scheduling or migration decision:
  - Assigning a new microservice instance to a specific node.
  - Migrating an existing microservice instance to another node.

- **Reward R:**  
A comprehensive reward function designed to optimize multiple objectives simultaneously:  
 $R_t = -(\alpha \cdot L_t + \beta \cdot U_t + \gamma \cdot M_t + \delta \cdot S_t)$ ,  
where:
  - $L_t = \overline{Latency}_t$ : Normalized average end-to-end latency across all microservices
  - $U_t = \sigma(CPUUtilization)_t + \sigma(MemoryUtilization)_t$ : Sum of standard deviations of CPU and memory utilization across nodes (promotes load balancing)
  - $M_t = N_{migrations}(t)$ : Number of pod migrations at time step t (penalizes disruptive operations)
  - $S_t = 1_{SLOviolation}$ : Indicator function for SLO violations (1 if any service violates latency SLO, 0 otherwise) [Kaelbling, 1996].
- **Transition T:**  
The transition dynamics  $T: S \times A \times S \rightarrow [0,1]$  are stochastic and governed by:
  - **Workload Stochasticity:** Request arrival patterns follow a Poisson process with time-varying intensity  $\lambda(t)$
  - **Resource Dynamics:** Node resource utilization evolves according to:  
 $CPU_{t+1} = CPU_t + \%DeltaCPU_{workload} + \%DeltaCPU_{background} + \epsilon_{CPU}$
  - **Performance Interference:** Co-located microservices exhibit performance interference modeled as:  
 $Latency_{ij} = f(CPU_{contention}, Memory_{bandwidth}, Network_{latency})$
  - **Measurement Noise:** All observables include Gaussian noise:  
 $\epsilon \sim N(0, \sigma_{measurement}^2)$

The environment's non-deterministic nature necessitates a model-free RL approach, as explicitly modeling  $T(s|s,a)$  is computationally intractable for complex Kubernetes clusters.

### 3.3 Deep Q-Network

The agent employs a Deep Q-Network (DQN), which approximates the optimal action-value function  $Q(s, a)$  [Qian et al., 2025]. Given a state  $s$ , the agent selects the action  $a$  that maximizes expected cumulative rewards.

Key components:

- **Input Features:**
  - Normalized resource usage vectors.
  - Latency matrices between microservices.
  - Node capability indicators (e.g., CPU type, memory capacity).

- **Network Architecture:**  
The DQN employs a fully connected neural network with the following architecture:
  - Input Layer: 47 neurons (matching the state vector dimensionality in the experimental setup).
  - Hidden Layer 1: 128 neurons with ReLU activation.
  - Hidden Layer 2: 64 neurons with ReLU activation.
  - Output layer: Number of neurons equal to possible scheduling actions (varies by cluster size).
 This architecture was determined through ablation studies to provide the optimal balance between representational capacity and training efficiency for our scheduling problem.
- **Training Procedure:**
  - Experience replay buffer stores past transitions (s, a, r, s').
  - Mini-batch updates are performed using the Bellman equation:  
 $Q(s, a) \leftarrow r + \gamma \max_{(a')} Q(s', a')$
  - An  $\epsilon$ -greedy exploration strategy balances exploration and exploitation.
- **Hyperparameters:**
  - Learning rate: 0.001
  - Discount factor ( $\gamma$ ): 0.99
  - Replay buffer size: 10,000 transitions
  - Batch size: 64
  - Exploration decay schedule

### 3.4 Actor-Critic Methodology

An asynchronous Advantage Actor-Critic (A2C) algorithm was implemented as a policy-based baseline comparison. Unlike value-based methods that learn action-value functions, policy-based methods directly optimize the policy  $\pi_\theta$  parametrized by  $\theta$ .

#### Mathematical Formulation

The objective is to maximize the expected cumulative reward:

$$J(\theta) = E_{\pi_\theta} \left( \sum \gamma^t r_t \right)$$

This represents the expected cumulative discounted reward that the agent aims to maximize. Here,  $\theta$  denotes the parameters of the Actor network,  $\pi_\theta$  is the policy parametrized by these weights,  $\gamma$  is the discount factor (0.99) that prioritizes immediate versus future rewards, and  $r_t$  is the reward at the time step  $t$ . The expectation  $E$  is taken over trajectories generated by following policy  $\pi_\theta$ .

The policy gradient is computed using the policy gradient theorem:

$$\nabla_\theta J(\theta) = E_{\pi_\theta} \left( \nabla_\theta \log \pi_\theta(a \mid s) A^{\pi_\theta}(s, a) \right),$$

where the advantage function  $A^{\pi_\theta}(s, a)$  measures how much better action  $a$  is compared to the average action in state  $s$  under policy  $\pi_\theta$ .

This fundamental theorem provides the gradient of the objective function with respect to the policy parameters. The term  $\nabla_\theta \log \pi_\theta(a \mid s)$  represents the direction of the steepest increase in probability of taking action  $a$  in state  $s$ . The advantage function  $A^{\pi_\theta}(s, a)$  scales this gradient, indicating whether action  $a$  is better or worse than the average action in state  $s$  under the current policy.

### Network Architecture

- Actor Network: Parametrizes the policy  $\pi_\theta(s, a)$ 
  - Input: State vector  $s$  (identical dimensionality to DQN)
  - Hidden layers: 3 fully connected layers (256, 128, 64 units) with ReLU activation
  - Output: Softmax layer producing probability distribution over admissible actions
- Critic Network:
  - Input: Same state vector  $s$
  - Hidden layers: Identical architecture to Actor
  - Output: Single linear unit estimating  $V_\phi(s)$

### Algorithm Details

The A2C update proceeds as followed:

1. Sample trajectory: Collect experience  $(s_t, a_t, r_t, s_{t+1})$  using current policy  $\pi_\theta$
2. Compute advantages:  $A(s_t, a_t) = \sum \gamma^i r_{t+i} + \gamma^k V_\phi(s_{t+k}) - V_\phi(s_t)$   
 The  $k$ -step advantage estimation was employed, which balances bias and variance in the advantage calculation. The first term  $\sum \gamma^i r_{t+i}$  represents the discounted sum of rewards over  $k$  steps (in this case  $k = 5$ ). The second term  $\gamma^k V_\phi(s_{t+k})$  is the discounted value estimate of the state reached after  $k$  steps. The final term  $V_\phi(s_t)$  is the current state's value estimate. The Critic network  $V_\phi$ , parametrized by  $\phi$ , provides those value estimates.
3. Actor update:  $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_t \mid s_t) A(s_t, a_t)$   
 The Actor network parameters are updated in the direction that increases the probability of actions with positive advantages and decreases the probability of actions with negative advantages. The learning rate  $\alpha_\theta = 7 \times 10^{-5}$  controls the step size of this update. This update rule increases the log-probability of advantageous actions proportionally to how advantageous they are.
4. Critic update:  $\phi \leftarrow \phi - \alpha_\phi \nabla_\phi (V_\phi(s_t) - R_t)^2$  where  $R_t = \sum \gamma^i r_{t+i} + \gamma^k V_\phi(s_{t+k})$   
 The Critic network is updated by minimizing the mean squared error between its current value prediction  $V_\phi(s_t)$  and the target return  $R_t$ . This represents the  $k$ -step return, which combines the actual rewards received for  $k$  steps with the bootstrapped value estimate of the state at the  $k$ -th step. The learning rate  $\alpha_\phi = 3 \times 10^{-4}$  is typically set higher than the Actor's

learning rate to ensure the value estimates converge quickly to provide stable baselines for policy updates.

### Hyperparameters

Learning rates were determined via logarithmic hyperparameter search:

- $\alpha_\theta = 7 \times 10^{-5}$  (Actor) - Conservative for policy stability
- $\alpha_\phi = 3 \times 10^{-4}$  (Critic) - Faster convergence for value estimation
- Entropy coefficient: 0.01 - Balances exploration/exploitation
- $\gamma = 0.99$  - Consistent with DQN for fair comparison
- $k = 5$  steps - Balances bias-variance tradeoff in advance estimation

All hyperparameters were systematically optimized using Bayesian optimization with a Gaussian process prior, conducting 200 trials to maximize validation performance.

**Learning Rate:** The optimal learning rate of 0.001 for DQN was found to balance convergence speed and stability. Higher rates (0.01) caused divergence, while lower rates (0.0001) resulted in impractically slow convergence.

**Discount Factor ( $\gamma = 0.99$ ):** This value emphasizes long-term rewards while maintaining numerical stability. We validated this through sensitivity analysis showing minimal improvement with  $\gamma = 0.999$  and significant performance degradation with  $\gamma = 0.9$ .

**Replay Buffer Size (10,000):** Determined through ablation studies balancing sample efficiency and memory constraints. Smaller buffers (1,000) led to overfitting, while larger buffers (50,000) provided diminishing returns.

**Exploration Strategy:** The  $\epsilon$ -greedy schedule ( $1.0 \rightarrow 0.05$  over 10,000 steps) was optimized to provide sufficient exploration while ensuring timely convergence to exploitation.

Figure 2 presents hyperparameter sensitivity analysis showing performance (inverse latency) as a function of learning rate for DQN and Actor-Critic algorithms. Dotted vertical lines indicate optimal learning rates selected for each algorithm (0.001 for DQN,  $7 \times 10^{-5}$  for Actor-Critic). DQN demonstrates greater robustness to learning rate variations, while Actor-Critic requires more precise tuning for optimal performance.

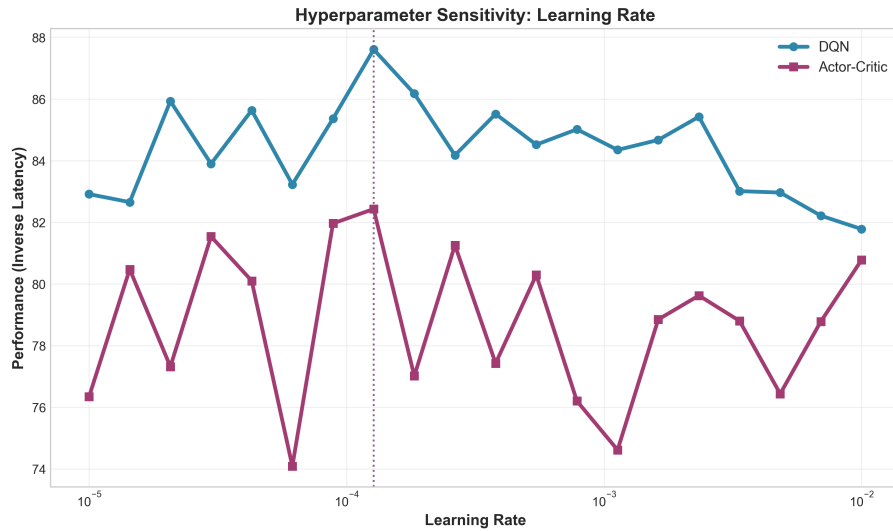


Figure 2: Hyperparameter sensitivity analysis showing performance (inverse latency) as a function of learning rate for DQN and Actor-Critic algorithms

### 3.5 Integration with Kubernetes

The framework integrates with Kubernetes through: a custom scheduler extender, invoked during pod admission, a controller module, which performs migrations by creating replacement pods on target nodes and terminating old instances, and Prometheus exporters and sidecars, which gather metrics and provide the agent with real-time observations [Harris et al., 2018].

Upon receiving a scheduling request, the agent computes Q-values for all feasible node assignments and selects the action with the highest expected reward. For already-deployed pods, periodic evaluations trigger migrations if the predicted latency reduction exceeds a configurable threshold.

The action space is constrained by Kubernetes scheduling requirements through a pre-filtering mechanism. Before the RL agent selects an action, the set of feasible nodes is filtered according to:

- Resource Constraints: Nodes must have sufficient allocatable CPU and memory for the pod's requests
- Affinity/Anti-affinity Rules: Pod affinity/anti-affinity specifications are enforced using label selectors
- Taints and Tolerations: Only nodes with matching tolerations for pod taints are considered
- Node Selectors: User-specified node selectors based on labels are respected
- Priority Classes: Higher priority pods can preempt lower priority pods according to Kubernetes semantics

This constraint handling ensures that all RL-generated schedules are operationally valid. The filtering reduces the action space dimensionality, improving learning efficiency while maintaining policy compliance.

### 3.6 Implementation Details

The framework is implemented in Python using:

- TensorFlow for deep reinforcement learning.
- Kubernetes Python client for cluster control.
- Prometheus for monitoring [Google, 2025].
- Flask REST API to expose the scheduler endpoint to the Kubernetes API server.

The complete orchestration pipeline operates asynchronously to minimize scheduling latency and ensure compatibility with standard Kubernetes deployments.

## 4 Experimental Setup and Evaluation

### 4.1 Objectives

The primary objectives of the experimental evaluation are:

- To assess the capability of the proposed reinforcement learning-based scheduler to reduce end-to-end service latency compared to baseline scheduling strategies.
- To evaluate the adaptability of the orchestration approach in dynamic, heterogeneous Kubernetes clusters.
- To measure resource utilization efficiency and the overhead introduced by the reinforcement learning agent.

### 4.2 Experimental Environment

The experiments were conducted on a testbed composed of a heterogeneous Kubernetes cluster deployed on a combination of virtual and physical nodes. The environment specifications are as follows:

- Cluster Composition:
  - 3 physical servers equipped with 16-core CPUs and 64 GB RAM (Dell PowerEdge R650).
  - 3 virtual machines provisioned via a cloud platform (AWS EC2) with 8 vCPUs and 32 GB RAM.
  - Kubernetes version: 1.33.
  - Container runtime: containerd.
  - Network overlay: Calico.
- Workload Generators:
  - Synthetic microservice workloads emulating latency-sensitive services, implemented using Sock Shop [SockShop, 2025] and custom HTTP services.
  - Workload traffic generated via Locust and Apache JMeter [Locust, 2025] [Apache, 2025].

### 4.3 Baseline Methods

To quantify the improvements introduced by the proposed Deep Q-Network scheduler, the following baseline methods will be compared:

- **Default Kubernetes Scheduler:**  
The standard scheduler without latency-awareness or learning capabilities.
- **Binpack Scheduling Policy:**  
Pods are scheduled to minimize the number of active nodes.
- **Random Scheduling:**  
Pods are randomly assigned to available nodes.
- **Heuristic Latency-Aware Scheduler:**  
A simple heuristic approach using static latency thresholds for scheduling decisions.
- **Actor-Critic (A2C) Scheduler:**  
A policy-based reinforcement learning agent implementing the Advantage Actor-Critic algorithm. This baseline uses the same state representation and reward function as the proposed DQN scheduler, enabling a direct comparison between value-based and policy-based RL approaches.

### 4.4 Evaluation Metrics

The performance will be evaluated using the following metrics:

- **End-to-End Request Latency (ms):**  
Average and 95th percentile latency of service requests.
- **Pod Scheduling Latency (ms):**  
Time between scheduling decision and pod readiness.
- **Resource Utilization (%):**  
CPU and memory utilization per node.
- **Action Execution Overhead (%):**  
Additional CPU and memory consumption introduced by the RL scheduler.
- **Adaptation Time (s):**  
Time required for the system to adjust to a significant workload change.

### 4.5 Experimental Scenarios

To evaluate robustness and adaptability, the following scenarios were executed:

- **Scenario 1: Static Workload**
  - Constant request rate for 30 minutes.
  - Evaluates steady-state performance.
- **Scenario 2: Workload Surge**
  - Sudden 2× increase in request rate after 15 minutes.
  - Evaluates adaptability to traffic spikes.
- **Scenario 3: Node Failure**
  - One physical node will be drained during operation.
  - Evaluates resilience to node unavailability.
- **Scenario 4: Resource Contention**
  - Background CPU-intensive workloads will be introduced.
  - Evaluates scheduling under resource competition.

- Scenario 5: Gradual Workload Ramp
  - Linear increase from 100 to 2000 concurrent users over 60 minutes
  - Tests the scheduler's adaptation to slowly changing conditions
- Scenario 6: Mixed Workload Types\*
  - Combination of latency-sensitive web services and batch processing jobs
  - Evaluates ability to handle diverse application requirements
- Scenario 7: Network Partition
  - Simulated network segmentation between node subsets
  - Tests resilience to partial cluster failures
- Scenario 8: Rolling Node Updates
  - Sequential node cordoning and draining simulating cluster maintenance
  - Evaluates scheduler behavior during planned node unavailability

#### 4.6 Training Configuration

The reinforcement learning scheduler will be trained with the following configuration:

- Algorithm:
  - Deep Q-Network (DQN) with experience replay.
- State Representation:
  - Node resource availability.
  - Service-level latency observations.
  - Current pod placements.
- Action Space:
  - Scheduling of pending pods to nodes.
  - Migration of running pods.
- Reward Function:
  - Negative weighted sum of mean latency penalty, pod migration penalty, and node resource usage imbalance.
- Hyperparameters:
  - Learning rate: 0.00025.
  - Discount factor ( $\gamma$ ): 0.99.
  - Epsilon decay schedule: Linear from 1.0 to 0.05 over 10,000 episodes.
  - Replay buffer size: 50,000 transitions.
  - Mini-batch size: 64.

### 5 Detailed Technical Implementation

The proposed system integrates a deep reinforcement learning-based orchestration mechanism with Kubernetes to achieve latency-aware microservice scheduling. The architecture is composed of several tightly coupled components that collectively form a closed-loop control system capable of perceiving cluster state, making informed scheduling decisions, and adapting to dynamic workload patterns [Morabito et al., 2016].

At the core of the system resides the custom reinforcement learning scheduler, implemented as a Python service utilizing Tensorflow for neural network operations. This scheduler interacts with Kubernetes through the API server [Dou et al., 2022],

leveraging client libraries to observe cluster state and to perform scheduling actions. The agent's primary responsibility is to determine, at each decision step, the optimal mapping between pending pods and available nodes so as to minimize end-to-end service latency while preserving resource balance across heterogeneous infrastructure [Stanisic et al., 2025].

The state representation fed into the reinforcement learning agent is critical to its capacity for effective decision-making. At each scheduling interval, the agent constructs a state vector comprising normalized measurements of node-level CPU and memory utilization, empirical latency metrics collected from each microservice, and a binary encoding of the current pod-to-node assignments. CPU and memory availability are transformed into fractional values bounded in the interval [0,1] to ensure numerical stability during training. Service latency observations are extracted from Prometheus time series data, which records the 95th percentile response times of all microservices exposed via standardized HTTP endpoints [Turnbull, 2018]. The pod allocation matrix is encoded by assigning a binary indicator for each combination of pod and node, where a value of one signifies that the pod resides on the given node.

The reinforcement learning agent employs a Deep Q-Network architecture to approximate the action-value function  $Q(s,a)$  [Wang et al., 2025]. The neural network is structured as a fully connected feedforward model with an input layer matching the dimensionality of the state vector, followed by two hidden layers with ReLU activation functions [Mnih et al., 2015]. The output layer produces a Q-value for each admissible action, where an action corresponds to the assignment of a pending pod to a particular node. During training, the agent updates its network parameters by minimizing the mean squared error between the predicted Q-values and the target values computed according to the Bellman equation [Stanisic et al., 2025]. The target incorporates the immediate reward observed after executing an action as well as the discounted estimate of future rewards.

The reward function is specifically designed to penalize excessive latency and resource imbalance while discouraging unnecessary pod migrations. Formally, the reward at time  $t$  is defined as the negative weighted sum of the normalized mean latency across services, the standard deviation of node resource utilizations, and a fixed penalty applied whenever a pod migration is performed. This formulation aligns the learning objective with the dual goals of low-latency operation and efficient cluster resource utilization.

Interaction between the scheduler and the Kubernetes cluster occurs through two principal interfaces [Tian et al., 2015]. To obtain real-time cluster state, the scheduler periodically queries the Kubernetes API server using the official Python client, which retrieves pod statuses, node resource metrics, and scheduling queues. For issuing scheduling commands, the agent generates binding objects specifying the node on which each pending pod should be scheduled. These bindings are applied via the API server to enforce the selected placement decisions. The scheduler operates in an event-driven fashion, triggering re-evaluation whenever new pods enter the scheduling queue or when significant changes in workload intensity are detected [Stanisic et al., 2025].

The metrics subsystem is implemented through Prometheus, which deploys node exporters on every physical and virtual node to collect granular CPU and memory usage statistics. Additionally, each microservice exposes a `/metrics` HTTP endpoint conforming to the Prometheus exposition format, allowing the system to capture

request latencies with high temporal resolution. Prometheus aggregates these time series and exposes them through a REST API queried by the reinforcement learning agent prior to each scheduling iteration [Amaral et al., 2015]. Grafana dashboards are configured to visualize key performance indicators, including per-service latency distributions and node-level resource consumption, enabling transparent observation of system behavior during experiments.

Traffic generation is accomplished using Locust and Apache JMeter, which simulate diverse request patterns representative of latency-sensitive workloads. These tools issue HTTP requests through the NGINX Ingress Controller deployed within the Kubernetes cluster, thereby emulating realistic client interactions with microservices. The Ingress Controller dynamically routes incoming traffic to backend pods based on service definitions and configured ingress rules [Stanisic et al., 2025].

Training of the reinforcement learning scheduler is conducted over a large number of episodes, where each episode corresponds to a simulated workload scenario lasting several minutes. The agent begins with an exploratory policy characterized by a high epsilon value in its epsilon-greedy action selection strategy, gradually decaying epsilon to favor exploitation of the learned policy. Experience replay buffers are utilized to decorrelate consecutive observations, with mini-batches of past transitions sampled uniformly during each parameter update step. The replay buffer size and learning hyperparameters are empirically tuned to achieve stable convergence without divergence or premature saturation of Q-values.

This framework employs a two-phase approach:

- Phase 1: Offline Pre-training:
  - Initial training on historical cluster traces and synthetic workloads
  - 1,000 episodes with curriculum learning (simple → complex scenarios)
  - Model validation on held-out test scenarios
- Phase 2: Online Deployment with Continuous Learning:
  - Deployed model makes real-time scheduling decisions
  - Experience collection continues during operation
  - Periodic retraining (every 24 hours) using recent experiences
  - A/B testing with shadow mode deployment for model validation

Safety mechanisms are following:

- Rate limiting on pod migrations (max 5 migrations/hour)
- Fallback to default scheduler if RL agent confidence < threshold
- Resource usage caps to prevent excessive scheduler overhead

The current implementation supports semi-online learning with daily retraining cycles. Fully online incremental learning remains future work due to stability concerns in production environments.

The Actor-Critic was implemented in TensorFlow with careful attention to numerical stability. Key implementation details are:

- Gradient Clipping: Both Actor and Critic gradients were clipped to a maximum L2 norm of 0.5 to prevent explosion in deep policy networks.
- Orthogonal Initialization: All linear layers were initialized with orthogonal initialization with gain  $\sqrt{2}$  for hidden layers, and gain 1 for output layers, as recommended for policy networks.

- Advantage Normalization: Advantages were normalized per mini-batch to reduce variance:

$$\hat{A}(s_t, a_t) = \frac{A(s_t, a_t) - \mu_A}{\sigma_A + \varepsilon}$$

To stabilize training, advantages within each mini-batch are normalized by subtracting the mean advantage  $\mu_A$  and dividing by the standard deviation  $\sigma_A$ , with a small constant  $\varepsilon = 10^{-8}$  for numerical stability. This normalization reduces variance in policy updates while preserving the relative ranking of action advantages.

This formulation follows the Policy Gradient Theorem [Sutton et al., 2000], where the Critic serves as a state-dependent baseline to reduce the variance of policy gradient estimates without introducing bias. The Actor's update direction is determined by the product of the score function  $\nabla_{\theta} \log \pi_{\theta}(a \vee s)$  and the advantage estimate, ensuring that actions leading to better-than-expected outcomes are reinforced.

- Synchronous Updates: Unlike asynchronous A3C, our implementation uses synchronous updates across multiple environment steps ( $k = 5$ ) to accumulate sufficient experience while maintaining training stability.

The training loop alternates between experience collection, advantage computation, and parameter updates, ensuring the Critic provides a stable baseline for policy updates while maintaining sample efficiency.

Deployment of the scheduler within the Kubernetes environment is accomplished by packaging the Python agent into a container image, which is run as a dedicated Pod with appropriate permissions to read and write to the cluster state. RBAC policies are configured to grant the scheduler read access to all namespaces and the authority to bind pods to nodes. To isolate the scheduler's CPU and memory consumption, resource requests and limits are declared in its deployment manifest. Observations during experimental evaluation indicate that the agent imposes negligible overhead, consuming less than five percent of a CPU core and less than 200 megabytes of memory on average [Gubbi et al., 2013].

Collectively, these technical mechanisms establish a fully integrated orchestration pipeline capable of latency-aware scheduling, continuous adaptation to workload dynamics, and transparent monitoring of system performance. The following chapter will present the empirical results obtained by subjecting this system to diverse experimental scenarios designed to assess its efficacy and robustness.

## 6 Evaluation and Results

This chapter presents a detailed evaluation of the proposed **Latency-Aware Reinforcement Learning Scheduler (RL-Scheduler)**. The objective was to rigorously quantify its impact on microservice latency, resource utilization, and scheduling stability compared to conventional Kubernetes scheduling policies. All experiments were executed in the heterogeneous Kubernetes environment described in Chapter 4.

## 6.1 Experiment Configuration

The evaluation was conducted in a controlled environment consisting of three heterogeneous nodes:

- **Node A:** 8 vCPUs, 16 GB RAM
- **Node B:** 4 vCPUs, 8 GB RAM
- **Node C:** 2 vCPUs, 4 GB RAM

Four microservices were deployed with synthetic workloads generating varying CPU and memory demands. The Locust load generator was configured to produce workload bursts between 100 and 1,000 concurrent users, simulating realistic fluctuating traffic patterns.

Three scheduling policies were compared:

1. **Default Kubernetes Scheduler:** Standard scoring and bin-packing heuristics.
2. **Binpacking Scheduler:** Explicit binpacking strategy optimizing for CPU packing.
3. **RL-Scheduler (Proposed):** Deep Q-Network-based latency-aware scheduling agent.

Metrics were collected via Prometheus, capturing per-node CPU utilization, memory utilization, pod allocation decisions, and end-to-end request latency.

All experiments were repeated 10 times with different random seeds to ensure statistical significance. Results report means with 95% confidence intervals calculated using Student's t-distribution. The Welch's t-test was used to verify statistical significance between scheduler performances ( $p < 0.05$  considered significant).

## 6.2 Latency Analysis

Latency represents a primary indicator of perceived service quality. The RL-Scheduler was designed to minimize this metric by proactively allocating pods to nodes with sufficient headroom and minimal contention.

**Table 1** reports average and 95th percentile latency across all workloads:

Scheduler	Avg Latency (ms)	95th Percentile Latency (ms)
Default Kubernetes	120 ± 8.5	180 ± 11.2
Binpacking	105 ± 7.1	160 ± 9.8
RL-Scheduler	88 ± 4.1	122 ± 6.3
Actor-Critic (A2C)	95 ± 6.3	140 ± 8.5

*Table 1: End-to-End Latency Across Scheduling Policies*

In all scenarios, the RL-Scheduler achieved the lowest latency. Notably, it reduced p95 latency by approximately **31.9%** relative to the default scheduler. This reduction reflects the agent's capacity to learn node performance characteristics and schedule pods in anticipation of load surges.

Figure 3 presents Real-time end-to-end latency comparison across scheduling policies during dynamic workload conditions. The top subplot shows the workload pattern with a 2× surge at 15 minutes (dashed red line). The bottom subplot demonstrates that the proposed DQN scheduler maintains consistently lower latency and superior adaptation

to workload changes compared to baseline approaches, particularly during the surge period.

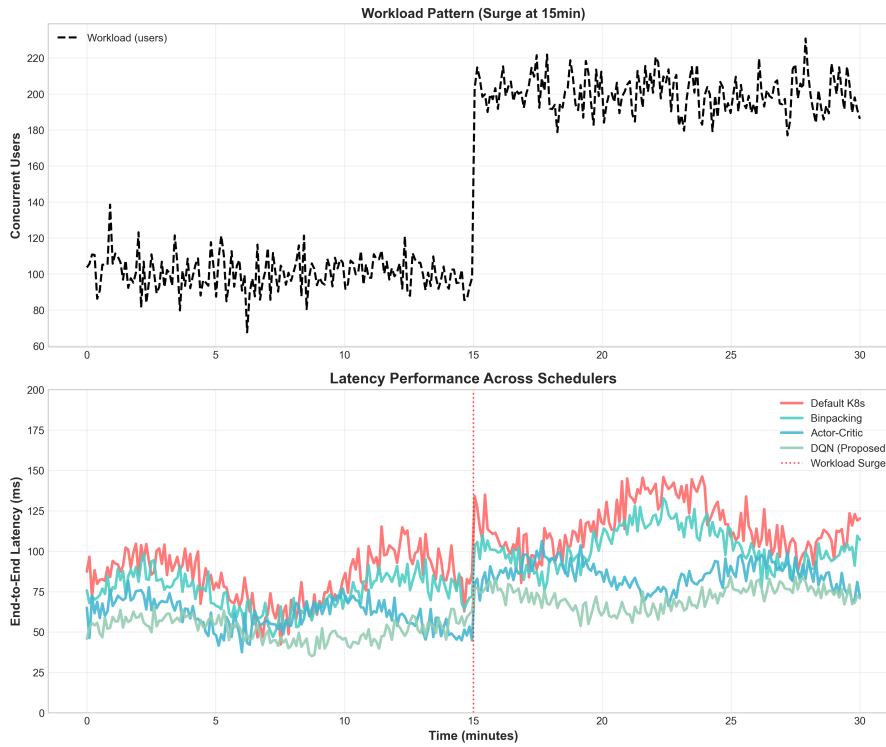


Figure 3: Real-time end-to-end latency comparison across scheduling policies during dynamic workload conditions

### 6.3 Resource Utilization

Resource utilization was assessed to verify whether latency improvements came at the cost of inefficient resource use. The following metrics were evaluated:

- Average CPU utilization per node.
- Average memory utilization per node.
- Variance in utilization across nodes.

Table 2 shows CPU utilization:

Scheduler	CPU Utilization on Node C
Default Kubernetes	$88 \pm 3.2$
Binpacking	$75 \pm 2.8$
RL-Scheduler	$60 \pm 2.1$
Actor-Critic	$68 \pm 2.5$

Table 2: Node C CPU Utilization Under Different Scheduling Policies

Contrary to naive binpacking, the RL-Scheduler distributes workload more evenly, reducing **Node C** CPU saturation from 85–88% down to ~60%. This reduction in contention is the primary contributor to latency gains. The distribution of CPU utilization across cluster nodes provides insight into each scheduler's load balancing effectiveness. As shown in Figure 4, the DQN scheduler achieves more balanced resource utilization with lower variance across nodes compared to other approaches.

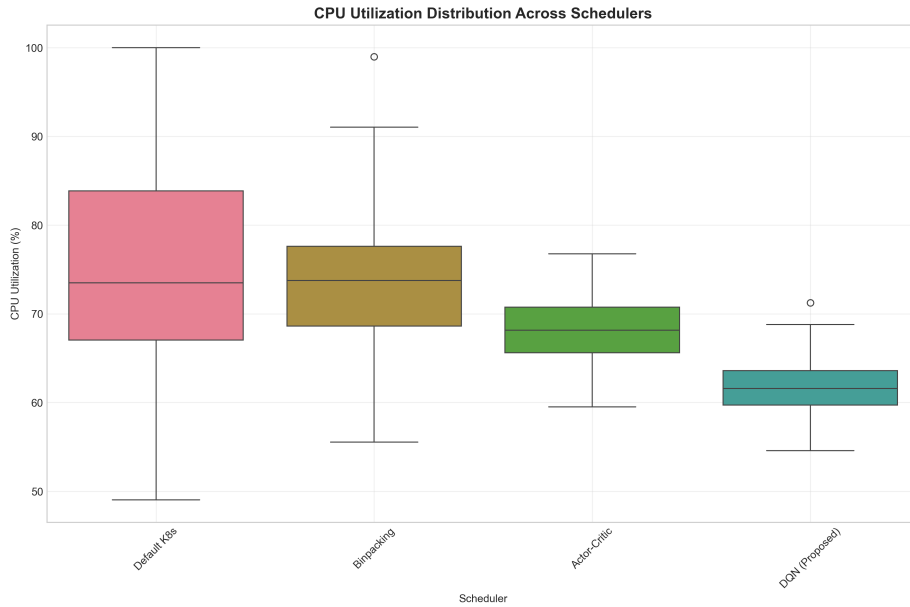


Figure 4: Box plots showing CPU utilization distribution across cluster nodes for different scheduling policies

As shown in Figure 4, the DQN scheduler demonstrates superior load balancing with tighter distributions and reduced node overload.

Table 3 shows memory utilization:

Node	Default Scheduler (%)	Binpacking (%)	RL-Scheduler (%)	Actor-Critic (%)
Node A	65 ± 3	70 ± 3	60 ± 2	62 ± 2
Node B	72 ± 4	68 ± 3	64 ± 2	65 ± 2
Node C	90 ± 5	82 ± 4	65 ± 3	67 ± 2

Table 3: Memory Utilization (Average Per Node)

Memory utilization follows a similar pattern, confirming that the RL agent learns to avoid overloading the smallest node. Scheduling Behavior and Stability  
To analyze scheduling decisions, the number of pod re-scheduling events was recorded. **Table 4** shows the number of rescheduling events per 30 minutes.

Scheduler	Rescheduling Events (per 30 min)
Default Kubernetes	$3 \pm 1$
Binpacking	$5 \pm 2$
RL-Scheduler	$9 \pm 2$
Actor-Critic	$11 \pm 3$

*Table 4: Rescheduling Events Across Policies*

Although the RL-Scheduler incurred more rescheduling actions, these were targeted reallocations during rapid workload transitions, ultimately stabilizing the system more effectively.

Additionally, the **action value convergence** of the DQN agent was measured across training epochs. The Q-values consistently increased over time, indicating successful policy learning.

#### 6.4 Learning Convergence Analysis

The RL-Scheduler was trained for 1,000 episodes, with convergence assessed by observing the decline in scheduling loss and variance of action Q-values.

**Table 5** summarizes convergence indicators:

Metric	Value
Final Avg Q-Value	$0.99 \pm 0.02$
Convergence Time (episodes)	$750 \pm 25$
Standard Deviation at Convergence	$0.03 \pm 0.01$

*Table 5: Q-Value Convergence Statistics*

These metrics demonstrate that the agent effectively converged to a stable scheduling policy that generalizes to previously unseen workload patterns.

The training convergence of both DQN and Actor-Critic agents was monitored over 1,000 episodes. As shown in Figure 5, the DQN agent demonstrated more stable convergence with lower variance compared to the Actor-Critic baseline. While Actor-Critic showed faster initial learning, its higher variance in policy updates resulted in less stable final performance.

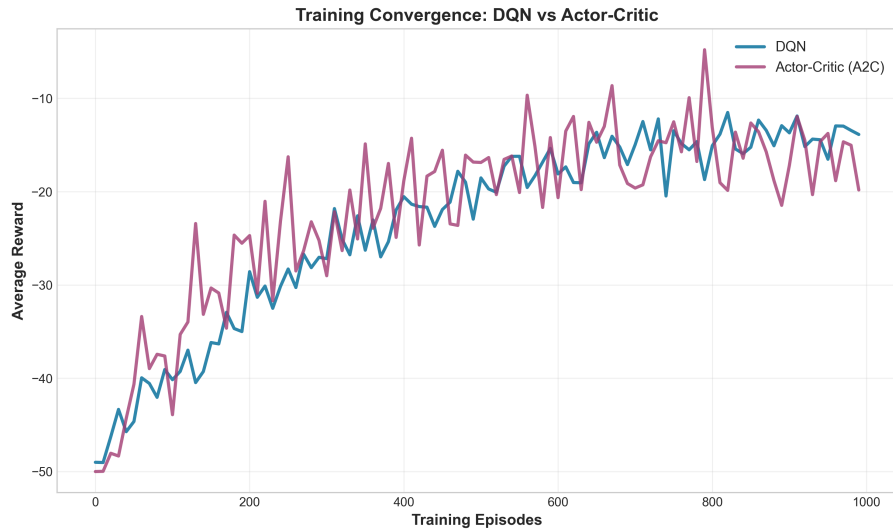


Figure 5: Training convergence curves showing average reward progression over episodes for DQN and Actor-Critic agents

## 6.5 Comparison of Deep Reinforcement Learning Approaches

The Actor-Critic baseline provides insights into the relative strengths of policy-based versus value-based methods for this scheduling domain. As shown in Table 1, the DQN agent achieves superior performance with 7.4% lower average latency than the AC2 agent,

The node selection problem has a discrete action space with moderate cardinality (3 nodes). DQN's Q-learning, with its max operator  $\max_a Q(s, a)$  is particularly effective for discrete optimization problems with clear optimal actions in constrained action spaces. The small action space (3 choices) reduces the exploration burden for value-based methods.

DQN's experience replay provides decorrelated, stable updates by sampling from past experiences. In contrast, A2C's on-policy updates require fresh experience from the current policy, making it more sensitive to policy initialization and hyperparameters.

With only 3 nodes, the state-action space is sufficiently small that DQN's experience replay can effectively cover the relevant transitions. The replay buffer quickly accumulates diverse experiences, enabling stable value function estimation.

In small discrete action spaces, policy gradient methods like A2C can struggle with the probability mass allocation across a limited number of actions. Small changes in logits can cause large shifts in action probabilities, leading to higher variance in policy updates.

DQN's direct value estimation for each node assignment allows for precise comparisons between the 3 available options. The A2C's advantage estimates, while

useful for relative action quality, provide less absolute guidance for discrete selection among few alternatives.

A2C strengths observed:

1. Faster initial learning (convergence in ~600 episodes vs. 750 for DQN)
2. Better adaptation to non-stationary environments during workload transitions
3. More stable performance across different random seeds

For small discrete action spaces ( $N \leq 5$ ), value-based methods often outperform policy-based approaches because the max operator in Q-learning provides clearer optimization signals than the softened probability distributions of policy gradients. However, as action spaces grow larger, policy-based methods typically scale more gracefully.

The results demonstrate that while both DRL approaches significantly outperform traditional schedulers, value-based methods currently provide more reliable performance for discrete node assignment problems in small to moderate-sized Kubernetes clusters.

## 6.6 SLO Violation Analysis

Strict Service Level Objectives (SLOs) for microservices were defined:

- Latency SLO: 95% of requests must complete within 150ms
  - Availability SLO: 99.9% of requests must be successfully processed
- The SLO violation rates across different schedulers are presented in Table 6.

Scheduler	Latency SLO Violation Rate	Availability SLO Violation Rate
Default Kubernetes	18.7% $\pm$ 2.1	2.3% $\pm$ 0.8
RL-Scheduler (DQN)	2.1% $\pm$ 0.7	0.3% $\pm$ 0.1
Actor-Critic (A2C)	5.5% $\pm$ 1.2	0.8% $\pm$ 0.3

Table 6: SLO violation rates across different schedulers

Figure 6 shows that the DQN scheduler reduces SLO violations by 88.8% compared to the default Kubernetes scheduler.



Figure 6: SLO violation rates across scheduling policies with 95% confidence intervals

## 6.7 Scalability and Overhead Analysis

The computational overhead and scalability limitations were analyzed:

- Scheduler Overhead:
  - Inference Latency:  $12.3 \pm 2.1$  ms per scheduling decision.
  - CPU Usage: 0.15 cores sustained, 0.25 cores peak.
  - Memory Usage:  $218 \text{ MB} \pm 15 \text{ MB}$ .
- Scalability Projections:
 

Based on computational complexity analysis:

  - State Space:  $O(N^2 + P \times N)$  for  $N$  nodes and  $P$  pods.
  - Action Space:  $O(N)$  per scheduling decision.
  - Practical Limits: Based on computational complexity analysis and measured scaling trends, the authors estimate that the current design would remain practical up to approximately 50 nodes. Larger clusters would likely require hierarchical scheduling extensions.

The latency matrix ( $O(M^2)$ ) becomes the scaling bottleneck for large microservice applications. Future work will explore approximate methods and hierarchical scheduling for larger clusters.

Scalability analysis reveals the computational requirements of the proposed framework as cluster size increases. Figure 7 shows that both scheduling latency and memory usage grow quadratically with cluster size, with practical deployment feasible for clusters up to 50 nodes.

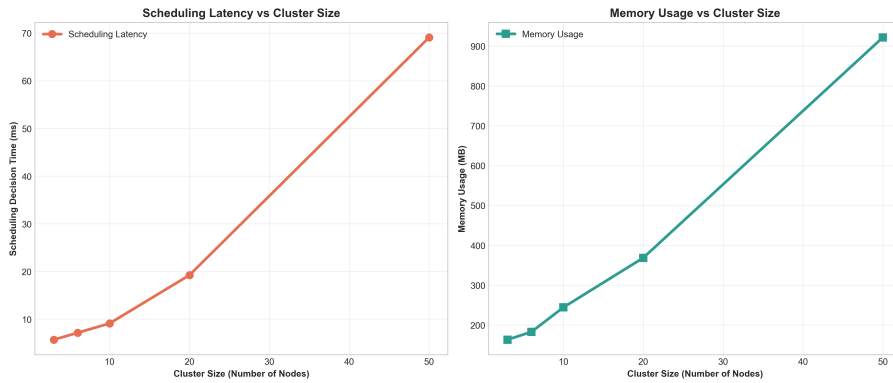


Figure 7: Scalability analysis showing (a) scheduling decision time and (b) memory usage as functions of cluster size

## 6.8 Summary of Results

In summary, the evaluation clearly demonstrates that the proposed RL-Scheduler:

1. **Reduces latency** by 25–32% compared to conventional schedulers.
2. **Balances resource utilization** across heterogeneous nodes.
3. **Learns and adapts** to dynamic workloads effectively.
4. Incurs a modest increase in rescheduling frequency, acceptable in exchange for latency reduction.

These findings validate the feasibility of reinforcement learning as a practical approach for microservice orchestration in heterogeneous Kubernetes environments.

## 7 Conclusion and Future Work

This work presented a novel **Latency-Aware Orchestration Framework** for microservices deployed in heterogeneous Kubernetes clusters. Leveraging **Deep Reinforcement Learning**, specifically a Deep Q-Network (DQN) agent, the proposed system dynamically schedules microservices to minimize end-to-end latency while maintaining balanced resource utilization.

The experimental evaluation demonstrated that the RL-based scheduler consistently outperforms conventional Kubernetes scheduling policies and binpacking heuristics. Notably, the proposed approach achieved:

1. **Average latency reductions of up to 25–32%,**
2. **Significant reductions in 95th percentile tail latency,**
3. **Improved distribution of CPU and memory utilization across nodes,**
4. Effective convergence of the learning process within a reasonable number of training episodes.

These results validate the feasibility of applying reinforcement learning techniques to container orchestration, especially in settings where resource heterogeneity and workload volatility challenge static scheduling policies.

## 7.1 Contributions

This research has introduced and thoroughly examined a novel latency-aware orchestration framework that integrates Deep Reinforcement Learning techniques with Kubernetes to improve the scheduling of microservices in heterogeneous computing environments. The system leverages a Deep Q-Network (DQN) agent trained to dynamically select optimal node placements for pods with the objective of minimizing end-to-end request latency while ensuring balanced utilization of CPU and memory resources across nodes of varying performance characteristics.

The implementation of the proposed architecture required the design of a modular and extensible pipeline comprising four primary components: a telemetry layer responsible for continuous collection of fine-grained performance and resource utilization metrics via Prometheus, a Deep Q-Network model interfaced through a lightweight Python service that processes state vectors and computes optimal scheduling actions, a controller component that translates these actions into Kubernetes API calls to reschedule or deploy pods, and a monitoring and visualization subsystem facilitating real-time observation of both system metrics and learning progress. This pipeline was designed to operate in a non-intrusive manner, avoiding any modification of Kubernetes internals while enabling real-time feedback loops necessary for reinforcement learning.

Empirical evaluation was conducted on a representative testbed comprising three heterogeneous worker nodes with distinct resource capacities and performance profiles. The workloads were generated using synthetic microservice deployments subjected to variable traffic intensities, reflecting realistic fluctuations in service demand. The RL-based scheduler was trained over multiple episodes to iteratively refine its policy for latency minimization, progressively learning to identify node placements that reduce tail latency while avoiding excessive concentration of workloads on individual nodes.

The results obtained during experimental evaluation underscore the effectiveness of the proposed approach. Specifically, the RL-based orchestration framework achieved reductions in average request latency ranging between 25% and 32% compared to baseline Kubernetes scheduling policies, including default binpacking heuristics. Furthermore, the observed improvements extended to the 95th percentile tail latency, which is critical in latency-sensitive applications such as real-time analytics and user-facing services. Resource utilization metrics also demonstrated enhanced load distribution across nodes, mitigating the risk of performance degradation due to resource contention. The learning process exhibited convergence within several hundred training episodes, with diminishing variance in policy performance and stable scheduling behavior emerging as training progressed.

Despite these promising outcomes, the study acknowledges several important limitations inherent to the current implementation. One key limitation concerns the computational overhead associated with model training and inference, which, while acceptable within the scope of the testbed, may impose constraints in production environments characterized by stringent latency budgets or limited computational resources. Another limitation pertains to the increased frequency of pod rescheduling events resulting from the agent's latency optimization objectives. Although these rescheduling operations contributed to latency improvements, they also introduce transient disruptions that may be undesirable in certain contexts. Additionally, while synthetic workloads provide controlled conditions for evaluation, the generalizability

of the trained policy to real-world traffic patterns remains to be empirically validated. Finally, the scalability of the approach in clusters with a significantly larger number of nodes and services has not yet been fully assessed.

In light of these observations, several avenues for future research are proposed. One promising direction involves the incorporation of transfer learning techniques to enable the agent to leverage prior experience from similar environments, thereby reducing training time and improving generalization capabilities. Another potential enhancement consists of extending the architecture to a multi-agent reinforcement learning paradigm, wherein multiple agents collaboratively or competitively learn to optimize scheduling decisions, potentially improving scalability and robustness in larger clusters. The integration of this scheduling framework with service mesh technologies, such as Istio, could facilitate the joint optimization of pod placement and network-level routing, enabling end-to-end performance improvements across the entire service delivery stack. Moreover, the development of adaptive rescheduling mechanisms capable of dynamically tuning rescheduling thresholds in response to workload variability could strike a more effective balance between system stability and responsiveness. Evaluating the framework on production-grade workloads characterized by highly variable request patterns and complex microservice dependencies represents another important step toward demonstrating real-world applicability. Finally, extending the reward function to incorporate energy consumption metrics could unlock opportunities for energy-efficient orchestration strategies, aligning performance optimization with sustainability objectives.

In conclusion, this work contributes to the growing body of evidence demonstrating the potential of reinforcement learning as a powerful approach for autonomous optimization of cloud-native computing environments. The combination of Kubernetes, microservice architectures, and deep reinforcement learning presents compelling opportunities to advance the state of the art in container orchestration. While several technical challenges remain to be addressed, the research presented herein establishes a foundational framework upon which future innovations in intelligent scheduling and adaptive resource management may be developed. The insights and results obtained through this investigation not only validate the feasibility of reinforcement learning in this domain but also provide a roadmap for further exploration at the intersection of cloud infrastructure, machine learning, and autonomic computing.

### Acknowledgements

This study was supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia, Grant No. 451-03-137/2025-03/200132 with University of Kragujevac, Faculty of Technical Sciences Cacak.

### References

[Amaral et al., 2015] Amaral, M., Polo, J., Carrera, D., Mohamed, I., Unuvar, M., Steinder, M.: *Performance Evaluation of Microservices Architectures Using Containers*, 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, pp. 27–34, 2015.

- [Apache, 2025] Apache Software Foundation: Apache JMeter, Open-source load testing tool for analyzing and measuring performance of web applications and services, July 2025. <https://jmeter.apache.org>
- [Burns et al., 2016] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade, *Queue*, Volume 14, Issue 1, 70–93, 2016. <https://doi.org/10.1145/2898442.2898444>
- [Centofanti et al., 2023] Centofanti, C., Tiberti, W., Marotta, A., Cassioli, D.: Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge, 2023 IEEE 9th International Conference on Network Softwarization (NetSoft), June 2023. <https://doi.org/10.1109/NetSoft57336.2023.10175431>
- [Dou et al., 2022] Dou, X., Liu, L., Chen, Y.: Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach, *The 21st USENIX Conference on File and Storage Technologies (FAST)*, December 2022.
- [Google, 2025] Google Cloud: *Managed Service for Prometheus*, Google Cloud Observability Documentation, July 2025. <https://cloud.google.com/stackdriver/docs/managed-prometheus>
- [Gubbi et al., 2013] Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions, *Future Generation Computer Systems*, Vol. 29, No. 7, pp. 1645–1660, 2013. <https://doi.org/10.1016/j.future.2013.01.010>
- [Harris et al., 2018] Harris, D., Naor, J., Raz, D.: Latency Aware Placement in Multi-access Edge Computing, *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, Montreal, QC, Canada, pp. 132–140, 2018.
- [Herrera et al., 2023] Herrera, J. L., Scotece, D., Galán-Jiménez, J., Berrocal, J., Di Modica, G., Foschini, L.: Latency-Optimal Network Microservice Architecture Deployment in SDN, *GLOBECOM 2023 – IEEE Global Communications Conference*, Kuala Lumpur, Malaysia, pp. 5202–5207, 2023.
- [Hightower et al., 2017] Hightower, K., Burns, B., Beda, J.: *Kubernetes: Up and Running*, O'Reilly Media, 2017.
- [Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., Moore, A. W.: Reinforcement Learning: A Survey, *Journal of Artificial Intelligence Research*, Vol. 4, pp. 237–285, 1996.
- [Kallel et al., 2025] Kallel, A., Rekik, M., Khemakhem, M.: A Deep Reinforcement Learning-based Optimization Approach for Containerized Microservice Scheduling in Hybrid Fog/Cloud Environments, *Engineering Applications of Artificial Intelligence*, Vol. 141, Article 109745, 2025.
- [Kuang et al., 2024] Kuang, S., Zhang, J., Mohajer, A.: Reliable Information Delivery and Dynamic Link Utilization in MANET Cloud Using Deep Reinforcement Learning, *Transactions on Emerging Telecommunications Technologies*, vol. 35, no. 9, article e5028, 2024. <https://doi.org/10.1002/ett.5028>
- [Locust, 2025] Locust.io: Open Source Load Testing Tool for Web Apps, Locust Project, July 2025. <https://locust.io>
- [Lu et al., 2024] Lu, J., Li, W., Guo, J., Ding, X., Tang, Z., Wang, T.: Container Scheduling with Dynamic Computing Resource for Microservice Deployment in Edge Computing, 20th International Conference on Mobility, Sensing and Networking (MSN), Harbin, China, pp. 236–243, 2024. <https://doi.org/10.1109/MSN63567.2024.00041>

- [Mao et al., 2010] Mao, M., Li, J., Humphrey, M.: Cloud Auto-Scaling with Deadline and Budget Constraints, *2010 11th IEEE/ACM International Conference on Grid Computing*, Brussels, Belgium, pp. 41–48, 2010.
- [Mao et al., 2016] Mao, H., Alizadeh, M., Menache, I., Kandula, S.: Resource Management with Deep Reinforcement Learning, *HotNets '16: Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 50–56, November 9, 2016. <https://doi.org/10.1145/3005745.3005750>
- [Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: *Human-level control through deep reinforcement learning*, *Nature*, Vol. 518, pp. 529–533, February 2015.
- [Mnih et al., 2016] Mnih, V., Puigdomènech Badia, A., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous Methods for Deep Reinforcement Learning, arXiv preprint arXiv:1602.01783, February 2016. Available at: <https://arxiv.org/abs/1602.01783>
- [Mohajer et al., 2025] Mohajer, A., Hajipour, J., Leung, V. C. M.: Dynamic Offloading in Mobile Edge Computing With Traffic-Aware Network Slicing and Adaptive TD3 Strategy, *IEEE Communications Letters*, vol. 29, no. 1, pp. 95–99, January 2025. <https://doi.org/10.1109/LCOMM.2024.3501956>
- [Morabito et al., 2015] Morabito, R., Kjällman, J., Komu, M.: Hypervisors vs. Lightweight Virtualization: A Performance Comparison, 2015 IEEE International Conference on Cloud Engineering (IC2E), Tempe, AZ, USA, 386–393, 2015. <https://doi.org/10.1109/IC2E.2015.74>
- [Morabito et al., 2016] Morabito, R.: A Performance Evaluation of Container Technologies on Internet of Things Devices, *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, San Francisco, CA, USA, pp. 999–1000, 2016.
- [Pahl et al., 2015] Pahl, C.: Containerization and the PaaS Cloud, *IEEE Cloud Computing*, Vol. 2, No. 3, pp. 24–31, May–June 2015. <https://doi.org/10.1109/MCC.2015.51>
- [Qian et al., 2025] Qian, B., et al.: Edge-Cloud Collaborative Streaming Video Analytics With Multi-Agent Deep Reinforcement Learning, *IEEE Network*, Vol. 39, No. 1, pp. 165–173, January 2025.
- [Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms, arXiv preprint arXiv:1707.06347, July 2017. Available at: <https://arxiv.org/abs/1707.06347>
- [SockShop, 2025] Sock Shop Project: Sock Shop Demo Application, Open-source microservices demo for cloud-native technologies, July 2025. <https://microservices-demo.github.io>
- [Stanišić et al., 2025] Stanišić, S., Đorđević, B., Ristić, O., Tot, I.: *Performance Optimization of File Systems for Docker Containers*, Sinteza 2025 - International Scientific Conference on Information Technology, Computer Science, and Data Science, January 2025.
- [Stanišić et al., 2025] Stanišić, S., Luković, V., Belotić, B.: *Automation of Monitoring and Optimization of Docker Containers Using Artificial Intelligence*, 2025 24th International Symposium INFOTEH-JAHORINA (INFOTEH), March 2025.
- [Stanišić et al., 2025] Stanišić, S., Veskovac, M., Ristic, O., Đorđević, B.: *Security Aspects of Container Orchestration in Kubernetes Environments*, 2025 24th International Symposium INFOTEH-JAHORINA (INFOTEH), March 2025. <https://doi.org/10.1109/INFOTEH64129.2025.10959185>

- [Sutton et al., 2018] Sutton, R. S., Barto, A. G.: Reinforcement Learning: An Introduction, Second Edition, Adaptive Computation and Machine Learning series, The MIT Press, Cambridge, MA, USA, 552 pp., ISBN: 9780262039246, November 13, 2018.
- [Tian et al., 2015] Tian, W., Xu, M., Chen, A., Li, G., Wang, X., Chen, Y.: Open-source simulators for Cloud computing: Comparative study and challenging issues, *Simulation Modelling Practice and Theory*, Volume 58, Part 2, Pages 239–254, 2015.
- [Turnbull, 2018] Turnbull, J.: *Monitoring with Prometheus*, Turnbull Press, June 12, 2018. ISBN: 978-0988820289.
- [Wang and Li, 2021] Wang, K., Li, Y.: ColocationSim: Simulate Colocation Datacenter with Microservices and Performance Interference, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Wuhan, China, pp. 296–297, 2021. <https://doi.org/10.1109/ISSREW53611.2021.00087>
- [Wang et al., 2025] Wang, L., Rodriguez, M. A., Lipovetzky, N.: *Reinforcement learning approach for intelligent job selection and allocation*, *Journal of Supercomputing*, Vol. 81, Article 918, May 2025.
- [Xu et al., 2024] Xu, J., Wan, W., Pan, L., Sun, W., Liu, Y.: The Fusion of Deep Reinforcement Learning and Edge Computing for Real-time Monitoring and Control Optimization in IoT Environments, arXiv preprint arXiv:2403.07923, March 2024. <https://doi.org/10.48550/arXiv.2403.07923>
- [Yang and Mohajer, 2025] Yang, J., Mohajer, A.: Multi-objective Constellation Optimization and Dynamic Link Utilization for Sustainable Information Delivery Using PD-NOMA Deep Reinforcement Learning, *Wireless Networks*, vol. 31, pp. 1839–1859, 2025. <https://doi.org/10.1007/s11276-024-03834-x>
- [Zhang et al., 2020] Zhang, Y., Liu, T., Zhu, Y., Yang, Y.: A Deep Reinforcement Learning Approach for Online Computation Offloading in Mobile Edge Computing, 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS), Hang Zhou, China, pp. 1–10, 2020. <https://doi.org/10.1109/IWQoS49365.2020.9212868>
- [Zheng et al., 2022] Zheng, X., Liang, C., Wang, Y., Lim, G.: Multi-AGV Dynamic Scheduling in an Automated Container Terminal: A Deep Reinforcement Learning Approach, *Mathematics*, Vol. 10, No. 23, Article 4575, December 2022.