

Development of a Load Balancing Framework for Microservices Architecture Using Reinforcement Learning

*¹Olawuwo Abdulmuiz Adesina, ²Muhammad Aliyu Suleiman, ¹Prema Kirubakaran and ²Salisu Ibrahim Yusuf

¹Information Technology Department, Nile University of Nigeria, Abuja, Nigeria.

²Computer Science Department, Nile University of Nigeria, Abuja, Nigeria.

*Corresponding author: abdulmuiz101@gmail.com

ABSTRACT

The increasing adoption of microservices architecture has improved the scalability, modularity, and maintainability of cloud-native applications. However, the dynamic and heterogeneous nature of microservices workloads creates significant challenges for conventional load-balancing mechanisms, which typically rely on predefined routing rules and have limited ability to respond to changing runtime conditions. This study develops and evaluates RL-MicroLB, an adaptive load-balancing framework that uses reinforcement learning to dynamically distribute requests among microservice instances. The framework employs Proximal Policy Optimization (PPO) and integrates Kubernetes, Istio Service Mesh, Prometheus, Locust, and StableBaselines3 within a closed loop traffic management architecture. Runtime performance measurements are collected through Prometheus and supplied as observations to the PPO agent. The agent determines traffic-routing weights, which are dynamically enforced through Istio VirtualService resources, while subsequent system measurements provide feedback for continued decision-making. The framework was implemented using the Sock Shop microservices application and evaluated under varying workload conditions. Experimental results showed that RL-MicroLB reduced P95 latency from 420 ms under Round Robin to 285 ms, representing a 32.1% reduction. Average response time decreased from 240 ms to 185 ms, CPU utilization variance decreased from 0.18 to 0.07, CPU balance score increased from 0.72 to 0.91, and throughput increased from 820 to 845 requests per second. Comparative evaluation against Least Connections, Weighted Round Robin, and Deep Q-Network further showed that PPO achieved the best overall performance among the evaluated approaches under the experimental conditions. The results demonstrate the feasibility of integrating reinforcement learning with Kubernetes monitoring and service-mesh traffic management for adaptive request-level load balancing in microservices environments.

Received: 03 July 2026

Accepted: 20 July 2026

Published: 08 Oct. 2026

Keywords: Microservices, Load Balancing, Reinforcement Learning, Proximal Policy Optimization, Kubernetes, Istio, Service Mesh, Cloud Computing

INTRODUCTION

Microservices architecture has become an important architectural approach for developing modern cloud-native applications. Unlike monolithic systems, microservices decompose applications into smaller, independently deployable and loosely coupled services that communicate through lightweight interfaces. This architecture supports independent scaling, modular development, fault isolation, continuous deployment, and more flexible resource allocation. However, the distributed nature of microservices also introduces challenges involving service coordination, traffic management, resource utilization, and maintenance of application performance under rapidly changing workloads. Load balancing is consequently essential for maintaining the availability and performance of microservices applications. Conventional approaches such as Round Robin (RR), Weighted Round Robin (WRR), and Least Connections (LC) distribute requests according to predefined rules. Although these approaches are computationally inexpensive and relatively straightforward to implement, their decisions do not adequately account for continuously changing runtime conditions. Microservice instances may differ in CPU utilization, memory usage, response latency, processing capacity, and current request load, while incoming traffic may

fluctuate substantially because of workload spikes, service failures, autoscaling events, and changes in user demand.

A static load-balancing policy can therefore continue directing requests according to its predetermined rule even when individual instances experience deteriorating performance. This can result in uneven resource utilization, overloaded instances, increased response times, higher tail latency, and reduced service quality.

Researchers have increasingly investigated machine learning and reinforcement learning as alternatives to static resource-management strategies. Reinforcement Learning (RL) is particularly relevant because an agent learns a decision policy through repeated interaction with an environment. Rather than permanently applying the same routing rule, an RL agent observes system conditions, performs an action, evaluates the resulting performance through a reward mechanism, and progressively improves its decision policy.

Previous studies have applied reinforcement learning to cloud resource scheduling, microservice placement, instance selection, and load balancing. Gasmi et al. investigated reinforcement-learning-based placement of microservices applications in edge environments. Boudieb et al. applied deep reinforcement learning to microservice instance selection and load balancing in dynamic fog environments. Other studies have investigated DQN, Double DQN, multi-

agent reinforcement learning, and multi-objective RL for adaptive resource-management problems.

These studies demonstrate the potential of learning-based resource management, but several limitations remain. A significant proportion of existing work concentrates on resource scheduling, placement, fog/edge resource allocation, or simulated environments rather than closed-loop request-level traffic management integrated with a Kubernetes service mesh. Value-based methods such as Q-learning and DQN can also become difficult to manage as state and action spaces increase in complexity. Multi-agent and multi-objective methods introduce additional challenges associated with coordination, convergence, and reward formulation.

Furthermore, practical integration between an RL decision-making agent, real-time telemetry, Kubernetes orchestration, and a service mesh capable of immediately enforcing learned traffic-routing decisions remains comparatively underexplored.

This study addresses this gap through RL-MicroLB, a closed-loop reinforcement-learning based framework for adaptive microservices load balancing. RL-MicroLB uses Proximal Policy Optimization (PPO) to determine traffic routing weights based on runtime observations collected through Prometheus. The selected routing weights are dynamically enforced through Istio Service Mesh, after which subsequent system measurements provide feedback for continued decision-making.

The major contributions of this study are:

Development of a closed loop adaptive load-balancing framework integrating PPO, Kubernetes, Prometheus, and Istio Service Mesh.

Formulation of microservices request distribution as an RL problem involving runtime state observations, traffic-routing actions, and performance-based reward feedback.

Implementation of a PPO-based controller capable of dynamically modifying Istio routing weights according to observed system conditions.

Practical implementation and evaluation using the Sock Shop microservices application.

Comparative evaluation against RR, LC, WRR, and DQN using P95 latency, average response time, throughput, CPU utilization balance, traffic stability, and workload-spike recovery.

The experimental results indicate that RL-Micro LB can improve application-level performance and resource utilization under changing workload conditions.

Related Work

Reinforcement Learning for Load Balancing

Reinforcement learning models' decision-making as repeated interaction between an agent and an environment. At time t , the agent observes state S_t , selects action A_t , receives reward R_t , and observes the subsequent state S_{t+1} . The objective is to learn a policy that maximizes cumulative reward.

Q learning represents one of the fundamental model free RL algorithms. However, maintaining a Q-table becomes impractical when the number of states and actions becomes large. Deep Q-Networks address this limitation by approximating action values using neural networks. DQN therefore provides greater scalability than conventional Q-learning but remains a value-based approach and may

experience challenges when applied to more complex traffic allocation decisions.

PPO is a policy gradient method that constrains policy updates to improve training stability. Its policy based formulation makes it suitable for multidimensional observations and adaptive traffic weight decisions.

Existing RL-Based Microservices Approaches

Existing studies demonstrate increasing interest in adaptive microservices resource management. Gasmi et al. applied reinforcement learning to microservices placement in edge environments. Their work demonstrated that learning mechanisms can support dynamic placement decisions based on application and infrastructure conditions.

Boudieb et al. developed a deep reinforcement learning approach for microservice-instance selection and load balancing in fog computing. Their approach considered dynamic requests and service dependencies and demonstrated improvements in delay and load distribution.

Multi-agent reinforcement learning has also been investigated for distributed load-balancing problems. Such methods allow several agents to coordinate decisions but introduce additional complexity because each agent operates within an environment influenced by the actions of other agents. Multi-objective RL extends adaptive resource management by optimizing multiple performance objectives simultaneously. Although this provides greater flexibility, reward design and preference alignment become more complex.

Research Gap

Existing research establishes that RL can improve dynamic resource management, but three gaps remain relevant to this study. First, many approaches focus on placement, scheduling, or instance selection rather than direct request level traffic allocation. Secondly, many evaluations rely primarily on simulations rather than integrating the RL controller with actual cloud-native traffic management infrastructure. Lastly, relatively limited work integrates real-time telemetry, RL decision-making, service-mesh enforcement, and performance feedback within one closed loop framework.

RL-MicroLB addresses this gap by integrating Prometheus, PPO, Kubernetes, and Istio to implement adaptive request-level load balancing.

MATERIALS AND METHODS

Methodology

Research Design

The study adopts a quantitative experimental research design involving the development, implementation, training, and comparative evaluation of an adaptive RL-based load-balancing framework. The experimental design compares RL-MicroLB with Round Robin, Least Connections, Weighted Round Robin, and Deep Q-Network under varying traffic conditions.

RL-MicroLB Frameworks

The proposed framework consists of four interconnected functional layers: Traffic Generation Layer; Observation and Monitoring Layer; RL Decision Layer; and Action Execution and Feedback Layer.

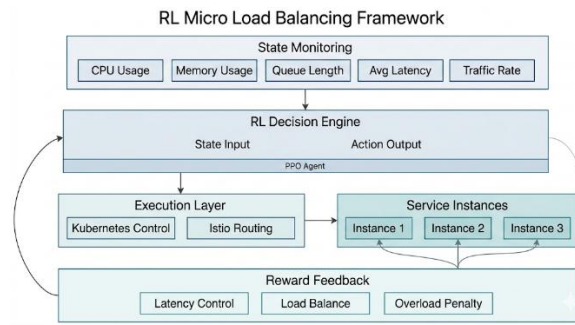


Figure 1: Micro Load Balancing Framework

The operational information flow is: Workload -> Update. This establishes a continuous feedback loop. We can further understand this via the conceptual framework in Figure below.

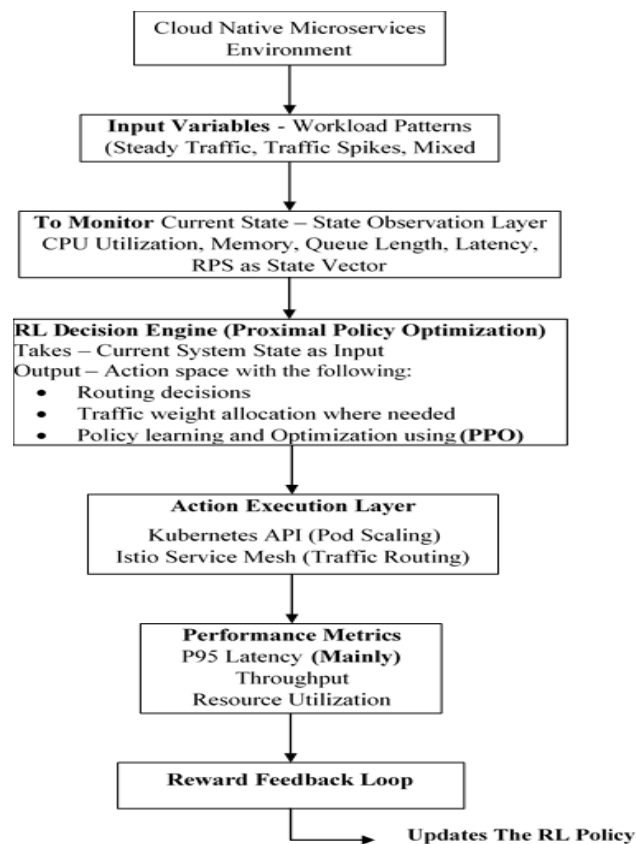


Figure 2: Conceptual Framework Diagram

Traffic Generation Layer

Locust was used to generate HTTP traffic against the deployed Sock Shop application. The generated workloads represented stable request conditions, changing traffic intensity, and sudden workload spikes. Requests entered the Kubernetes environment through the Istio Ingress Gateway and were subsequently distributed according to the currently configured routing weights.

Observation and Monitoring Layer

Prometheus was used to collect runtime measurements from the microservices environment. The framework monitored system and application indicators including CPU utilization, request latency, and request rate. These observations were retrieved using PromQL queries. The state at decision interval t can be represented conceptually as $S_t = [CPU_t, Memory_t, Queue_t, Latency_t, ArrivalRate_t, \dots]$.

PPO Decision Engine

The RL decision engine was implemented using Stable-Baselines3, with Proximal Policy Optimization selected as the learning algorithm. At time t , the agent observes state S_t and selects traffic-routing action A_t according to the policy $\pi_\theta(A_t|S_t)$. The selected action determines the traffic distribution between available service versions. PPO was selected because of its relatively stable policy-update mechanism and suitability for multidimensional state representations.

Reward Function

The reward function encourages the agent to reduce latency and improve resource balance while penalizing overloaded instances:

$$r_t = \alpha(1 - L_t/L_{max}) + \beta(1 - |U_i - U_{avg}|) - \gamma P_{overload}$$

where L_t is observed latency, L_{max} is the maximum latency reference, U_i is utilization of service instance i , U_{avg} is average utilization, $P_{overload}$ is the overload penalty, and α , β , and γ are weighting coefficients.

Action Execution through Istio

The experimental deployment contained two frontend service versions, frontend-v1 and frontend-v2, configured as Istio

subsets. An Istio DestinationRule defined the subsets, while an Istio VirtualService determined the proportion of traffic routed to each version. The PPO agent dynamically adjusted these routing weights through the Kubernetes Python client. A 5-second control interval was used to allow configuration changes to propagate before subsequent measurements were evaluated.

Experimental Environment

Table 1: Experimental Environment

Component	Technology	Function
Cloud environment	Amazon EKS	Kubernetes hosting
Container orchestration	Kubernetes	Microservices deployment
Test application	Sock Shop	Experimental workload
Service mesh	Istio	Traffic routing
Monitoring	Prometheus	Runtime telemetry
Traffic generator	Locust	Workload generation
RL algorithm	PPO	Adaptive decision-making
RL library	Stable-Baselines3	PPO implementation
Controller	Kubernetes Python client	Istio configuration
Training monitoring	TensorBoard/logging	Training observation

RL-MicroLB Control Algorithm

Step 1: Generate traffic using Locust.
 Step 2: Route incoming traffic through Istio.
 Step 3: Collect runtime measurements through Prometheus.
 Step 4: Construct state S_t .
 Step 5: Provide S_t to the PPO policy.
 Step 6: Generate action A_t .
 Step 7: Convert A_t into Istio traffic weights.
 Step 8: Patch the VirtualService.
 Step 9: Allow the environment to operate under the new routing policy.
 Step 10: Collect the next performance observations.

Step 11: Calculate reward R_t .
 Step 12: Return R_t and $S_{(t+1)}$ to the PPO environment.
 Step 13: Continue the control loop.

PPO Training Procedure

The PPO model was trained using Stable-Baselines3 in a Gym-compatible environment. The console output confirmed CPU-based execution, wrapping of the environment using DummyVecEnv, logging to logs/PPO_v1_0, progressive environment interaction, checkpoint storage, and completion of the configured training process. Checkpoints were configured at nominal intervals of 1,000 training steps.

Table 2: Checkpoints Configured at Nominal Intervals of 1,000 Training Steps

Saved Checkpoint	Reported PPO Total Timesteps
1,000	2,048
2,000	4,096
3,000	6,144
4,000	8,192
5,000	10,240
6,000	12,288
7,000	14,336
8,000	16,384
9,000	18,432
10,000	20,480

The difference between checkpoint labels and Stable-Baselines3 total_timesteps results from PPO processing complete rollout batches. Consequently, the number of environments timesteps processed can exceed the nominal number supplied for individual training segments. The visible execution speed ranged approximately from 658 to 1,076 frames per second.

The logs confirm successful PPO training execution and checkpoint creation. They should not be interpreted as evidence of reward convergence because episodic reward values are not displayed in the available console output.

Evaluation Metrics

P95 Latency: response time within which 95% of requests are completed.

Average Response Time: average time required to complete application requests.

Throughput: number of successfully processed requests per second.

CPU Utilization Variance: variance in CPU utilization across service instances; lower values indicate more balanced utilization.

CPU Balance Score: measure of how evenly computational demand is distributed across instances.

Traffic Stability Index: measure of stability in traffic distribution under changing workloads.

Spike Recovery Time: time required for the system to recover following a sudden workload increase.

RESULTS AND DISCUSSION

Results

Framework Implementation

The RL-MicroLB framework successfully connected workload generation, monitoring, reinforcement-learning decision-making, and service-mesh traffic control. Locust generated requests against the Sock Shop application while Prometheus collected runtime performance information. The PPO controller processed observations and modified Istio routing weights through the Kubernetes API. This demonstrated that RL decisions could be translated into live traffic-management actions without restarting the microservices.

Istio Ingress Gateway, DestinationRule, Initial VirtualService routing weights, Locust workload, Prometheus monitoring, and RL-MicroLB routing weights after PPO action.

Training Execution Performance

The PPO training process completed all ten configured checkpoint stages. Stable-Baselines3 reported increasing total environment timesteps from 2,048 during the first displayed rollout to 20,480 timesteps at the completion of the tenth checkpoint stage. The available console output reports execution measurements such as FPS, elapsed time, iterations, and total timesteps. It does not report episodic reward, policy loss, value loss, latency, or CPU balance at individual training checkpoints. Therefore, no unsupported claim of reward convergence is made from these screenshots.

Round Robin versus RL-MicroLB

Table 3: Round Robin versus RL-MicroLB

Metric	Round Robin	RL-MicroLB	Improvement
P95 Latency	420 ms	285 ms	32.1% reduction
Average Response Time	240 ms	185 ms	22.9% reduction
CPU Utilization Variance	0.18	0.07	61.1% reduction
CPU Balance Score	0.72	0.91	26.4% increase
Throughput	820 req/s	845 req/s	3.0% increase

RL-MicroLB reduced P95 latency from 420 ms to 285 ms, average response time from 240 ms to 185 ms, and CPU utilization variance from 0.18 to 0.07. The CPU balance score

increased from 0.72 to 0.91, while throughput increased from 820 to 845 requests per second.

Comparative Evaluation

Table 4: Comparative Evaluation

Metric	LC	WRR	DQN	PPO
P95 Latency (ms)	390	365	310	285
Average Response Time (ms)	225	210	195	185
CPU Variance	0.14	0.11	0.09	0.07
CPU Balance Score	0.78	0.83	0.87	0.91
Traffic Stability Index	0.74	0.79	0.82	0.87
Spike Recovery Time (s)	30	28	22	18

PPO achieved the lowest P95 latency and average response time, the lowest CPU variance, the highest CPU balance score and traffic stability index, and the shortest spike recovery time among the evaluated approaches.

increased workload conditions, however, RL-MicroLB maintained lower P95 latency. This behaviour is consistent with the adaptive structure of RL-MicroLB, in which changes in runtime performance influence subsequent traffic-routing decisions.

P95 Latency under Variable Traffic

Under relatively stable workloads, the difference between Round Robin and RL-MicroLB was less pronounced. During

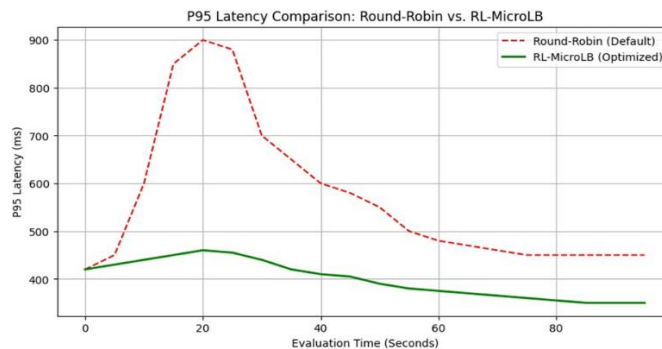


Figure 3: Latency Comparison of Round Robin Vs. RL-MicroLB

Summary of Results

P95 latency decreased by 32.1% compared with Round Robin.

Average response time decreased by 22.9%.
CPU utilization variance decreased by 61.1%.
CPU balance score increased from 0.72 to 0.91.

Throughput increased from 820 to 845 requests per second. Traffic stability reached 0.87 under PPO. Spike recovery time decreased to 18 seconds. PPO produced better results than the evaluated LC, WRR, and DQN approaches across all reported comparative metrics. PPO training executed through all ten configured checkpoint stages, with Stable-Baselines3 reporting up to 20,480 environment timesteps.

Discussion

The experimental results indicate that PPO based RL-MicroLB provided improved load-balancing performance compared with the evaluated conventional and learning-based approaches under the experimental conditions.

The reduction in P95 latency is particularly significant. Round Robin produced a P95 latency of 420 ms, whereas RL-MicroLB achieved 285 ms, representing a 32.1% reduction. This indicates that dynamically modifying routing weights according to runtime observations was more effective during the experiment than distributing requests using a fixed sequence.

The CPU-utilization results reinforce this interpretation. CPU utilization variance decreased from 0.18 to 0.07, corresponding to a 61.1% reduction. This corrects the earlier discussion that incorrectly described the result as an increase. A reduction in CPU variance indicates that computational demand was distributed more evenly across the evaluated service instances. This interpretation is supported by the increase in CPU balance score from 0.72 to 0.91.

The combination of reduced P95 latency and improved CPU balance is important. It indicates that the observed latency improvement was accompanied by better distribution of resource demand rather than simply transferring overload from one service instance to another.

Throughput increased from 820 to 845 requests per second. Although this 3.0% increase is smaller than the latency and resource-balance improvements, it indicates that the adaptive routing process did not achieve lower latency at the expense of request processing capacity.

The traffic-spike results further demonstrate the adaptive behaviour of the framework. PPO achieved an 18-second spike recovery time compared with 22 seconds for DQN, 28 seconds for WRR, and 30 seconds for LC.

The result can be explained by the closed-loop architecture: Observe -> Decide -> Act -> Measure -> Learn. Prometheus supplies observations, PPO determines the routing action, Istio enforces the decision, and subsequent measurements provide feedback.

The comparison with DQN is also noteworthy. DQN achieved better performance than the traditional algorithms across several indicators, supporting the broader argument that adaptive learning can improve dynamic load balancing. PPO nevertheless achieved lower latency, lower CPU variance, better CPU balance, greater traffic stability, and faster spike recovery under the experimental configuration.

The findings are consistent with earlier studies demonstrating the potential of reinforcement learning for dynamic microservice management. However, RL-MicroLB differs from approaches primarily concerned with service placement or scheduling because it focuses directly on request-level traffic weights and connects the RL controller to Kubernetes and Istio.

The results must nevertheless be interpreted within the boundaries of the experiment. They demonstrate that PPO-based RL-MicroLB outperformed the selected baselines under the evaluated application topology, workload configuration, and deployment environment. They do not

establish that PPO is universally superior to every load-balancing algorithm under all possible microservices workloads.

A further limitation of the present evidence is that the available training-console logs establish successful training execution but do not contain reward values. Consequently, claims concerning reward convergence should be supported by TensorBoard scalar data if those logs remain available.

CONCLUSION

This study developed and evaluated RL-MicroLB, an adaptive reinforcement-learning-based framework for load balancing in cloud-native microservices.

The framework integrates Prometheus-based runtime monitoring, a PPO decision engine, Kubernetes orchestration, and Istio Service Mesh within a closed-loop control architecture. Unlike conventional load-balancing algorithms that rely primarily on predetermined rules, RL-MicroLB observes changing runtime conditions and dynamically modifies traffic-routing weights.

Experimental evaluation demonstrated that RL-MicroLB reduced P95 latency by 32.1% compared with Round Robin. Average response time decreased by 22.9%, CPU utilization variance decreased by 61.1%, CPU balance score increased from 0.72 to 0.91, and throughput increased by 3.0%. PPO also achieved the best reported results among the evaluated LC, WRR, DQN, and PPO approaches for P95 latency, average response time, CPU variance, CPU balance, traffic stability, and spike recovery.

The study therefore demonstrates the practical feasibility of integrating reinforcement learning with Kubernetes monitoring and Istio traffic management for adaptive request-level load balancing. The contribution of the work extends beyond the selection of PPO as an algorithm: the proposed framework demonstrates how monitoring, state representation, RL decision-making, traffic enforcement, and feedback can be integrated into a practical adaptive load-balancing loop.

Limitations and Future Work

RL training introduces computational overhead and requires interaction with the environment before a useful policy can be obtained. The quality of routing decisions also depends on the accuracy and timeliness of monitoring information. Furthermore, the experimental evaluation used a specific microservices application and topology, and the reward formulation primarily concentrates on latency and resource balance.

Future work should investigate larger and more heterogeneous microservices environments, multi-agent reinforcement learning, combined autoscaling and load-balancing optimization, multi-objective reward functions incorporating cost and energy consumption, resilience to service and network failures, longer-duration and hyperscale workload evaluation, and quantitative analysis of reward and performance metrics throughout PPO training.

REFERENCES

Boudieb, W., Malki, A., Malki, M., Badawy, A., & Barhamgi, M. (2024). Microservice instances selection and load balancing in fog computing using deep reinforcement learning approach. *Future Generation Computer Systems*, 156, 77–94. DOI: <https://doi.org/10.1016/j.future.2024.03.010>.

Daruvuri, R. (2023). Dynamic load balancing in AI-enabled cloud infrastructures using reinforcement learning and

- algorithmic optimization. *World Journal of Advanced Research and Reviews*, 20(1), 1327–1335. DOI: <https://doi.org/10.30574/wjarr.2023.20.1.2045>.
- Gasmi, K., Abassi, K., Romdhani, L., & Debauche, O. (2023). Reinforcement learning-based approach for microservices-based application placement in edge environment. In *2023 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. DOI: <https://doi.org/10.1109/ISCC58397.2023.10218098>.
- Kamila, N. K., Frnda, J., Pani, S. K., Das, R., Islam, S. M. N., Bharti, P. K., & Muduli, K. (2022). Machine learning model design for high performance cloud computing and load balancing resiliency: An innovative approach. *Journal of King Saud University – Computer and Information Sciences*, 34(10), 9991–10009. DOI: <https://doi.org/10.1016/j.jksuci.2022.10.001>.
- Kaveri, P. R., & Lahande, P. V. (2023). Reinforcement learning to improve resource scheduling and load balancing in cloud computing. *SN Computer Science*, 4, 188. DOI: <https://doi.org/10.1007/s42979-022-01609-9>.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.
- Yao, Z., Ding, Z., & Clausen, T. H. (2022). Multi-agent reinforcement learning for network load balancing in data center. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. DOI: <https://doi.org/10.1145/3511808.3557133>.



©2026 This is an Open Access article distributed under the terms of the Creative Commons Attribution 4.0 International license viewed via <https://creativecommons.org/licenses/by/4.0/> which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is cited appropriately.