

Adaptive CPU Scheduling using a Q-Learning Meta-Scheduler

Abhaya Bhatta

*Department of Software Engineering
Nepal College of Information Technology
Pokhara University
Balkumari, Lalitpur, Nepal
abhaya.221603@ncit.edu.np*

Aryam Ghimire

*Department of Software Engineering
Nepal College of Information Technology
Pokhara University
Balkumari, Lalitpur, Nepal
aryan.221612@ncit.edu.np*

Bhusan Thapa

*Department of Software Engineering
Nepal College of Information Technology
Pokhara University
Balkumari, Lalitpur, Nepal
bhusan@ncit.edu.np*

Tathastu Subedi

*Department of Software Engineering
Nepal College of Information Technology
Pokhara University
Balkumari, Lalitpur, Nepal
tathastu.221647@ncit.edu.np*

Zoyeb Shrestha

*Department of Software Engineering
Nepal College of Information Technology
Pokhara University
Balkumari, Lalitpur, Nepal
zoyeb.221648@ncit.edu.np*

Abstract—Conventional CPU scheduling policies such as First-Come First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority scheduling apply fixed, pre-defined rules that are optimal only under specific workload conditions and cannot adapt to dynamic and heterogeneous execution environments. This paper presents an adaptive meta-scheduler that employs tabular Q-learning to dynamically select among classical scheduling policies based on the runtime state of the system. A discrete-event CPU simulation environment models process lifecycle behaviour, and a Gymnasium-compatible reinforcement learning (RL) environment exposes a discretised five-dimensional state representation, a four-action policy-selection space, and a reward function that penalises accumulated waiting and turnaround time while rewarding process completions and CPU utilisation. The agent is evaluated on four workload profiles—CPU-bound, I/O-bound, bursty, and mixed—generated from Poisson arrivals and exponential or uniform burst distributions. Experimental results show that the agent learns a meaningful state-dependent policy, exhibiting a clear preference for Round Robin under high-variance workloads. Coarsening the state representation from 3,125 to 243 discrete states raises state-space coverage from 2.34% to 12.35% for a fixed training budget, and the resulting policy remains competitive with the strongest static baseline on bursty and CPU-bound workloads, with a gap of 5.6% and 13.8% in average waiting time respectively. The framework is modular and reproducible, with a full-stack implementation comprising a FastAPI backend, PostgreSQL persistence, and a React dashboard.

Keywords —*Reinforcement Learning, Q-Learning, CPU Scheduling, Operating Systems, Meta-Scheduler, Process Scheduling, Workload Characterization*

I. INTRODUCTION

CPU scheduling is a core function of operating systems (OS), responsible for deciding which process in the ready queue should be assigned the CPU at any given instant.

Modern systems contend with highly dynamic and unpredictable load patterns: processes may be CPU-bound, I/O-bound, or bursty, and their arrival and execution characteristics vary over time. Classical OS texts and queueing-theory treatments establish the workload dependence of scheduling policies and the balance between fairness, responsiveness, and throughput in multiprogrammed systems [10]–[12]. Traditional scheduling algorithms—FCFS, SJF, Round Robin, and Priority scheduling—are favoured for their efficiency and simplicity, but they apply static, pre-set rules that remain unchanged throughout execution regardless of the workload in progress. For example, SJF minimises average waiting time for short tasks but is prone to starvation, while Round Robin improves responsiveness at the cost of increased context switching. A strategy that is effective under one workload is therefore unlikely to remain effective under another, leading to suboptimal CPU utilisation, increased waiting time, and reduced system responsiveness [1].

Reinforcement learning (RL) offers a principled mechanism for adaptive decision making in such settings. An RL agent learns a policy through repeated interaction with an environment by maximising a scalar reward signal, without requiring an explicit model of the environment. The RL framework and Q-learning update rule are formally established in the literature [8], [9], [15], and prior work in resource management and cluster scheduling has shown that learned policies can adapt to changing workload conditions better than fixed heuristics [3]–[5], [16], [17]. However, these systems typically rely on deep RL and large state spaces, incurring substantial training cost, slow convergence, and heavy computational overhead that limit their use in lightweight or embedded environments [7].

This paper proposes a lightweight tabular Q-learning meta-

scheduler that selects among existing classical scheduling policies rather than replacing them. The system is designed as a meta-scheduler layered on top of a discrete-event CPU simulator: at each decision interval it observes a coarsened, discretised representation of the system state—ready-queue length, mean remaining burst, CPU utilisation, completion ratio, and dominant process class—and selects the scheduling algorithm to apply for the next interval. The main contributions of this work are summarised as follows:

- A discrete-event CPU scheduling simulator together with four classical schedulers (FCFS, SJF/SRTF, Round Robin, and Priority), validated against hand-calculated textbook workloads.
- A Gymnasium-compatible RL environment that formalises scheduling policy selection as a Markov decision process, including a principled reward function and a configurable workload generator producing four distinct workload profiles.
- An empirical analysis of state-space coarsening and reward-coefficient calibration, showing that coarsening the state space from 3,125 to 243 states raises visited-state coverage from 2.34% to 12.35% under a fixed training budget.
- A quantitative evaluation of the learned policy against all four baselines across all workload profiles using waiting time, turnaround time, response time, CPU utilisation, and throughput, together with policy-decision and convergence validation.

The remainder of this paper is organised as follows. Section II reviews related work. Section III describes the proposed methodology. Section IV presents the experimental setup, and Section V discusses the results. Finally, Section VI concludes the paper and outlines future research directions.

II. RELATED WORK

Classical scheduling algorithms have been studied extensively, and their performance characteristics under various workloads are well understood [1], [2], [10], [11]. In real operating systems, fairness and latency trade-offs are typically managed through queue-based policies such as Round Robin and priority variants [13], [14]. Nevertheless, because these algorithms are static, they cannot respond to changes in workload or system state, motivating research into learning-based schedulers.

Fan and Lan [3] proposed DRAS-CQSim, an open-source framework that couples deep RL agents with CQSim, an event-based HPC scheduling simulator, supporting both deep Q-learning and policy-gradient methods alongside conventional policies such as FCFS and SJF. While the framework improves scheduling efficiency and enables comparison between traditional and RL-based approaches, it supports only two RL algorithms and leaves open the question of inference latency and computational overhead before large-scale deployment. In a similar direction, Zhang et al. [4] presented RLScheduler, an automated HPC batch job scheduler built on RL with a kernel-based network structure and trajectory filtering to

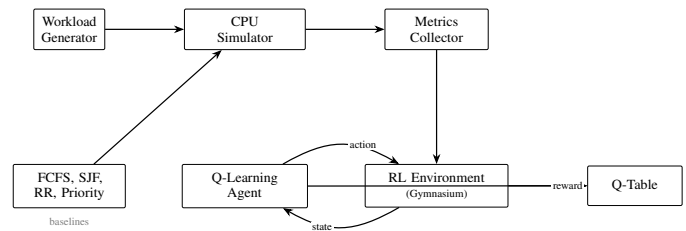


Fig. 1. Architecture of the proposed RL-based scheduling framework. The agent observes a discretised system state and selects a classical policy for the next decision interval.

stabilise learning. The authors demonstrate stable performance on unseen workloads, but the approach still depends on function approximation and a comparatively complex training pipeline.

Beyond HPC, RL has been applied to adaptive resource scheduling in complex system environments. The broader literature on deep RL for resource management shows that policy learning can improve decision quality under variable demand [15]–[17]. Li et al. [5] report that Q-learning-based schedulers can outperform conventional methods in task completion time, throughput, waiting time, and resource utilisation on traces such as Google Cluster Data. Nguyen et al. [6] combine fuzzy Q-learning with the MPQUIC transport scheduler, illustrating the broader applicability of tabular Q-learning to online scheduling decisions. However, these works remain heavily dependent on simulation environments, require considerable training time before convergence, and face scalability problems when state spaces grow large. Xing et al. [7] address mixed database workloads using hierarchical deep RL with graph neural networks and meta-learning; while powerful, the framework combines numerous advanced techniques, is tailored to database environments rather than general OS scheduling, and requires significant computational resources.

Taken together, the literature indicates that RL-based scheduling can improve adaptability and resource utilisation, but at the cost of high training complexity, long convergence times, and limited scalability of the learned policy. This motivates a lightweight, modular alternative: a tabular Q-learning meta-scheduler with a deliberately coarsened state representation, fast training, and transparent evaluation against classical baselines under controlled workload conditions.

III. PROPOSED METHODOLOGY

The proposed framework aims to improve scheduling efficiency by letting a Q-learning agent decide, at runtime, which classical scheduling policy to apply to the current set of queued processes. The overall pipeline is illustrated in Fig. 1 and consists of four stages: workload generation, discrete-event CPU simulation, state observation and reward computation, and Q-learning based policy selection.

A. Discrete-Event CPU Simulation

Each process is represented by a Process Control Block (PCB) capturing the process identifier, arrival time, total

TABLE I
WORKLOAD GENERATION PARAMETERS

Workload	Arrival rate λ	Burst distribution	Mean burst μ
CPU-bound	2	Exponential	15
I/O-bound	5	Exponential	4
Bursty	1	Uniform	—
Mixed	3	Exponential	10

burst time, remaining burst, priority, lifecycle state (NEW, READY, RUNNING, TERMINATED), and workload class. The simulation engine advances a global clock in fixed tick increments while managing process admission, context switching, and CPU idle/busy accounting; arrival admission is performed in amortised $O(1)$ time per tick using an admission pointer. Turnaround and waiting time are derived post-execution as

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}, \quad (1)$$

$$T_{\text{waiting}} = T_{\text{turnaround}} - B_{\text{total}}, \quad (2)$$

where $T_{\text{completion}}$ is the simulation clock time at termination and B_{total} is the total burst time of the process.

Four classical schedulers are implemented under a common scheduler interface with per-tick and completion hooks: FCFS (earliest arrival, non-preemptive), SJF implemented as Shortest-Remaining-Time-First (SRTF), Round Robin with a default quantum of four time units, and Priority scheduling with ties broken by arrival time. Each scheduler was validated independently against hand-calculated textbook workloads through unit tests.

B. Workload Generation

The workload generator produces reproducible process batches by sampling process attributes from statistical distributions that approximate realistic OS behaviour. Process arrivals are modelled as a Poisson process: the inter-arrival time Δt follows an exponential distribution

$$\Delta t \sim \text{Exponential}\left(\frac{1}{\lambda}\right), \quad \Delta t > 0, \quad (3)$$

where λ is the arrival rate in processes per time unit. CPU burst times follow an exponential distribution with mean μ for the CPU-bound and I/O-bound profiles, and a uniform distribution $B \sim \text{Uniform}(1, 50)$ for the bursty profile. Table I lists the parameterisation of the four supported workload profiles; priority values are sampled uniformly from the given range at process creation. All sampling uses a seeded random generator for full reproducibility.

C. Reinforcement Learning Formulation

The scheduling problem is modelled as a Markov decision process whose policy is a mapping from a discretised description of the system to one of four scheduling policies. At each decision point, the observed system state is a five-dimensional vector

$$s = \langle \hat{Q}, \hat{B}, \hat{U}, \hat{C}, \hat{P} \rangle, \quad (4)$$

TABLE II
STATE SPACE COMPONENTS AND DISCRETISATION

Component	Raw metric	Bins (thresholds)
$\hat{Q}$	Ready-queue length	low (≤ 1), med (≤ 5), high
$\hat{B}$	Mean remaining burst	short (≤ 3), med (≤ 8), long
$\hat{U}$	CPU utilisation	low ($\leq 33\%$), mod ($\leq 67\%$), high
$\hat{C}$	Completion ratio	low ($\leq 33\%$), mod ($\leq 67\%$), high
$\hat{P}$	Dominant process class	empty, predictable, variable

where $\hat{Q}$ is the ready-queue length, $\hat{B}$ the mean remaining burst of queued processes, $\hat{U}$ the CPU utilisation over the preceding interval, $\hat{C}$ the fraction of the workload completed, and $\hat{P}$ the dominant process class. Each dimension is discretised into three bins; the discretisation thresholds are summarised in Table II. This yields $3^5 = 243$ discrete states, implemented as a `MultiDiscrete` observation space.

The action space is $\mathcal{A} = \{\text{FCFS, SJF, RR, Priority}\}$, and each selected action is applied for a decision interval of ten simulation ticks before the agent decides again. The reward signal is computed at the epoch level from the change in system-level metrics between the start and end of a decision interval, so that the agent optimises global scheduling behaviour rather than greedy per-process outcomes. Let $W(t)$ be the waiting time accumulated across all processes, $T(t)$ the accumulated turnaround time, $C(t)$ the number of completed processes, and $B(t)$ the cumulative CPU-busy time at simulation time t . Given snapshots at t and $t + \Delta$, with deltas ΔW , ΔT , ΔC , ΔB and CPU utilisation $\rho = \Delta B / \Delta$, the per-epoch reward is

$$R = -\lambda_W \Delta W - \lambda_T \Delta T + \lambda_C \Delta C + \lambda_U \rho, \quad (5)$$

with weights $\lambda_W = 0.1$ (waiting-time penalty), $\lambda_T = 0.1$ (turnaround-time penalty), $\lambda_C = 1.0$ (reward per completed process), and $\lambda_U = 0.5$ (CPU-utilisation reward). The waiting- and turnaround-time terms are penalties because these metrics grow monotonically; the completion term rewards throughput directly; and the utilisation term discourages the agent from idling the CPU purely to avoid wait-time penalties.

D. Q-Learning Agent

The agent maintains a tabular estimate $Q(s, a)$ of the expected cumulative reward for taking action a in state s , updated according to the Bellman equation [8]

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \quad (6)$$

where α is the learning rate, γ the discount factor, r the immediate reward, and s' the observed next state. Action selection follows an ϵ -greedy policy [9]: with probability ϵ a random action is chosen, and otherwise the action maximising $Q(s, a)$ is selected. To shift from exploration to exploitation progressively, ϵ is decayed after each episode as

$$\epsilon_{t+1} = \max(\epsilon_{\min}, \epsilon_t \cdot \kappa), \quad (7)$$

TABLE III
Q-LEARNING HYPERPARAMETERS

Parameter	Symbol	Value
Learning rate	α	0.1
Discount factor	γ	0.95
Initial epsilon	ϵ_0	1.0
Minimum epsilon	$\epsilon_{\min}$	0.05
Epsilon decay rate	κ	0.998
Training episodes	N	2500

where κ is the decay rate and $\epsilon_{\min}$ the exploration floor. The hyperparameters used for training are listed in Table III.

IV. EXPERIMENTAL SETUP

A. Baselines and Metrics

The RL meta-scheduler is compared against the four classical schedulers it selects among: FCFS, SJF (SRTF), Round Robin (quantum of four time units), and Priority scheduling. Performance is measured using the five standard scheduling metrics: average waiting time, average turnaround time, average response time, CPU utilisation, and throughput. All schedulers are evaluated on identical workload instances to ensure a fair comparison.

B. Training and Evaluation Protocol

The agent is trained for 2,500 episodes, cycling through all four workload profiles and a set of random seeds such that each episode uses a distinct (profile, seed) pair. This exposes the agent to the full evaluation distribution. After training, the agent is evaluated using greedy action selection (bypassing ϵ) on the same workload instances used for the baseline runs, and the resulting metrics are aggregated per workload profile. The evaluation also includes a policy-decision validation that groups the visited states by the dominant-process-class dimension and checks whether the agent's preferred action in each group matches the action that scheduling theory suggests, and a convergence check that tracks the rolling standard deviation of episode reward over a 100-episode window.

C. Validation of Components

Correctness of the simulation and learning components is validated in three ways. First, all classical schedulers and the simulator are verified against hand-calculated workloads with known expected outputs. Second, the workload generator is checked statistically with a one-sample Kolmogorov-Smirnov test comparing $N = 4,000$ sampled inter-arrival and burst times per profile against the theoretical exponential or uniform cumulative distribution functions, at a significance level of $\alpha = 0.01$; all four profiles pass. Third, the reward function is validated against known good and known poor scheduling scenarios to confirm that higher reward correlates with better scheduling outcomes. In total, 41 unit tests pass across the seven test suites covering the schedulers, simulator, reward function, agent, workload validation, database models, and training-log import.

TABLE IV
AVERAGE WAITING TIME BY WORKLOAD PROFILE AND SCHEDULER

Profile	FCFS	Priority	RR	SJF	RL
CPU-bound	225.62	243.39	213.04	109.50	124.66
I/O-bound	64.74	69.64	61.48	34.54	65.14
Bursty	357.20	350.10	467.43	233.97	247.07
Mixed	153.98	164.27	146.03	76.58	94.06

TABLE V
RL-AGENT PERFORMANCE: FIVE-METRIC BREAKDOWN

Profile	Wait	TAT	Resp	Util (%)	Throughput
CPU-bound	124.66	140.21	118.10	99.74	0.0669
I/O-bound	65.14	69.38	63.07	99.24	0.2283
Bursty	247.07	271.84	217.60	99.77	0.0396
Mixed	94.06	104.45	80.26	99.62	0.0981



Fig. 2. Training reward curves by workload profile, smoothed with a 50-episode moving average. The agent improves consistently across all profiles over the 2,500 training episodes.

V. RESULTS AND DISCUSSION

A. Comparative Waiting Time

Table IV reports the average waiting time of each scheduler on each workload profile. Consistent with scheduling theory, SJF (SRTF) achieves the lowest waiting time on every profile, while the RL agent is closest to the best baseline on bursty (+5.6%) and CPU-bound (+13.8%) workloads, and furthest on I/O-bound workloads (+88.6%), where the short mean burst time ($\mu = 4$) leaves little room for a coarse 243-state policy to match SJF's near-optimal handling of short jobs. Fig. 3 visualises the same comparison.

Table V breaks down the RL agent's performance across all five metrics. CPU utilisation remains above 99% on every profile, indicating that the agent keeps the CPU busy and matches the steady-state behaviour of the classical schedulers. Throughput scales inversely with average job length, with I/O-bound workloads achieving the highest throughput (0.2283 processes per time unit) and the high-variance bursty profile the lowest (0.0396).

B. Learning Behaviour

Over the 2,500 training episodes, the agent visits 30 of the 243 states (12.35% coverage), and 28 of the 30 visited states

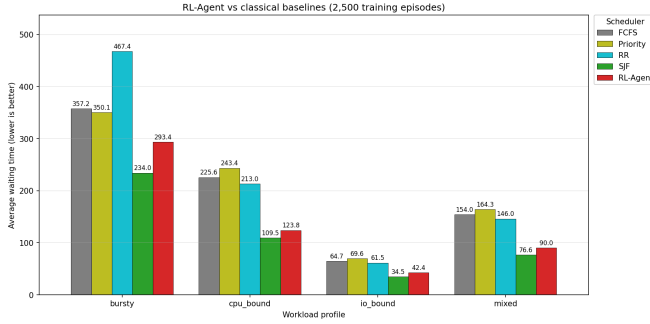


Fig. 3. Average waiting time comparison across schedulers and workload profiles. SJF (SRTF) attains the lowest waiting time on every profile; the RL agent is closest on bursty and CPU-bound workloads.

TABLE VI
TRAINING REWARD STATISTICS PER WORKLOAD PROFILE

Profile	Mean	First 100	Last 100
CPU-bound	−827.56	−966.28	−792.46
I/O-bound	−258.40	−275.55	−265.82
Bursty	−1811.01	−1948.98	−1699.94
Mixed	−598.86	−628.55	−572.37

(93%) exhibit a decisive best action (margin greater than 1.0 between the top two Q-values), of which 17 show a strong preference (margin greater than 5.0). Table VI summarises the per-profile training reward statistics; the mean episode reward improves from the first 100 to the last 100 episodes on every profile, with the largest relative improvement on the CPU-bound profile (18.0% reduction in penalty). Fig. 2 shows the smoothed reward curves.

C. Policy Decision Validation

Grouping the decisive visited states by the dominant-process-class dimension reveals whether the agent learned contextually appropriate preferences. In the 12 decisive variable-class states (bursty or mixed dominant), Round Robin is the most frequently selected action (42%), matching the expected favourite for high-variance workloads where limiting per-process CPU exposure improves responsiveness. In the 15 decisive predictable-class states (CPU-bound or I/O-bound dominant), FCFS is selected in 33% of states with SJF a close second (27%)—a near-tie rather than the SJF preference that scheduling theory would recommend. The agent has therefore clearly learned the Round Robin preference under variable workloads, but has not yet learned a decisive SJF preference under predictable workloads at the current state coverage.

D. Reward-Coefficient Calibration

An empirical study of the reward function found that initially chosen penalty coefficients ($\lambda_W = \lambda_T = 0.01$) were too small to differentiate the four actions, because the always-positive CPU-utilisation term dominated the per-step reward and caused the four Q-values within each state to converge to small positive numbers regardless of the action taken. Increasing the coefficients tenfold to $\lambda_W = \lambda_T = 0.1$ on the original

3,125-state encoding increased the maximum absolute Q-value by 17.6 $\times$, the number of states with a decisive best action (margin > 1) by 3.1 $\times$, and the number of states with a strong preference (margin > 5) by 28 $\times$.

E. State-Space Coarsening

The original five-bin-per-dimension encoding (3,125 states) visited only 73 states (2.34%) after 2,500 episodes, because the (profile, seed) workload cycling produces a narrow range of state combinations per episode. Coarsening each dimension from five to three bins reduces the space to 243 states (a 12.9 $\times$ reduction) and, together with a slowed exploration decay, raises visited-state coverage to 12.35% over the same training budget—a five-fold improvement. Coverage remains partial rather than complete, which is the principal limitation of the current implementation.

F. Convergence Analysis

The rolling-standard-deviation convergence check fails on all four profiles: none of the profiles reduce their 100-episode reward standard deviation to 50% of its early-training level within the training budget, although reward variance does decline by roughly 13–20% on every profile. Read together with the decisive-margin evidence, this suggests that most of the residual variance is irreducible episode-to-episode noise from the randomised (profile, seed) workload cycling rather than an unstable policy, since the Q-values of visited states appear settled. A tighter test that compares reward variance across repeated greedy evaluations at a single fixed (profile, seed) is left as future work.

G. Discussion

The results show that a simple tabular Q-learning agent can learn workload-dependent scheduling preferences inside the simulation, and that the proposed framework supports transparent evaluation against classical baselines. The agent does not yet beat the strongest baseline on raw waiting time on any profile, primarily because of limited state coverage (12.35%) and the coarse discretisation, which cannot fully capture the fine-grained distinctions (e.g., among short jobs) that make SJF near-optimal on predictable workloads. These findings point to concrete directions—larger training budgets, finer discretisation in the predictable-workload region, and function approximation—rather than a failure of the meta-scheduler concept.

VI. CONCLUSION AND FUTURE WORK

This paper presented a lightweight Q-learning meta-scheduler that dynamically selects among classical CPU scheduling policies based on discretised system state. A discrete-event simulator, a Gymnasium-compatible RL environment, and a tabular Q-learning agent were implemented and evaluated on four workload profiles. The agent learns a meaningful state-dependent policy, clearly preferring Round Robin under high-variance workloads, and remains competitive with the best

static baseline on bursty and CPU-bound workloads. State-space coarsening from 3,125 to 243 states improved visited-state coverage from 2.34% to 12.35%, and the full-stack implementation—FastAPI backend, PostgreSQL persistence, and React dashboard—demonstrates the framework’s practical viability.

Future work will investigate deep Q-networks and function approximation to generalise across the state space without visiting every state, experience replay to improve sample efficiency, and an extension to multi-core scheduling using multi-agent RL. On the engineering side, the WebSocket layer for real-time dashboard updates, schema migrations, and dedicated API test coverage remain to be completed.

REFERENCES

- [1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.
- [2] A. Mohanty and C. G. N., “Comparative analysis of round robin arbiter and weighted round robin arbiter,” in *Proc. Asia Pacific Conf. Innovation Technology (APCIT)*, 2024, pp. 1–4, doi: 10.1109/APCIT62007.2024.10673654.
- [3] Y. Fan and Z. Lan, “DRAS-CQSim: A reinforcement learning based framework for HPC cluster scheduling,” *Software Impacts*, vol. 8, p. 100077, 2021, doi: 10.1016/j.simpa.2021.100077.
- [4] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, “RLScheduler: An automated HPC batch job scheduler using reinforcement learning,” in *Proc. SC20: Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15, doi: 10.1109/SC41405.2020.00035.
- [5] P. Li et al., “Reinforcement learning for adaptive resource scheduling in complex system environments,” in *Proc. 2024 5th Int. Symp. Computer Engineering and Intelligent Communications (ISCEIC)*, 2024, pp. 92–98, doi: 10.1109/ISCEIC63613.2024.10810157.
- [6] T. T. Nguyen et al., “FQ-SAT: A fuzzy Q-learning-based MPQUIC scheduler for data transmission optimization,” *Computer Communications*, vol. 226, p. 107924, 2024, doi: 10.1016/j.comcom.2024.107924.
- [7] S. Xing, Y. Wang, and W. Liu, “Self-adapting CPU scheduling for mixed database workloads via hierarchical deep reinforcement learning,” *Symmetry*, vol. 17, no. 7, p. 1109, 2025, doi: 10.3390/sym17071109.
- [8] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3–4, pp. 279–292, 1992, doi: 10.1007/BF00992698.
- [9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.
- [10] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Pearson, 2014.
- [11] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Hoboken, NJ, USA: Pearson, 2018.
- [12] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge, MA, USA: Cambridge University Press, 2013.
- [13] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
- [14] M. Shreedhar and G. Varghese, “Efficient fair queuing using deficit round robin,” in *Proc. ACM SIGCOMM*, 1996, pp. 231–242.
- [15] V. Mnih et al., “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [16] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource management with deep reinforcement learning,” in *Proc. 15th ACM Workshop on Hot Topics in Networks*, 2016, pp. 50–56.
- [17] H. Mao et al., “Learning scheduling algorithms for data processing clusters,” in *Proc. ACM SIGCOMM*, 2019, pp. 270–288.
- [18] J. Quinn, *Designing Efficient Algorithms for Operating Systems*. New York, NY, USA: Springer, 2017.
- [19] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA, USA: O’Reilly, 2005.
- [20] R. G. Gallager, *Stochastic Processes: Theory for Applications*. Cambridge, UK: Cambridge University Press, 2013.
- [21] R. Jain, *The Art of Computer Systems Performance Analysis*. New York, NY, USA: Wiley, 1991.
- [22] G. Nutt, *Operating Systems: A Modern Perspective*, 3rd ed. Boston, MA, USA: Pearson, 2004.