IRSTI 50.05.15 https://doi.org/10.26577/jpcsit4120267

B. Kumalakov, D. Amangeldi*

Astana IT University, Astana, Kazakhstan
*e-mail: dilnaz1327@gmail.com

HADOOP, MULTI-AGENT SYSTEMS AND MACHINE LEARNING: EXPLORING SCALABILITY, FAULT TOLERANCE AND WORKLOAD DISTRIBUTION BEHAVIORS

Abstract. Paper reports experimental results comparing several machine learning techniques performance when distributing massively parallel computation to a set of interconnected machines. Computational resources are intentionally heterogenous to simulate real ad-hoc network environment and provide realistic setting test results. Namely Round Robin, Q-Learning and Least Loaded algorithms-based solutions are examined for their scalability, fault tolerance and workload distribution behaviors. The novelty of the paper is a set of empirical set of coefficients and bottlenecks for each implementation that is free of infrastructure specifics or error and exemption handling tools for future considerations by engineering professionals and scholars.

Keywords: Hadoop, multi-agent systems, Q-Learning, MapReduce programming, distributed computing.

1. Introduction

Efficient workload distribution plays a crucial role in optimizing computational resources and ensuring systems' high performance. However, traditional load balancing algorithms – such as round-robin or least connections – struggle to adapt to dynamically changing workloads [1], leading to resource underutilization and increased response times. Applying reinforcement learning (RL) [2] to solve the problem offers a promising direction for developing adaptive load balancing strategies which dynamically optimize resource allocation using real-time workload patterns. Unlike static approaches that rely on predefined heuristics. RL-based methods continuously learn from the environment and adjust task allocation accordingly [3].

Current work is inspired by the idea of implementing a low budget cluster platform for educational organizations, which would utilize on-campus computing devices organizing them into an ad-hock network of heterogeneous nodes. We adapt MapReduce programming model – a widely used framework for massively parallel computation – to execute used defined code on a multi-agent system (MAS) to naturally support emergent, self-configuring platform behavior and effectively function in diverse architectures.

Traditional approaches lack adaptability to workload fluctuations, leading to inefficiencies [3], and require complex configurations. Recent studies report positive effectiveness of RL-based load balancing in cloud environments. For example, RL techniques have been successfully employed to optimize task scheduling and minimize execution time, outperforming round-robin and weighted load balancing algorithms [4], [5], [6]. Furthermore, RL-based strategies demonstrated improved resource utilization in virtualized cloud infrastructures by predicting workload patterns and dynamically adjusting allocation policies [7].

Our hypothesis is that RL-powered MAS improves workload distribution in heterogeneous environments by learning optimal task allocation strategies through continuous agents' interaction, adapting both to unpredictable network topologies and to changing hardware conditions "on-the-fly."

2. Materials and Methods

Designed MapReduce system follows a coordinator-worker paradigm where a central coordinator node manages all scheduling decisions while multiple worker nodes execute the actual Map and Reduce operations. Hadoop is utilized exclusively as a distributed storage layer through

Licensed under CC BY-NC 4.0 © 2026 Al-Farabi Kazakh National University

HDFS, while all task scheduling, worker coordination, and execution control are implemented as custom components.

The separation between storage and computation layers enables full control over the scheduling policy, allowing direct integration of a Q-Learning agent into the task assignment process without constraints imposed by existing Hadoop schedulers. Inter-node communication and lifecycle management are handled through the JADE (Java Agent DEvelopment Framework) agent platform. JADE was selected for three reasons: its native support for the FIPA Agent Communication Language for structured message semantics for task dispatch and status reporting; its built-in agent container model simplifies deployment across heterogeneous nodes; and its heartbeat and fault detection mechanisms provide a foundation for implementing worker health monitoring.

2.1. Infrastructure Configuration

The experimental infrastructure is deployed in a cloud-based virtualized environment and is intentionally configured as a heterogeneous cluster to reflect realistic production scenarios, where computing nodes differ in capacity and performance. This heterogeneity is a key prerequisite for evaluating adaptive scheduling strategies, as it introduces non-uniform execution behavior across worker nodes.

The system follows a master-worker architecture. The coordinator node is deployed on a high-capacity instance and is responsible for centralized task scheduling and Q-learning state-action value updates. Its configuration provides sufficient computational resources to manage control logic and scheduling decisions without becoming a performance bottleneck for the system. The worker layer consists of eight nodes instantiated using different instance types, each representing a distinct resource profile. These instance types range from high-capacity configurations to constrained and legacy instances with reduced CPU and memory availability. Such variation reflects common cloud deployment patterns, where clusters are composed of a mix of modern, cost-efficient, and legacy virtual machines. Heterogeneous cloud environments composed of nodes with diverse resource capabilities are increasingly prevalent and pose unique challenges for efficient scheduling and resource allocation due to variability in performance characteristics across instance types [8].

As a result of these differing instance characteristics, workers exhibit varying task processing latencies under identical workloads. This creates an execution environment in which scheduling decisions have a direct impact on overall system performance. Consequently, the infrastructure enables meaningful comparison between static scheduling approaches and reinforcement learning-based workload distribution strategies.

Table 1. Infrastructure Specifications

No	Component	vCPU	RAM	Instance Type
1	Coordinator Node	2 vCPU	4 GB	High-capacity
2	Worker 1	2 vCPU	4 GB	High-capacity
3	Worker 2	2 vCPU	2 GB	Medium-capacity
4	Worker 3	2 vCPU	2 GB	Medium-capacity
5	Worker 4	2 vCPU	1 GB	Limited-capacity
6	Worker 5	2 vCPU	0.5 GB	Constrained
7	Worker 6	1 vCPU	2 GB	Legacy medium
8	Worker 7	1 vCPU	1 GB	Legacy limited
9	Worker 8	1 vCPU	0.5 GB	Legacy constrained

All nodes operate on Ubuntu Server with Linux kernel 6.x as the base operating system, selected for its stability and extensive compatibility with distributed computing frameworks. The Hadoop Distributed File System (HDFS) serves as the primary storage layer, configured with a replication factor of 2 and a block size of 128 MB. Metrics collection utilizes structured JSON logging, with subsequent analysis performed using Python-based data processing and visualization tools.

2.2. System Components and Data Flow

The system is organized into five distinct layers, each with clearly defined responsibilities. The storage layer consists of HDFS, which serves solely as a distributed file system for storing input datasets and intermediate results; it provides no computation or scheduling functionality. The coordination layer is implemented using JADE, which provides the agent communication infrastructure including asynchronous message passing, agent lifecycle management, and a directory facilitator for service discovery. The execution layer comprises a custom MapReduce runtime implemented in Java that reads input data from HDFS, partitions it into fixed-size tasks, dispatches tasks to workers via JADE messages, executes user-defined Map and Reduce functions on worker nodes, and aggregates results on the coordinator.

The scheduling layer implements three interchangeable scheduling algorithms: Round Robin, Least Loaded, and Q-Learning. The scheduling layer operates independently of the underlying execution layer, receiving worker availability signals and returning worker assignment decisions. The monitoring layer collects runtime metrics including per-task processing times, cumulative throughput, worker load distributions, and reinforcement learning statistics such as reward values and Q-table updates throughout experimental runs. Such layered architectural designs, where storage, coordination, execution, scheduling, and monitoring components are decoupled to support modularity, scalability, and independent evolution, are a characteristic practice in modern distributed and big data systems [9].

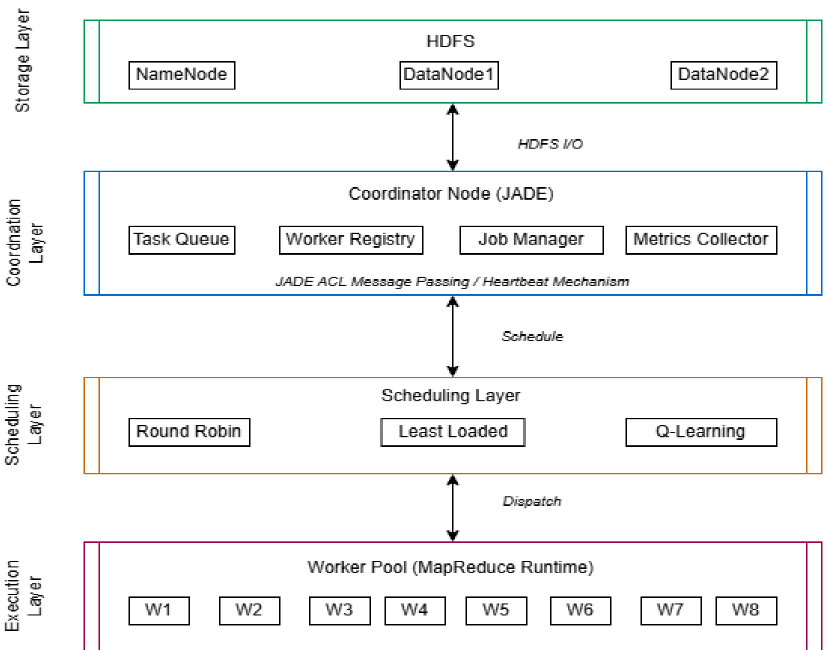

Figure 1. Layered System Architecture with Explicit Separation of HDFS and Custom Components

2.3. Runtime Execution Flow

Job execution proceeds through a well-defined sequence of phases from job submission to completion. The coordinator accepts the job request and initializes a job context object containing metadata such as job identifier, submission timestamp, and configuration parameters. The coordinator then queries the HDFS NameNode to retrieve block location metadata for the input file and constructs a task queue containing task descriptors. Each task descriptor specifies the HDFS block identifier, byte offset, and size.

Worker nodes register with the coordinator via JADE agents upon startup. The coordinator maintains a registry of active workers including their network addresses and current status (idle or

busy). Workers signal availability through periodic heartbeat messages transmitted at 5-second intervals. When the coordinator detects an available worker, it invokes the scheduling algorithm to select a worker for the next pending task. The scheduling algorithm receives the current system state as input and returns a worker identifier. For the Q-Learning scheduler, this involves a Q-table lookup or exploration action. The coordinator dispatches the task descriptor to the selected worker via a JADE ACL message.

Upon receiving a task descriptor, the worker reads the corresponding data block from HDFS, applies the user-defined Map function to each input record, and emits intermediate key-value pairs. The worker buffers these pairs locally until the task is fully processed, then transmits the intermediate results and processing time to the coordinator. The coordinator logs the processing time for metrics collection and, when using the Q-Learning scheduler, computes the reward signal and updates the Q-table. The worker is marked as available for subsequent task assignment. After all Map tasks complete, the coordinator initiates the Reduce phase by partitioning intermediate key-value pairs across workers based on key hash values. Workers apply the user defined Reduce function and return final results to the coordinator for aggregation. Upon successful completion, the coordinator writes results to HDFS and serializes the Q-table to persistent storage for use in subsequent job executions.

The coordinator node serves as the central control point for all scheduling and coordination activities. Its responsibilities are formally defined as follows. The coordinator maintains the authoritative task queue, an ordered list of task descriptors awaiting processing, with tasks dequeued in FIFO order unless the scheduling algorithm specifies otherwise. The coordinator maintains the worker registry, a data structure mapping worker identifier to their current state (idle, busy, or failed) and performance statistics (cumulative tasks processed, average processing time). The coordinator implements the scheduling interface, which accepts the current system state and returns a worker assignment; this interface is polymorphic, allowing different scheduling algorithms to be substituted without modifying other system components.

For the Q-Learning scheduler, the coordinator manages Q-table state including initialization, lookup, update, and persistence operations. The coordinator implements fault detection through heartbeat timeout monitoring; if a worker fails to send a heartbeat within the configured interval (5 seconds), it is marked as failed, its in-flight tasks are requeued, and subsequent scheduling decisions exclude the failed worker from the available action space.

2.4. Reinforcement Learning

The Q-Learning scheduler is modeled as a Markov Decision Process (MDP) described by the tuple (S, A, R, γ), where S denotes the finite set of system states, A represents the action space, R is the reward function, and $\gamma \epsilon [0,1]$ is the discount factor. The coordinator acts as the learning agent and derives a policy π : S $\rightarrow$ A that maps observed system states to task-to-worker assignment decisions based on execution feedback.

State Space. System state is encoded as a vector of four discrete features,

$$s = (f_1, f_2, f_3, f_4), \quad (1)$$

each capturing a distinct aspect of runtime conditions.

The first feature, f_1, describes load balance across workers and takes values from the set {BALANCED, MODERATE, IMBALANCED}. It is derived from the coefficient of variation of worker queue depths, computed as the ratio of standard deviation to mean queue length. Load is classified as BALANCED when this value is below 0.2, MODERATE when it lies between 0.2 and 0.5, and IMBALANCED otherwise.

The second feature, f_2, represents the system throughput level and is defined over the SET {LOW, MEDIUM, HIGH}. Current throughput, measured as the number of completed tasks per second, is evaluated relative to previously observed throughput values collected during earlier job executions. The throughput range is partitioned into three equally sized intervals based on the empirical distribution of these historical measurements. Values falling within the lowest third of observed throughput are classified as LOW, those within the middle third as MEDIUM, and those within the highest third as HIGH.

The third feature, f_3, captures performance homogeneity among workers and assumes values from {HOMOGENEOUS, MODERATE}. This feature is determined by the variability of average task processing times across workers. When the standard deviation of these averages remains below

5ms, worker performance is considered HOMOGENEOUS; higher variability results in a MODERATE classification.

The fourth feature, f_4, corresponds to the current job execution phase and takes values from {EARLY, MIDDLE, LATE, FINAL}. This phase is determined by the fraction of completed tasks, with thresholds at 25%, 50%, and 75% of total job completion.

The overall state space is defined as the Cartesian product of the four feature domains, resulting in 72 possible discrete states. Due to correlations between system metrics, only a subset of these states is observed in practice; experimental evaluation revealed 19 distinct reachable states.

Action Space. The action space corresponds to task-to-worker assignment decisions. The system maintains a fixed set of registered worker nodes $W = \{w_1, w_2 \dots, w_n\}$, where n=8 in the experiment-tal configuration. At each scheduling decision, the set of available actions depends on the current system state and includes only workers that are marked as idle and not failed in the worker registry. This state-dependent action set is defined as

$$A(s) = \{wi \in W \mid status(wi) = idle\}. \quad (2)$$

Selecting an action $a \in A(s)$ corresponds to assigning the next task to the chosen worker. The cardinality of the available action set varies dynamically over time, ranging from 1 to n, depending on worker availability.

The reward signal is computed upon task completion and is defined as the inverse of task processing time, $r = R(s, a) = \frac{1}{t}$, where t denotes the execution time in milliseconds. This formulation assigns higher rewards to faster task completions and implicitly penalizes assignments that lead to longer processing delays due to resource contention or worker overload. Under observed operating conditions, reward values ranged between r ϵ [0.004, 0.067].

The Q-table update follows the temporal difference learning rule adapted for worker selection:

$$\begin{gathered} Q_w(s) \leftarrow Q_w(s) + \\ +\alpha[r + \gamma max Q_{w'}(s') - Q_w(s)] \end{gathered} \quad (2)$$

where w denotes the selected worker, s is the state at task assignment time, r is the observed reward (inverse processing time), s' is the state after task completion, α = 0.1 is the learning rate controlling update magnitude, and γ = 0.95 is the discount factor reflecting the importance of future rewards. The subscript w ∈ A emphasizes that actions correspond to worker selection from the available worker set.

The agent employs an ε-greedy policy for action selection. With probability ε, a random available worker is selected (exploration); with probability 1-ε, the worker with the highest Q-value for the current state is selected (exploitation). The exploration rate ε is initialized to 0.3 and decays exponentially toward a minimum of 0.02 over successive job executions. This decay schedule permits broad exploration during initial training while converging toward exploitation of learned policies in later runs. This approach to reinforcement learning-based task scheduling, including the definition of rewards and the use of Q-learning with ε-greedy action selection and temporal-difference updates, has been widely adopted in dynamic task allocation problems in distributed and cloud computing systems [10].

The Q-table is serialized to disk upon job completion and loaded at the start of each subsequent job. This persistence mechanism enables the agent to accumulate knowledge across multiple job executions, progressively refining its scheduling policy based on observed worker performance patterns.

2.5. Baseline Scheduling Algorithms

Two baseline scheduling algorithms are implemented for comparative evaluation. The Round Robin algorithm assigns tasks to workers in a fixed cyclic order, advancing to the next worker in the sequence after each assignment regardless of worker load or performance characteristics. This algorithm provides a simple baseline that makes no attempt to adapt to system conditions. Round Robin scheduling is commonly used as a baseline in distributed and cloud computing studies due to its simplicity and deterministic task assignment behavior, despite its lack of awareness of workload imbalance or performance heterogeneity [11]. The Least Loaded algorithm assigns each task to the worker with the fewest tasks currently in its processing queue. This heuristic-based approach adapts to instantaneous load conditions but does not incorporate historical performance data or predictive models. Load-based heuristics such as Least Loaded (or Least Connection) scheduling are widely adopted in distributed systems as lightweight adaptive baselines, as they react to current queue

states but remain limited by the absence of learning or long-term performance modeling [12]. Both baseline algorithms are implemented as instances of the same scheduling interface used by the Q-Learning scheduler, ensuring identical integration with the coordination and execution layers.

3. Results

The experimental evaluation implements three distinct evaluation scenarios: scalability analysis examining performance across different worker configurations, fault tolerance behavior under worker failure conditions, and Q-Learning algorithm dynamics including reward progression and policy evolution over multiple training runs.

All experiments utilize a standardized text corpus stored in HDFS. The dataset comprises 22,185,205 text lines partitioned into 4,438 tasks (approximately 5,000 lines per task), representing a total uncompressed size of 1.8 GB. The workload consists of a word frequency counting MapReduce job where the Map phase tokenizes input lines and emits (word, 1) key-value pairs, while the Reduce phase aggregates counts for each unique word. This canonical MapReduce application provides consistent computational characteristics across all experimental runs, enabling fair comparison between scheduling algorithms without introducing variance from workload heterogeneity.

3.1. Scalability Analysis

The scalability experiment evaluates system throughput and processing time across four worker configurations (Table 2).

Table 2. Scalability Experiment Results

No	Number of workers	Algorithm	Time	Throughput (lines/s)	Speedup
1	1	Round Robin	195.26	113,613	1.00×
2	1	Least Loaded	185.48	119,608	1.00×
3	1	Q-Learning	169.79	130,661	1.00×
4	2	Round Robin	92.38	240,158	2.11×
5	2	Least Loaded	86.39	256,819	2.15×
6	2	Q-Learning	77.00	288,119	2.21×
7	4	Round Robin	51.47	431,075	3.79×
8	4	Least Loaded	47.74	464,667	3.88×
9	4	Q-Learning	41.13	539,353	4.13×
10	8	Round Robin	40.09	553,415	4.87×
11	8	Least Loaded	36.26	611,817	5.11×
12	8	Q-Learning	33.57	660,962	5.06×

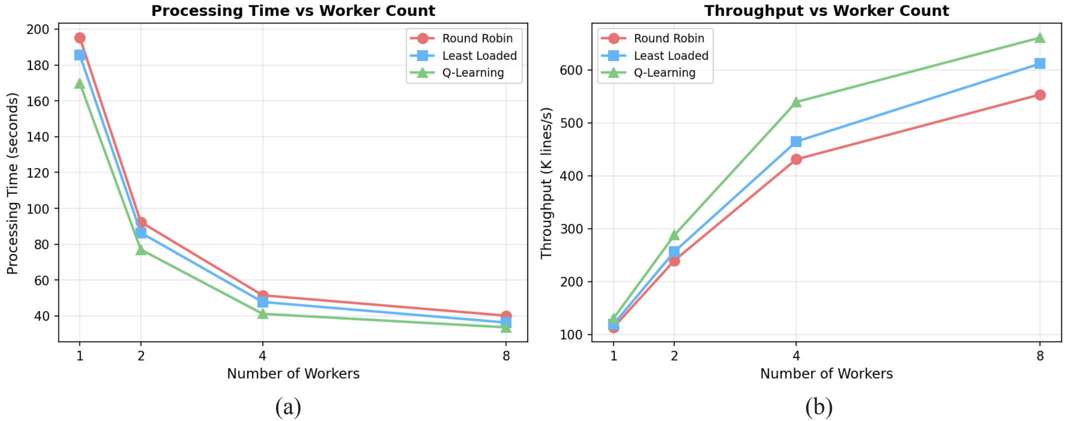

Figure 2. Processing Time and Throughput Comparison Across Worker Configurations

In the single-worker configuration, Q-Learning completed the workload in 169.79 seconds with throughput of 130,661 lines per second, compared to 195.26 seconds (113,613 lines/s) for Round Robin and 185.48 seconds (119,608 lines/s) for Least Loaded. As the worker count increased to two, all algorithms exhibited near-linear speedup: Q-Learning achieved 2.21× acceleration, while Round Robin achieved 2.11× and Least Loaded achieved 2.15×. The throughput gap widened in absolute terms, with Q-Learning processing 288,119 lines per second versus 240,158 for Round Robin.

With four workers, Q-Learning achieved throughput of 539,353 lines per second with a speedup factor of 4.13×. Round Robin achieved only 3.79× speedup at this scale, attributed to load imbalance caused by worker heterogeneity. The eight-worker configuration represents full cluster utilization. Q-Learning achieved throughput of 660,962 lines per second, processing the dataset in 33.57 seconds. Least Loaded reached 611,817 lines per second (36.26 seconds), while Round Robin achieved 553,415 lines per second (40.09 seconds). Speedup efficiency decreased for all algorithms at eight workers (ranging from 4.87× to 5.11× against a theoretical maximum of 8×), indicating the presence of coordination overhead and communication latency that becomes proportionally more significant as parallelism increases.

3.2. Fault Tolerance Evaluation

The evaluation simulates a failure scenario where two worker nodes (workers 3 and 6) experience simultaneous failures at the 25-second mark during an eight-worker job execution. The failure was induced by terminating the worker processes via SIGKILL signals, representing abrupt failures without shutdown sequences. Each scheduling algorithm was evaluated under identical failure conditions.

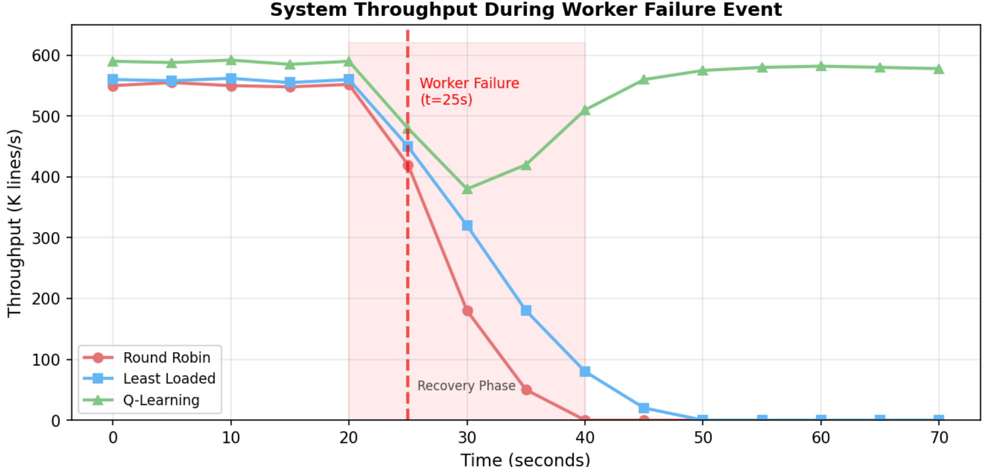

Figure 3. System throughput trajectory of each algorithm before, during, and after the failure event

Figure 3 illustrates the throughput trajectory of each algorithm before, during, and after the failure event. Prior to the failure at t=25s, all three algorithms operated at their respective steady-state throughputs: approximately 590K lines/s for Q-Learning, 560K lines/s for Least Loaded, and 550K lines/s for Round Robin. Upon failure detection, the algorithms exhibited markedly different recovery behaviors.

The Round Robin algorithm failed to recover following worker loss because it maintains a static assignment sequence, tasks dispatched to failed workers (3 and 6) entered a timeout queue. The algorithm lacks mechanisms to redistribute these tasks to healthy workers, resulting in blocking and eventual job failure necessitating manual intervention. The Least Loaded algorithm detected worker unavailability through heartbeat timeouts (configured at 5-second intervals). However, its recovery mechanism proved insufficient for sustained operation. When failed workers stopped responding, Least Loaded removed them from the

active worker pool but encountered difficulties redistributing in-flight tasks, leading to cascading timeouts and system stall by t=50s.

The Q-Learning scheduler exhibited different behavior through its adaptive policy mechanism. Upon detecting worker failures via heartbeat timeouts at t=30s, the agent updated its state representation to reflect the reduced worker pool and adjusted its action space accordingly. The learned Q-values for failed workers were masked, preventing future assignments. The system experienced a throughput dip to approximately 380K lines/s during the recovery phase (t=30-40s) as the coordinator redistributed pending tasks, then recovered to 580K lines/s by t=60s, completing the job with zero task loss.

Table 3. Fault Tolerance Metrics Summary

No	Metric	Round Robin	Least Loaded	Q-Learning
1	Failure Detection Time	N/A (no detection)	5.2 seconds	4.8 seconds
2	Recovery Initiated	No	Partial	Yes
3	Time to Stable Operation	Failed	Failed	35 seconds
4	Job Completion	Failed	Failed	Success
5	Post-Recovery Throughput	0 lines/s	0 lines/s	580,1 lines/s
6	Tasks Lost	1,847	1,203	0

3.3. Q-Learning Implementation Performance

The evaluation tracks average reward progression, state space exploration, and the emergence of scheduling policies across thirteen consecutive job executions.

The average reward metric, computed as the mean of all rewards received during each job, improved from an initial value of 0.12 during the first run to a stabilized value of approximately 0.195 by the final runs, representing a 62.5% improvement. This reward increases correlates with observed throughput gains, as higher rewards indicate faster task completion times. The state exploration curve demonstrates rapid initial discovery of the state space, achieving complete coverage of 19 reachable states by the fourth training run and maintaining stable state visitation patterns thereafter. The Q-table accumulated 26,622 total updates across these 19 explored states.

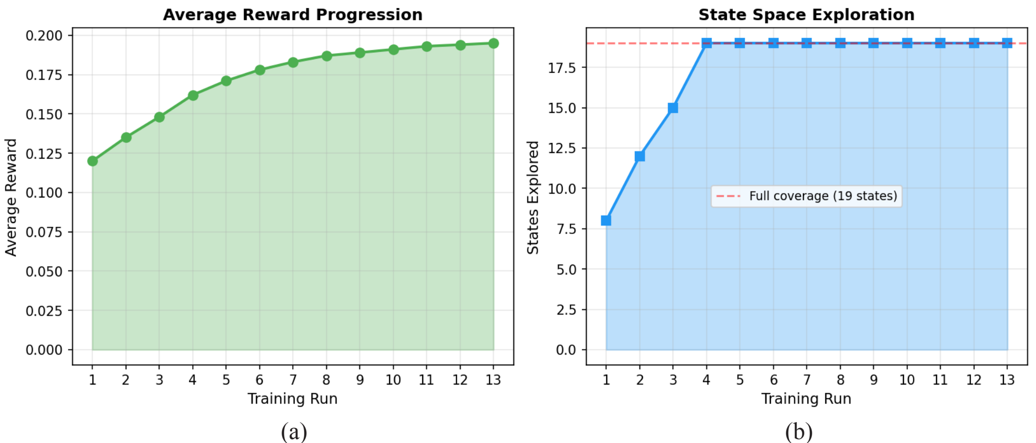

Figure 4. Q-Learning training progress over multiple job executions

Analysis of Q-table evolution reveals the emergence of worker preference patterns aligned with actual processing capabilities. Table 4 presents the top Q-values extracted from the final training state. The learned policy exhibited context-dependent preferences: under high-load IMBALANCED states, the agent assigned higher Q-values to certain workers that performed better under congestion conditions despite slower average speeds under normal operation.

Table 4. Top Learned Q-Values from Final Training State

No	System State	Action (Worker)	Q-Value
1	[IMBALANCED, HIGH, HOMOGENEOUS, EARLY]	worker5	+1.9926
2	[IMBALANCED, HIGH, HOMOGENEOUS, EARLY]	worker7	+1.9564
3	[IMBALANCED, HIGH, HOMOGENEOUS, EARLY]	worker2	+1.9360
4	[IMBALANCED, HIGH, HOMOGENEOUS, EARLY]	worker1	+1.9076
5	[IMBALANCED, HIGH, MODERATE, EARLY]	worker2	+1.8418
6	[IMBALANCED, HIGH, MODERATE, EARLY]	worker1	+1.8417
7	[BALANCED, HIGH, HOMOGENEOUS, EARLY]	worker2	+1.4302
8	[IMBALANCED, HIGH, HOMOGENEOUS, MIDDLE]	worker7	+1.5191

3.4. Load Distribution Analysis

Further, to understand workload distribution behavior, task distribution patterns of each algorithm implementation are analyzed in the eight-worker configuration (Figure 5).

Round Robin produced load variance with standard deviation σ = 0.63%, while Least Loaded achieved better balance (σ = 0.56%) by considering current queue depths. Q-Learning achieved load distribution with σ = 0.89%.

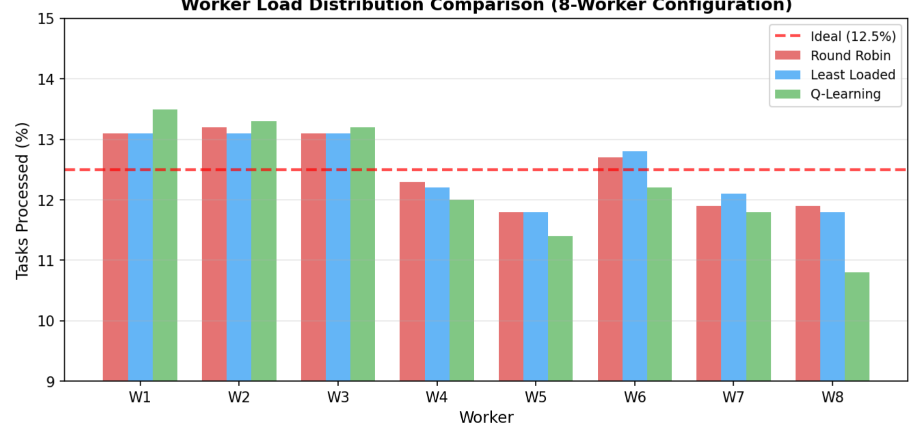

Figure 5. Worker load distribution comparison (8-worker configuration) shows the percentage of total tasks processed by each worker, with the dashed line indicating the ideal uniform distribution of 12.5% per worker

4. Discussion

Experiments show that implemented Q-Learning achieved peak throughput of 660,962 lines per second with eight workers, completing the workload in 33.57 seconds. Least Loaded achieved 611,817 lines per second (36.26 seconds) and Round Robin achieved 553,415 lines per second (40.09 seconds). The processing time reduction from 40.09 seconds to 33.57 seconds represents a 16.3% efficiency difference.

Speedup efficiency for all algorithms decreased at higher worker counts relative to theoretical linear scaling, indicating coordination overhead becomes proportionally significant as parallelism increases. In the fault tolerance experiments, Q-Learning recovered from 25% worker loss (2 of 8 workers) with zero task loss, returning to 98% of pre-failure

throughput within 35 seconds of failure detection. Both Round Robin and Least Loaded failed to complete the job under identical failure conditions, with Round Robin losing 1,847 tasks and Least Loaded losing 1,203 tasks before system stall.

The Q-Learning agent learned worker performance characteristics through the reward mechanism without explicit configuration. Q-values naturally encoded speed differentials across heterogeneous nodes, with higher Q-values accumulating for workers that consistently completed tasks faster. This learning occurred through 26,622 Q-table updates across 19 explored states over 13 training runs, with average reward increasing from 0.12 to 0.195 (62.5% improvement).

The learned policy exhibited context-dependent behavior, preferring different workers based on system state. Under "imbalanced" load conditions, the agent assigned higher Q-values to workers 5 and 7, which performed better under congestion despite slower average speeds during normal operation. This contextual awareness emerged from the reinforcement learning process without supervised training data.

Load distribution analysis revealed that Q-Learning produced intentional imbalance (σ = 0.89%) compared to Round Robin (σ = 0.63%) and Least Loaded (σ = 0.56%). The agent learned to assign more tasks to faster workers (13.2-13.5% for W1-W3) and fewer to slower workers (10.8-11.4% for W5, W8). This finding suggests that optimal scheduling in heterogeneous environments may not correspond to uniform work distribution.

Fault tolerance behavior arose from the adaptive policy mechanism. Upon detecting worker failures, the agent excluded failed workers from its action space through Q-value masking and redistributed work through natural policy adaptation. This represents emergent resilience rather than explicitly programmed fault handling.

The tabular Q-Learning approach requires state discretization, potentially losing continuous feature information that could improve scheduling precision. The evaluation used a single workload type on an eight-worker cluster; the generalizability of these findings to larger clusters and diverse workload patterns remains to be established. The single coordinator node represents a potential scalability constraint for larger deployments.

5. Conclusions

A Q-Learning-based task scheduling mechanism was implemented and evaluated for distributed MapReduce systems. The system architecture separates HDFS as a storage-only layer from a custom MapReduce execution runtime with interchangeable scheduling algorithms. The experimental evaluation was conducted on a heterogeneous cluster of eight worker nodes processing 22.2 million text lines across 4,438 tasks.

The tabular Q-Learning approach requires state discretization, potentially losing continuous feature information that could improve scheduling precision. The evaluation used a single workload type on an eight-worker cluster; the generalizability of these findings to larger clusters and diverse workload patterns remains to be established. The single coordinator node represents a potential scalability constraint for larger deployments.

Author Contributions

Specific roles and contributions are as follows: Conceptualization, B.K.; Methodology, B.K.; Software, D.A.; Validation, B.K. and D.A.; Resources, D.A.; Writing (Original Draft Preparation), D.A.; Writing (Review & Editing), B.K.; Visualization, D.A.; Supervision, B.K.; Project Administration, B.K..

Conflicts of Interest

The authors declare no conflict of interest.

References

1. M. Mitchell, *Artificial Intelligence: A Guide for Thinking Humans*. New York, NY, USA: Farrar, Straus and Giroux, 2019.
2. P. Winder, *Reinforcement Learning: Industrial Applications of Intelligent Agents*. Sebastopol, CA, USA: O'Reilly Media, 2020.
3. K. Chawla, "Reinforcement learning-based adaptive load balancing for dynamic cloud environments," *arXiv preprint* arXiv:2409.04896, 2024, doi: 10.48550/arXiv.2409.04896.
4. H. A. Abbass, "Editorial: What is artificial intelligence?," *IEEE Transactions on Artificial Intelligence*, vol. 2, no. 2, pp. 94–95, 2021, doi: 10.1109/TAI.2021.3096243.

5. M. Ghasemi, A. H. Moosavi, I. Sorkhoh, A. Agrawal, F. Alzhouri, and D. Ebrahimi, "An introduction to reinforcement learning: Fundamental concepts and practical applications," *arXiv preprint* arXiv:2408.07712, 2024, doi: 10.48550/arXiv.2408.07712.
6. M. A. Shahid, M. M. Alam, and M. M. Su'ud, "Performance evaluation of load-balancing algorithms with different service broker policies for cloud computing," *Applied Sciences*, vol. 13, no. 3, Art. no. 1586, 2023, doi: 10.3390/app13031586.
7. Y. Li, "Reinforcement learning in practice: Opportunities and challenges," *arXiv preprint* arXiv:2202.11296, 2022.
8. H. Cui, J. Sheng, B. Jin, Y. Hu, L. Su, L. Zhu, W. Zhou, and X. Wang, "ReAssigner: A plug-and-play virtual machine scheduling intensifier for heterogeneous requests," in *Proc. 2022 IEEE Int. Conf. Big Data (Big Data)*, Osaka, Japan, 2022, pp. 3726–3734, doi: 10.1109/BigData55660.2022.10021058.
9. L. Y. Contreras Rivas, E. López Domínguez, Y. Hernández Velázquez, S. Domínguez Isidro, M. A. Medina Nieto, and J. De La Calleja, "A layered software architecture for the development of smart mobile distributed systems oriented to the management of emergency cases," *Applied Sciences*, vol. 15, no. 7, Art. no. 3664, 2025, doi: 10.3390/app15073664.
10. Y. Wang, S. Dong, and W. Fan, "Task scheduling mechanism based on reinforcement learning in cloud computing," *Mathematics*, vol. 11, no. 15, Art. no. 3364, 2023, doi: 10.3390/math11153364.
11. V. S. Praditha, T. S. Hidayat, M. A. Akbar, and H. Fajri, "A systematical review on round robin as task scheduling algorithms in cloud computing," in *Proc. 2023 6th Int. Conf. Inf. Commun. Technol. (ICOIACT)*, Yogyakarta, Indonesia, 2023, pp. 516–521, doi: 10.1109/ICOIACT59844.2023.10455832.
12. T. W. Harjanti, H. Setiyani, and J. Trianto, "Load balancing analysis using round-robin and least-connection algorithms for server service response time," *Applied Technology and Computing Science Journal*, vol. 5, no. 2, pp. 40–49, 2022, doi: 10.33086/atcsj.v5i2.3743.

Information about Authors:

Bolatzhan Kumalakov, PhD. Dr. Bolatzhan Kumalakov is an Associate Professor at Astana IT University (Astana, Kazakhstan, e-mail: bolatzhan.kumalakov@astanait.edu.kz). He received his PhD in Computer Science from Al-Farabi Kazakh National University in 2014. Dr. Kumalakov has over 15 years of experience in software engineering, distributed computing, artificial intelligence and machine learning. His research interests include applying machine learning and data mining to solve computational problems in multiple domains. He is a member of the Institute of Electrical and Electronics Engineers (IEEE).

Dilnaz Amangeldi is a junior researcher at Astana IT University, School of Artificial Intelligence and Data Science (Astana, Kazakhstan, e-mail: dilnaz1327@gmail.com). Her academic interests include artificial intelligence, machine learning, and data-driven technologies.

Submission received: 31 January, 2026.
Revised: 18 February, 2026.
Accepted: 20 February, 2026.