Open Access Research Journal of
Science and Technology
Journals home page: https://oarjst.com/
ISSN: 2782-9960 (Online)

OARJ OPEN ACCESS RESEARCH JOURNALS

(RESEARCH ARTICLE)

Check for updates

A FORMALLY VERIFIED AI FRAMEWORK FOR SCHEDULING OPTIMISATION IN SLURM-BASED DISTRIBUTED HPC SYSTEMS

Robert W. Bakyayita [1, *], Brian Kasozi [2] and Denise Joanita Birabwa [3]

[1] Friedrich-Alexander-Universität Erlangen-Nurenberg (FAU), Germany Department of Artificial Intelligence in Biomedical Engineering.
[2] Uganda Martyrs University, Nkozi Department of Computer and Information Systems.
[3] Kyambogo University, Uganda Department of Electrical and Electronics Engineering.
* Corresponding Author

ORCID Details
Robert W. Bakyayita: 0009-0001-8580-9196
Brian J Kasozi: 0000-0001-6569-3969
Denise Joanita Birabwa: 0000-0002-9477-1556

Open Access Research Journal of Science and Technology, 2026, 17(02), 099–108

Publication history: Received on 04 July 2026; revised on 11 August 2026; accepted on 13 August 2026

Article DOI: https://doi.org/10.53022/oarjst.2026.17.2.0076

ABSTRACT

This paper presents an artificial intelligence (AI)-enabled scheduling framework for Slurm that integrates formal verification techniques with data-driven optimization methods. The proposed architecture combines concepts from queueing theory, Markov chain modeling, graph theory, mathematical optimization, reinforcement learning, and swarm intelligence to support adaptive scheduling decisions while preserving system correctness and reliability. By unifying these complementary approaches, the framework enhances scheduling efficiency, improves resource utilization, and increases operational robustness in dynamic HPC environments.

Experimental evaluation demonstrates that the proposed framework consistently outperforms conventional scheduling approaches across multiple performance metrics. Specifically, it achieves higher job throughput, shorter queue waiting times, improved resource utilization, and stronger fault tolerance, highlighting its effectiveness for next-generation HPC workload management.

Keywords: High-Performance Computing (HPC), Slurm Workload Manager, Distributed Computing, Parallel Computing, Cluster Com-putting, Artificial Intelligence, Formal Verification, Queueing Theory, Markov Chain Modeling, Graph Theory, Reinforcement Learning, Swarm Intelligence, Resource Scheduling, Fault-Tolerant Systems, Scheduling Optimisation.

1 INTRODUCTION

High-Performance Computing has become the backbone of modern science and engineering. Climate modeling, genomics, large-scale AI training, and materials simulation all depend on HPC systems solving problems that were unthinkable a decade ago. An uncomfortable truth, however, lies beneath that raw power: the scheduler that decides which job runs on which machine still operates on heuristics designed for a simpler era.

Slurm is the most widely adopted workload manager in current use [1]. It is reliable, flexible, and extensively validated. Yet as clusters scale to tens of thousands of nodes and job submissions become increasingly heterogeneous and bursty, Slurm's default scheduling algorithms exhibit limitations. Long queue delays are observed, nodes remain idle while jobs wait for resources that never become available, and administrators manually adjust parameters based on intuition rather than data. More concerningly, no formal guarantee exists that a scheduling decision will not violate system constraints or trigger cascading failures.

The approach presented here is motivated by a simple observation: the mathematical and computational tools for substantial improvement already exist. Queueing theory provides a language for describing job arrival and service processes [2]. Markov chains enable modeling state transitions in the scheduler as probabilistic events [3]. Graph theory supports representation of job dependencies, data locality, and network topology [4]. Optimization theory offers frameworks linear programming, integer programming, and convex optimization to formulate scheduling as a problem of maximizing utilization while minimizing wait times [5]. Reinforcement learning enables learning optimal policies from experience without explicitly modeling every system detail [6]. Swarm intelligence, particularly particle swarm optimization and ant colony optimization, can explore scheduling spaces that would paralyze traditional solvers [7], [8].

Performance, however, is only half the equation. On a research cluster, a misbehaving scheduler might slow a simulation; on a production system controlling medical imaging or autonomous vehicle training, the same bug could have serious consequences. Scheduling optimization is therefore not treated as a purely numerical problem. Formal software verification model checking, invariant analysis, and temporal logic specifications is introduced to prove, where possible, that scheduling policies respect safety properties, avoid deadlocks, and guarantee fairness [9], [10].

The proposed framework does not discard Slurm and start from scratch. Instead, AI-driven decision modules and verification layers are embedded into existing Slurm components. The scheduler continuously observes job queues, node states, and historical patterns. A reinforcement learning agent with a reward function shaped by queuing-theoretic metrics proposes candidate allocations. Swarm optimization runs in the background, exploring alternative schedules and surfacing improvements. A verification layer translates scheduling decisions into transition systems and checks them against formally specified invariants, such as "no job waits indefinitely if resources become available" and "memory limits are never exceeded across co-located jobs."

A prototype was built and tested on a modest but realistic cluster environment. Compared to Slurm's default backfill and priority-based algorithms, the proposed approach shows measurable gains: 47% higher throughput, shorter waiting times for short and medium jobs, and better resource utilization under heavy load. More importantly, the verification component identified three subtle configuration issues in the test setup that could have led to job starvation under specific conditions issues that the baseline Slurm configuration allowed without warning.

The costs of AI-driven scheduling are acknowledged. Inference time for the reinforcement learning agent, periodic exploration for swarm optimization, and added latency for verification all introduce overhead. These are addressed through a hybrid design: fast heuristic decisions for routine submissions, and deeper AI-plus-verification cycles for complex, resource-intensive, or high-priority jobs. The system learns to classify job types and apply the appropriate level of reasoning.

The remainder of this paper is organized as follows. Section II reviews related work at the intersection of HPC scheduling, AI, and formal methods. Section III presents the mathematical formulation of the scheduling model using queuing theory and Markov chains. Section IV describes the reinforcement learning and swarm intelligence algorithms adapted for Slurm. Section V explains the verification layer, including temporal logic specifications and the model checking approach. Section VI reports the experimental setup and results. Section VII concludes with limitations and directions for future work.

2 RELATED WORK

Efforts to improve HPC scheduling have spanned decades. Most fall into one of two categories: elegant theory that never runs on a real Slurm cluster, or practical heuristics that work well but lack theoretical guarantees. The work presented here sits between these extremes, attempting to achieve the best of both.

The earliest serious attempts originated from operations research. Job-to-node assignment was formulated as mixed-integer programming problems and solved using tools like CPLEX or Gurobi [11]. For small clusters with a handful of nodes and predictable jobs, this approach works well. However, with a thousand nodes and jobs arriving at random times, the solver either takes an impractical amount of time or fails to converge. Feitelson and Rudolph identified this scalability limitation in the late 1990s [12], and the problem remains unresolved.

Subsequently, heuristics emerged. First-fit, best-fit, and worst-fit for memory allocation. Back-filling, which was innovative when introduced in the early 2000s [13]. Conservative backfill versus aggressive backfill. Fair-share schedulers that prevent any single user from monopolizing the cluster [14]. These constitute Slurm's default configuration today. They are fast, predictable, and easy to explain to system administrators. They are also blind: a backfill scheduler cannot look more than one job ahead and has no memory of past traffic patterns.

Machine learning entered the field around 2015. Supervised learning was initially used to estimate job runtime more accurately, as Slurm's default runtime estimates are notoriously poor. Random forests and gradient boosting machines were trained on historical job logs [15]. This helped, but addressed only a single parameter. More ambitious projects explored reinforcement learning. Mao and colleagues at Berkeley demonstrated that RL could learn scheduling policies outperforming hand-tuned heuristics in simulated environments [16]. Simulation, however, is not reality. When deployment on real clusters was attempted, the same problems emerged: slow inference, non-stationary workloads, and the possibility that the RL agent might learn to starve certain job classes if starvation improved its reward.

Swarm intelligence has a smaller but dedicated following. Particle swarm optimization has been applied to static scheduling problems where all jobs are known in advance [17]. Ant colony optimization, with its pheromone trail metaphor, maps naturally to job placement decisions [18]. The limitation is that swarm methods require many iterations to converge, and each iteration evaluates candidate schedules against an objective function. For dynamic scheduling with jobs arriving continuously, running a full swarm optimization before every decision is infeasible. Some researchers have proposed running swarm optimization in a background thread and updating a policy cache [19], though convincing large-scale results have not been demonstrated.

Formal verification in scheduling is even rarer. Model checking has been used for real-time systems and embedded controllers for years, with excellent tools available: SPIN, NuSMV, and TLA+ [20]–[22]. Applying them to a Slurm scheduler, however, requires modeling a system with thousands of states, nondeterministic job arrivals, and time-varying resource availability. The state space explodes. Work from the European Space Agency verified a simplified scheduler for satellite tasking [23], but that scheduler handled only a few dozen jobs. Other researchers used abstract interpretation to prove memory safety properties of scheduler code [24], which is not equivalent to proving scheduling correctness.

The approach presented here does not claim to solve all these problems. It is pragmatic: reinforcement learning is used for the core scheduling policy because it adapts to workload patterns; swarm optimization is used offline to discover good initial policies and explore alternatives; queuing theory and Markov chains are used to build a compact mathematical model that keeps the RL state space manageable; and formal verification is applied only to critical properties on simplified abstractions. This is bounded verification, not perfect verification, but it catches many mistakes that would otherwise reach production.

3 MATHEMATICAL FORMULATION

The scheduling problem is formally formulated in this section.

3.1 System Model

This subsection presents the concrete parameters of the simulation environment. All values reported here are derived directly from the implementation that executed on the Apple M5 Pro-based test system.

Cluster Configuration Consider a cluster with $M = 50$ compute nodes. This number was chosen because it is the largest configuration that fits comfortably within 24GB of RAM while leaving sufficient memory for the verification layer and Python runtime overhead. Each node i has:

- **CPU (Central Processing Unit) cores:** $c_i = 16$ virtual cores. This value was selected because most research clusters allocate between 16 and 32 cores per node, and 16 cores enabled realistic job packing while keeping the state space manageable.
- **Memory (RAM):** $m_i = 64$ gigabytes (GB). This matches typical medium-sized HPC (High-Performance Computing) nodes. The Z3 SMT (Satisfiability Modulo Theories) solver's integer variables scaled linearly with memory, and 64GB per node kept the SMT formulas under 20GB.
- **GPU (Graphics Processing Unit) units:** $g_i = 2$ on a subset of nodes. In this simulation, 10 of the 50 nodes had GPUs. The remaining 40 nodes were CPU-only. This imbalance forced the scheduler to make intelligent placement decisions.

Resources are divisible only to certain granularities. A CPU core is the smallest allocatable unit; half-cores cannot be allocated. Memory is allocated in megabyte (MB) chunks. GPUs are discrete devices and cannot be shared across jobs. These constraints match the behavior of real Slurm (Simple Linux Utility for Resource Management) clusters.

Job Arrival Model Jobs arrive according to a non-homogeneous Poisson process. The arrival rates were calibrated using the DAS-2 trace [12], which contains real job submissions from a Dutch super-computing center.

Each job j is described by a tuple (a_j, r_j, d_j, t_j) where:

- a_j is the arrival time (when the job enters the queue). In the simulator, this is a timestamp measured in seconds from the start of the experiment.
- r_j is the requested resource vector (cpu_j, mem_j, gpu_j). From the DAS-2 trace, three job classes were derived:
- **Short jobs (60% of workload):** $cpu_j \in [1, 4]$, $mem_j \in [4, 16]$ GB, $gpu_j = 0$. These are typically test runs or single-node simulations.
- **Medium jobs (30% of workload):** $cpu_j \in [4, 8]$, $mem_j \in [16, 32]$ GB, $gpu_j \in [0, 1]$. These include data processing and moderate parallel workloads.
- **Long jobs (10% of workload):** $cpu_j \in [8, 16]$, $mem_j \in [32, 64]$ GB, $gpu_j \in [0, 2]$. These correspond to production simulations or large-scale model training runs.
- d_j is the duration how long the job would run if uninterrupted. Durations range from 30 seconds for short jobs to 12 hours for long jobs. The DAS-2 (Distributed Amsterdam Supercomputer 2) distribution was truncated at 12 hours because each experiment was limited to 6 hours of simulated time.
- t_j is the task graph for parallel jobs a DAG (Directed Acyclic Graph) showing dependencies between subtasks. For jobs requesting more than 4 CPU cores, a random DAG with 2 to 8 tasks was generated. For smaller jobs, a single task was assumed. The DAG edges represent communication volume, which affects placement decisions.

Scheduling Events and Decision Frequency: The scheduler's job is to decide, at each scheduling event, which jobs to start on which nodes, and in what order. A scheduling event occurs whenever:

- **A job finishes:** This happens every 30 to 300 seconds on average, depending on the workload mix. The Rust simulator detected completions by decrementing job durations each time step.
- **A new job arrives:** Arrivals were configured every 10 seconds on average, with a coefficient of variation of 1.5 to create bursty traffic. This is closer to real HPC behavior than a smooth Poisson process.
- **A periodic idle check:** Every 5 seconds with no events, the scheduler runs a lightweight backfill scan. This prevents nodes from sitting idle when jobs are waiting but no completion has occurred.

In the Python simulator, time was discretized into 1-second steps. This is coarse but sufficient; most HPC scheduling decisions happen at the second scale, not the millisecond scale. The Rust simulator used the same discretization for consistency.

Concrete Parameters from the Simulation: Table I shows the exact parameters used. These are the values that produced stable learning and meaningful comparisons.

Table 1: System model parameters used in simulation

Parameter	Value
Number of nodes (M)	50
CPU cores per node (c_i)	16
Memory per node (m_i)	64 GB
GPU-enabled nodes	10 (20%)
GPUs per GPU node (g_i)	2
Arrival rate (mean)	0.1 jobs/second
Arrival burst factor	1.5 (coefficient of variation)
Short job proportion	60%
Medium job proportion	30%
Long job proportion	10%
Short job duration range	30 sec – 5 min
Medium job duration range	5 min – 2 hours
Long job duration range	2 hours – 12 hours
Time step granularity	1 second
Scheduling decision overhead	89 ms (full framework)

Parameter Selection Rationale: The rationale for each parameter choice is as follows.

50 nodes: The 24GB memory limit on the test system constrained the maximum feasible cluster size. At 50 nodes, the Python environment storing node states, job queues, and verification logs consumed approximately 18GB. Increasing the cluster to 100 nodes exceeded 22GB and triggered swapping, which degraded simulation performance by a factor of 7. The Rust simulator could handle 500 nodes, but this required optimized compilation and the elimination of Python runtime overhead.

16 cores per node: This value represents a balance between scheduler stress and state space growth. Fewer cores (e.g., 8) would not sufficiently stress the scheduler. More cores (e.g., 32) would combinatorially expand the action space. At 50 nodes with the RL agent's state vector dimension of $3 + 3M + 30 \approx 183$, the 128-unit hidden layer provided adequate capacity without over parameterization.

64GB memory per node: The Z3 SMT solver allocates symbolic integers per memory unit. With 64GB per node, the abstract model has $50 \times 64 = 3200$ memory variables. The solver handled this in under 100ms for most queries. At 128GB per node, the same queries took 300-500ms too slow for periodic verification.

Job size distribution (60/30/10): This distribution was derived from analyzing 10,000 random jobs in the DAS-2 trace. Short jobs dominate HPC workloads, but long jobs consume most of the resources. This imbalance is exactly what makes scheduling difficult. A naive scheduler prioritizes short jobs and starves long ones. The verification layer caught this pattern twice.

Burst factor of 1.5: In a pure Poisson process, arrivals are memoryless. Real HPC clusters exhibit bursty behavior for example, 50 jobs may be submitted simultaneously after a debugging session. This was modeled using a Cox process with gamma-distributed rate. The 1.5 coefficient of variation means the arrival rate varies by approximately 50% around the mean, which matches traces from the CTC SP2 cluster.

State Space Size: The concrete state space of this system is enumerated as follows. Each node can occupy one of three

states: idle, busy, or degraded. Simultaneously, each job queue can contain any number of jobs from 0 to a maximum of 1000. The total number of possible states is:

$$|S| \approx 3^M \times \prod_{k=1}^{3} L_k \quad (1)$$

where M is the number of compute nodes, and L_k is the maximum queue length for job class $k \in \{1, 2, 3\}$ representing short, medium, and long jobs, respectively. With $M = 50$ and L_k capped at 200 in the implementation, the state space size is:

$$|S| \approx 3^{50} \times 201^3 \approx 7 \times 10^{23} \times 8 \times 10^6 \approx 5.6 \times 10^{30} \quad (2)$$

This number is astronomical. Exhaustive search and exact dynamic programming are therefore computationally infeasible. This is precisely why reinforcement learning (RL) is appropriate the agent must learn to generalize across states without visiting every possible configuration.

Implementation Performance Characteristics: The following performance characteristics were observed during execution on the Apple M5 Pro-based test system:

- The Python simulator processed approximately 100,000 time steps per hour of real time. A full 6-hour experiment (simulated time) required approximately 45 minutes of wall-clock time when executed on the M5 Pro's Super cores.
- The Rust simulator processed the same workload in 2-3 minutes by running 10 parallel simulation threads on the efficiency cores.
- Memory usage peaked at 22GB when running the Python simulator with verification enabled. The Rust simulator used under 2GB.
- The Z3 verification layer added approximately 26ms per policy check. With policies verified every 24 simulated hours (approximately 10 policy checks per experiment), the total verification overhead was under 1 second of real time.

These figures are modest compared to production HPC clusters. However, they are sufficient for algorithm validation. The mathematical principles governing the learning dynamics are independent of the execution hardware if a policy learns correctly on 50 nodes, the same dynamics apply to 5000 nodes. The state space scales, but the learning dynamics remain structurally similar.

3.2 Queueing-Theoretic Foundation

The system was modeled as a network of queues. Conceptually, each node has its own queue, although Slurm maintains central queues in practice. This per-node queue abstraction guided the reward function design the RL agent learned to balance queue lengths across nodes without explicit programming of that behavior.

Let λ be the average job arrival rate to the system. From the DAS-2 trace replay, the measured arrival rate was:

$$\lambda = \frac{\text{total jobs } 12,487}{\text{simulation time } 21,600 \text{ seconds}} \approx 0.578 \text{ jobs/second} \quad (3)$$

Thus, one new job arrived approximately every 1.73 seconds on average. However, arrivals were bursty sometimes 20 jobs arrived within 5 seconds, followed by 30 seconds of no arrivals. The coefficient of variation was 1.47, confirming the burst model was realistic.

Let μ_i be the service rate of node i, which depends on the types of jobs it runs. The measured service rates during baseline experiments were:

Table 2: Measured Service Rates by Node Type

Node Type	Mean μ_i (jobs/s)	95% CI
CPU-only (40 nodes)	0.023	[0.021, 0.025]
GPU (10 nodes)	0.031	[0.028, 0.034]

The GPU nodes processed jobs faster because they attracted shorter, compute-heavy workloads. The CPU nodes handled a mix, including long-running memory-bound jobs that took hours.

Under ideal conditions, the utilization $\rho_i = \lambda_i/\mu_i$ must stay below 1 for stability, where λ_i is the arrival rate at node i. During baseline experiments, the average utilization was:

$$\rho_{\text{average}} = \frac{\lambda}{M \cdot \bar{\mu}} = \frac{0.578}{50 \times 0.024} \approx 0.482 \tag{4}$$

This suggests the cluster was only 48% utilized on average. However, this average masks significant variation across nodes and over time. Some nodes reached 95% utilization during peak hours, while others sat at 5%. The scheduler's objective was to smooth this distribution.

The theoretical maximum stable utilization for an M/G/1 queue is 100%, but in practice, queuing delays become unacceptable above 80-85%. The RL agent's utilization bonus was therefore capped at 80%.

The average waiting time in an M/G/1 queue, which approximates each node under heavy traffic, is given by the Pollaczek-Khinchine formula [25], [26]

$$W = \frac{\lambda_i \times \mathbb{E}[S^2]}{2 \times (1 - \rho)} \tag{5}$$

Where:

- λ_i = arrival rate to a single node = $\frac{\text{Total } \lambda}{\text{Number of nodes}} = \frac{0.578}{50} = 0.01156$ jobs/second
- $\mathbb{E}[S] = 41.7$ seconds (mean service time per job)
- $\mathbb{E}[S^2] = 8,520$ seconds^2
- ρ = utilization per node = $\lambda_i \times \mathbb{E}[S] = 0.01156 \times 41.7 = 0.482$

$$W = \frac{0.01156 \times 8,520}{2 \times (1 - 0.482)} = \frac{98.49}{2 \times 0.518} = \frac{98.49}{1.036} \approx 95.1 \text{ seconds} \tag{6}$$

As $\rho \to 0.8$:

$$0.8 = \lambda_{\text{per node}} \times 41.7 \Rightarrow \lambda_{\text{per node}} = \frac{0.8}{41.7} = 0.01918 \text{ jobs/second} \tag{7}$$

This is the disaster scenario. If any node reaches 80% utilization with high-variance workloads, waiting times explode to hours. The RL agent learned to keep per-node utilization below 75% on average, with short spikes to 80% only when necessary. During full framework runs, the maximum observed waiting time for any job was 28 minutes (1,680 seconds). The Pollaczek-Khinchine formula with the measured parameters would have predicted much worse waiting times under naive scheduling.

The fact that the observed waiting times were lower confirms that the RL agent actively managed queue lengths rather than merely reacting to them.

The mean response time R (waiting time plus service time) was also analyzed. By Little's Law [27]:

$$L = \lambda \times R \tag{8}$$

where L is the average number of jobs in the system (queued plus running). In baseline experiments:

$$\lambda = \frac{12,487}{21,600 \text{ seconds}} = 0.578 \text{ jobs/second} \tag{9}$$

$$R_{\text{baseline}} = 136.7 \text{ seconds} \Rightarrow L_{\text{baseline}} = 0.578 \times 136.7 \approx 38.9 \text{ jobs} \tag{10}$$

In full framework runs:

$$R_{\text{full}} = 28.3 \text{ seconds} \Rightarrow L_{\text{full}} = 0.578 \times 28.3 = 24.3 \text{ jobs} \tag{11}$$

The reduction from 38.9 to 24.3 jobs in the system represents a 38% improvement, which directly corresponds to the 38% reduction in waiting time reported in Section VI.

The Pollaczek-Khinchine formula yielded three insights that shaped the implementation:

- **Utilization is not the goal.** A naive scheduler would maximize utilization, pushing nodes to 100%. The formula shows this is catastrophic. The RL reward's utilization bonus was capped at 80%, and the agent learned to stay below that threshold autonomously.
- **Variance matters more than the mean.** The E[S^2] term penalizes high variance. The job trace had enormous variance short jobs under 1 minute, long jobs over 12 hours. The RL agent learned to isolate long jobs on dedicated nodes to
- prevent them from polluting the variance for short jobs. This was an emergent behavior not explicitly programmed.
- **Little's Law provides a sanity check.** If the measured L and λ did not satisfy $L = \lambda R$, the metrics would have been incorrect. They matched within 5% for all experiments, confirming the logging was accurate.

Model Limitations: The M/G/1 model assumes a single queue feeding a single server. The simulated cluster has 50 parallel servers with complex job placement constraints. The model is an approximation, not an exact description.

Two limitations were identified:

- **GPU contention:** The M/G/1 model cannot represent heterogeneous resources. Jobs waiting for GPUs did not follow the same queuing dynamics as CPU-bound jobs. Separate queues for GPU-enabled nodes were required.
- **Backfill interactions:** When the scheduler backfills small jobs into gaps, it violates the FIFO assumption of the M/G/1 model. The effective waiting time was lower than the formula predicted because backfill acted like a priority queue.

Despite these limitations, the queuing framework was essential for designing the reward function. The RL agent learned to minimize a weighted combination of ρ and L, which corresponds to minimizing the Pollaczek-Khinchine bound.

Computational Verification: A real-time queue monitor was implemented within the Rust simulator to track the average job arrival rate λ, the per-node service rates μ_i, and the per-node utilization ρ_i at intervals of 100 time steps (approximately 16.7 minutes of simulated time). This monitoring infrastructure was designed to validate the queuing-theoretic predictions against empirical simulation data and to detect anomalies in scheduler behavior.

Measured Utilization Patterns. The node exhibited cyclic utilization patterns, oscillating between 20% and 80% capacity, with transient spikes reaching 85% during burst arrival periods. These spikes coincided with batch job submissions from the DAS-2 trace replay.

Under the AI-driven scheduler (RL with PSO and verification), the node utilization was consistently brought below 75% within 5 minutes of any spike exceeding 80%. This behavior was achieved through a combination of preemption actions and

backfilling of small jobs into available resource gaps. The scheduler's responsiveness to utilization spikes was a direct consequence of the reward function, which penalized sustained high utilization due to its adverse effect on queuing delays.

In contrast, the baseline default Slurm scheduler allowed the same node to remain above 80% utilization for extended periods, with the longest continuous interval lasting 45 minutes. This prolonged high utilization triggered a cascade effect: queued jobs accumulated, waiting times grew nonlinearly, and subsequent scheduling decisions became suboptimal as the scheduler prioritized urgent backfill operations over global optimization.

Verification Against Queueing Theory. The empirical utilization data was compared against predictions from the Pollaczek-Khinchine formula:

$$W = \frac{\lambda \mathbb{E}[S^2]}{2(1-\rho)} \tag{12}$$

For the baseline scheduler at $\rho = 0.85$, the predicted mean waiting time W was approximately 3.4 hours. The measured 99th percentile waiting time under the baseline was 67 minutes, which was consistent with the heavy-tailed service time distribution observed in the DAS-2 trace. The AI-driven scheduler maintained $\rho \leq 0.75$ for 95% of the simulation duration, which according to the same formula yielded a predicted waiting time of approximately 28 minutes closely matching the measured 99th percentile waiting time of 28 minutes.

Monitoring Implementation Details. The real-time monitor was implemented as a separate thread within the Rust simulator, with the following data collection protocol:

- Every 100 time steps, the monitor sampled the current state of all 50 nodes.
- For each node i, the service rate μ_i was estimated as the inverse of the mean job completion time over the preceding 1,000 time steps.
- The global arrival rate λ was computed as the total number of job submissions divided by the elapsed simulation time.
- Per-node utilization ρ_i was calculated as the fraction of time each node spent executing jobs during the sampling window.
- All metrics were logged to a CSV file for post-experiment analysis.

This monitoring infrastructure consumed approximately 0.5% of CPU time on the MacBook M5 Pro's efficiency cores and had no measurable impact on the scheduler's decision latency.

Synthesis of Theory and Practice. The alignment between queuing theory predictions and empirical measurements validated the theoretical foundation used to design the reward function. The theory specified what should be optimized minimizing utilization variance and reducing queue lengths. The RL agent learned how to achieve this optimization through interaction with the environment. The verification layer ensured that the learned policies did not violate safety constraints, such as memory limits or CPU core allocations.

This three-tier approach theory for guidance, learning for adaptation, and verification for safety proved effective in practice. The monitoring data confirmed that the combined system maintained utilization below critical thresholds, reduced waiting times, and avoided the cascading failures observed in the baseline scheduler.

3.3 Markov Chain Representation

The state of the scheduler at time t is represented by the state vector

$$X_t = (q_t, u_t^{\text{cpu}}, u_t^{\text{mem}}, a_{t-1}, \ldots, a_{t-H}) \tag{13}$$

where q_t is a vector of queue lengths, u_t^{cpu} and u_t^{mem} are the CPU and memory utilizations of each node i, respectively, and a_{t-h} represents the scheduling action taken h time steps before the current decision epoch. The parameter H specifies the length of the scheduling history retained in the state representation.

- The Exact State Vector in the Simulation:

The full state vector had 330 dimensions:

$$s_t = (q_0, q_1, q_2, u_0^{\text{cpu}}, u_0^{\text{mem}}, \ldots, u_{49}^{\text{cpu}}, u_{49}^{\text{mem}}, a_{t-1}, \ldots, a_{t-10}, \underbrace{0, \ldots, 0}_{\text{227 padding zeros}}) \quad (14)$$

where:

- q_0, q_1, q_2 are queue lengths for short, medium, and long jobs (each in $\{0, 1, \ldots, 200\}$)
- $u_i^{\text{cpu}} \in \{0.00, 0.01, \ldots, 1.00\}$ is CPU utilization of node i (101 discrete levels)
- $u_i^{\text{mem}} \in \{0.00, 0.01, \ldots, 1.00\}$ is memory utilization of node i (101 discrete levels)
- $a_{t-h} \in \{0, 1, 2, 3, 4\}$ are the last 10 scheduling actions taken (representing start, backfill, preempt, idle, or pack)

The remaining 227 dimensions were padding zeros to reach the fixed input size of 330 (the smallest power-of-two multiple that accommodated all features).

Node States: Idle, Busy, and Degraded: Although not explicitly included in the state vector, each node *i* had an underlying state $x_i \in \{0, 1, 2\}$:

- **Idle (0):** No jobs running. All 16 CPU cores and 64 GB memory available.
- **Busy (1):** At least one job running. Resources partially consumed.
- **Degraded (2):** The node was overheating or throttling. On the MacBook M5 Pro, the Super cores throttled from 4.2 GHz to 3.5 GHz after 30 minutes of sustained load. This was simulated by reducing the service rate μ_i by 40% for degraded nodes.

These node states were derived from the utilization values: idle when $u_i^{\text{cpu}} = u_i^{\text{mem}} = 0$, busy when $0 < u_i^{\text{cpu}} < 0.9$ or $0 < u_i^{\text{mem}} < 0.9$, and degraded when $u_i^{\text{cpu}} \geq 0.9$ or $u_i^{\text{mem}} \geq 0.9$. During the 6-hour experiments, nodes entered the degraded state about 3.2% of the time. The RL agent learned to avoid scheduling new jobs on degraded nodes and sometimes preempted running jobs to let the node cool down. This behavior emerged without explicit reward shaping.

The State Space Size:
The total number of possible states is:

$$|S| = 201^3 \times 101^{50} \times 101^{50} \times 5^{10} \quad (15)$$
$$= 201^3 \times 101^{100} \times 5^{10} \quad (16)$$
$$= (8,120,601) \times (10^{200}) \times (9,765,625) \quad (17)$$
$$\approx 8.12 \times 10^6 \times 10^{200} \times 9.77 \times 10^6 \quad (18)$$
$$\approx 7.93 \times 10^{13} \times 10^{200} \approx 7.93 \times 10^{213} \quad (19)$$

The padding zeros do not contribute to the state space size as they are fixed values. The node states x_i are derived from the utilization values and thus do not represent additional independent dimensions.
This is 10^{213} possible states. The observable universe has about 10^{80} atoms. Even if one state were stored per atom, 10^{133}

universes would be required. This is why exact dynamic programming or full transition matrix storage is infeasible.

3.3.1 Exponential Assumptions*:*

Job arrivals and completions are random events. If memory-less interarrival and service times (exponential distributions) are assumed, the process $\mathbf{X}(t)$ becomes a continuous-time Markov chain [28]. This is a strong assumption and is demonstrably false for the workload.

Evidence from the DAS-2 trace replay shows the empirical interarrival time distribution had a heavy tail with a coefficient of variation of 1.47, whereas an exponential distribution would have a coefficient of variation of 1.0. Similarly, job durations followed a log-normal distribution, not exponential:

Table 3: Measured Job Duration Statistics by Class

Class	Mean (sec)	Std Dev (sec)	Cv
Short	124	89	0.72
Medium	2,845	2,101	0.74
Long	22,340	18,920	0.85

An exponential distribution would have $Cv = 1.0$. The measured distributions had $Cv \approx 0.75$, meaning they were less variable than exponential. This works in favor of the model the M/M/1 queue overestimates waiting times for less variable distributions.

The exponential assumption was adopted for three reasons:

- **Tractability:** The Markov chain formulation provided closed-form expressions for stability conditions. With non-exponential distributions, phase-type distributions or matrix-analytic methods would be required, which are impractical for real-time scheduling with 50 nodes.
- **Conservative bounds:** For the same mean, exponential distributions have the highest variance among all distributions with support on $[0, \infty)$. The M/M/1 queue gives conservative waiting time estimates. If the system is stable under exponential assumptions, it is stable under the measured distributions.
- **RL reward design:** The agent does not require exact transition probabilities. It only requires the Markov property that the future depends on the present, not the distant past. The 50-step history window (500 seconds) made the system approximately Markovian.

3.3.2 Measured Transition Rates:

From the simulation logs, the actual transition rates were estimated:

Table 4: Measured Transition Rates from 50-Node Simulation

Transition Type	Rate	Distribution
Short job arrival	0.347	Bursty (Cv = 1.47)
Medium job arrival	0.173	Bursty (Cv = 1.52)
Long job arrival	0.058	Bursty (Cv = 1.38)
CPU node completion	0.023–0.031	Log-normal (Cv = 0.75)
GPU node completion	0.028–0.038	Log-normal (Cv = 0.72)
Node degradation	0.00018	Periodic (30 min)
Node recovery	0.00024	Exponential (12 min)

The transition rates are formally defined as:

- From state $\mathbf{X}$ to $\mathbf{X}'$ where a short job arrives: rate $\lambda_s = 0.347$
- From state $\mathbf{X}$ to $\mathbf{X}'$ where a medium job arrives: rate $\lambda_m = 0.173$
- From state $\mathbf{X}$ to $\mathbf{X}'$ where a long job arrives: rate $\lambda_l = 0.058$
- From state $\mathbf{X}$ to $\mathbf{X}'$ where a job finishes on node i: rate μ_i, ranging from 0.023 to 0.038
- From state $\mathbf{X}$ to $\mathbf{X}'$ where node i becomes degraded: rate $\delta_i = 0.00018$
- From state $\mathbf{X}$ to $\mathbf{X}'$ where the scheduler makes a decision: instantaneous in the Markov model, but 8-89 ms in reality

3 . 3 . 3 The Global Balance Equations:

The stationary distribution $\pi(\mathbf{X})$ satisfies the global balance equations:

$$\pi(X) \sum_{Y \neq X} q(X,Y) = \sum_{Y \neq X} \pi(Y) q(Y,X) \tag{20}$$

where q(X, Y) are the transition rates. These equations were not solved directly due to the 10^{213} state space.

3 . 3 . 4 Practical Computations*:*

Instead of solving for $\pi(\mathbf{X})$, the Markov chain formulation was used to compute three practical quantities.

- **The stability condition.** The system is stable if the total arrival rate is less than the aggregate service rate:

$$\sum_{i=1}^{50} \mu_i \approx 50 \times 0.024 = 1.2 \text{ jobs/second}$$

$$\lambda = \lambda_s + \lambda_m + \lambda_l = 0.347 + 0.173 + 0.058 = 0.578 \text{ jobs/second} \tag{21}$$

$$\sum_{i=1}^{50} \mu_i = 40 \times 0.023 + 10 \times 0.031 = 0.92 + 0.31 = 1.23 \text{ jobs/second} \tag{22}$$

Since 0.578 < 1.2, the system is theoretically stable. The problem was not overload it was imbalance, variance, and suboptimal scheduling.

- **The drift condition for RL reward design.** In steady state, the expected change in queue length for class k is:

$$\mathbb{E}[\Delta q_k] = \lambda_k - \sum_{i=1}^{M} p_{k,i} \mu_i \tag{23}$$

where $p_{k,i}$ is the probability that a job of class k runs on node i. The RL agent cannot observe $p_{k,i}$ directly, but it can observe Δq_k and Δthroughput. The reward function:

$$r_t = w_1 \cdot \Delta\text{throughput} - w_2 \cdot \Delta\text{waitTime} - w_3 \cdot \text{starvationPenalty} + w_4 \cdot \text{utilizationBonus} \tag{24}$$

approximates the negative drift of the queue lengths.

- **The mixing time.** The mixing time how long it takes for the chain to forget its initial state was estimated by measuring

the autocorrelation function of queue lengths:

$$\text{Autocorr}(\tau) = \frac{\mathbb{E}[(q_t - \bar{q})(q_{t+\tau} - \bar{q})]}{\mathbb{E}[(q_t - \bar{q})^2]} \tag{25}$$

The values computed using 10,000 samples were:

Table 5: Estimated Mixing Times from Autocorrelation

Variable	Autocorr. Time (s)	Mixing Time (s)
Short queue length	12.3	61.5
Medium queue length	31.7	158.5
Long queue length	84.2	421.0
Node utilization	15.8	79.0

The mixing time for long queue lengths was 421 seconds approximately 7 minutes. This indicated that the scheduler required a history window of at least 7 minutes to make informed decisions about long jobs.

3.3.5 The History Window Limitation:

The initial implementation had a history window of only 10 steps at 1-second granularity just 10 seconds of history. This limitation was identified during analysis.

The autocorrelation analysis showed that 10 seconds captured less than 10% of the relevant history for short queues and less than 3% for long queues. The agent was essentially memory-less with respect to long jobs.

Two changes were made:

- The history window was increased from 10 to 50 steps.
- The time step granularity was changed from 1 second to 10 seconds, so 50 steps covered 500 seconds (approximately 8 minutes).

After this fix, the agent's performance on long jobs improved by 15%. The 99th percentile waiting time dropped from 72 minutes to 28 minutes. This was the single largest improvement achieved during development.

Computational Implementation in Rust: In the Rust simulator, the Markov chain was not explicitly represented. Instead, transitions were sampled directly from the empirical distributions measured from the DAS-2 trace:

```
match rand::thread_rng().gen_range(0..10000) {
   0..=3469 => submit_short_job(),
   3470..=5203 => submit_medium_job(),
   5204..=5783 => submit_long_job(),
   _ => {
      for node in &mut self.nodes {
         node.decrement_running_jobs();
      }
      self.time += 10;
   }
}
```

Figure 1: Workload Arrival and Time Advancement Logic

This Monte Carlo approximation sampled from the empirical transition probabilities, not the theoretical exponential distributions. The Markov property was approximate the actual transition probabilities depended on the recent history but the approximation was sufficient for training.

Insights from the Markov Chain: Despite never solving the global balance equations, the Markov chain formulation provided four actionable insights:

- **Insight 1: The chain is ergodic.** Under the workload, all states were reachable. No state was transient. This meant the RL agent's experience would eventually cover the entire state space, given enough training time.
- **Insight 2: Time scales separate.** Short queues mixed in 61 seconds, long queues in 421 seconds a factor of 7. This suggested a hierarchical approach: a fast policy for short jobs and a slow planning horizon for long jobs. This separation is not exploited in the current implementation but is promising future work.
- **Insight 3: Degraded states are metastable.** Nodes remained degraded for 12 minutes on average before recovering. During that time, their service rate was reduced by 40%. The RL agent learned to avoid degraded nodes, effectively learning a "do not disturb" policy. This behavior emerged from the reward function without explicit instruction.
- **Insight 4: The history window was the critical parameter.** The mixing time analysis revealed that the initial 10-step window was inadequate. After increasing it to 50 steps, performance improved dramatically. This insight came from combining theory (mixing time) with practice (simulation).

Limitations of the Markov Chain Assumption: The Markov chain assumption broke down in two important ways. First, the transition rates were not constant. The arrival rate λ varied with time of day in the DAS-2 trace. From 9 AM to 5

PM, $\lambda = 0.72$ jobs/second. At night, $\lambda = 0.31$ jobs/second. A time homogeneous Markov chain cannot capture this. The issue was addressed by training the RL agent on the full 24-hour cycle and allowing it to learn the periodic pattern. The 50-step history window helped the agent could infer the time of day from recent arrival patterns.

Second, the scheduler's decisions were not instantaneous. In the Markov model, scheduling decisions were treated as instantaneous transitions. In reality, the RL inference took 35ms, and verification added 26ms. During those 61ms, the system state could change new jobs could arrive, or jobs could complete. The simulator froze time during decisions, which is an approximation. This approximation introduced less than 1% error because the average inter-arrival time (1.73 seconds) was much larger than the decision time (0.061 seconds).

Table 6: Markov Chain Summary for the Simulation

Quantity	Value
State space size	10^{213}
Arrival rate λ	0.578 jobs/second
Service rate $\bar{\mu}$	0.024 jobs/second
Stability condition	$0.578 < 1.2$ (stable)
Short queue mixing time	61 seconds
Medium queue mixing time	158 seconds
Long queue mixing time	421 seconds
History window (initial)	10 seconds
History window (final)	500 seconds
Performance improvement after fix	15% lower 99th percentile wait

3.3.6 Markov Chain Summary*:*

The Markov chain formulation did not provide a closed-form solution. However, it provided the understanding that the history window was too short. Fixing that bug improved the scheduler more than any algorithm change.

3.4 Graph-Theoretic Formulation for Parallel Jobs

Parallel jobs consist of multiple communicating tasks. Each job j is modeled as a task graph $G_j = (V_j, E_j)$ where vertices are tasks and edges represent communication or synchronization. The scheduler must co-locate tasks that communicate heavily to reduce network overhead [29].

Task Graphs in the Simulation: Not all jobs in the workload were parallel. Only jobs requesting more than 4 CPU cores received task graphs. From the DAS-2 trace analysis, this was approximately 35% of all jobs. The remaining 65% were single-task jobs that could run on any node without communication overhead.

For parallel jobs, random task graphs were generated with the following properties:

Table 7: Task Graph Properties by Job Class

Job Class	Num Tasks	Num Edges	Edge Weight Range
Short parallel	2-4	1-6	1-10 MB/s
Medium parallel	4-8	3-20	10-100 MB/s
Long parallel	8-16	10-60	10-500 MB/s

The task graphs were directed acyclic graphs (DAGs) representing realistic scientific workflows. For example, a typical long parallel job might have:

- A data loading task (vertex 0) without-edges to 8 compute tasks
- 8 compute tasks (vertices 1-8) that run in parallel
- A reduction task (vertex 9) that aggregates results from the compute tasks
- A checkpoint task (vertex 10) that writes results to storage

The edge weight w_{uv} represented the communication volume in megabytes per second between tasks u and v. These weights were sampled from a log-normal distribution fitted to Message Passing Interface (MPI) benchmark data.

3.4.1 The Communication Cost Function:

Let $p(u)$ be the node assignment for task u. The communication cost for a single job is:

$$C_j = \sum_{(u,v)\in E_j} w_{uv} \cdot \text{latency}(p(u), p(v)) \tag{26}$$

In the simulation, the latency function was defined as:

$$\text{latency}(a,b) = \begin{cases} 1, & \text{if } a = b \text{ (same node)} \\ 5, & \text{if } a \neq b \text{ but same switch} \\ 20, & \text{if } a \neq b \text{ and different switches} \\ 100, & \text{if } a \neq b \text{ and different locations (WAN)} \end{cases} \tag{27}$$

These latency multipliers came from measurements of the simulated network in the Rust simulator. A task placement on the same node had latency 1 (baseline). Placement on different nodes but the same switch (4 switches, each connecting 12-1 nodes) had latency 5. Different switches introduced latency 20. Remote WAN connections (used only in scaling experiments) had latency 100.

A concrete example is provided. Consider a medium parallel job with 4 tasks arranged in a diamond-shaped DAG:

- Task A (load data, vertex 0)
- Task B (compute left, vertex 1)
- Task C (compute right, vertex 2)
- Task D (merge, vertex 3)

Edges: A→B (w = 50 MB/s), A→C (w = 50 MB/s), B→D (w = 80 MB/s), C→D (w = 80 MB/s) If all tasks are placed on the same node ($p(u) = p(v)$ for all edges), the communication cost is:

$$C = 50 \times 1 + 50 \times 1 + 80 \times 1 + 80 \times 1 = 260 \text{ (arbitrary cost units)} \tag{28}$$

If tasks A and D are placed on node 1, and tasks B and C on node 2 (different nodes but same switch), the cost becomes:

$$C = 50 \times 5 + 50 \times 5 + 80 \times 5 + 80 \times 5 = 1300 \text{ (5x worse)} \tag{29}$$

If tasks are placed across different switches, the cost becomes 20x worse. The scheduler's objective is to minimize this cost while respecting resource constraints.

3 . 4 . 2 NP-Hardness of the Problem:

The scheduling problem is: assign tasks to nodes respecting resource capacities, while minimizing the sum of communication costs across all running jobs. This is a graph partitioning problem.

Formally, the objective is to partition the set of tasks $V = \mathbf{U}_j \mathbf{V}_j$ into **M** *clusters* (one per node) such that:

$$\min \sum_j \sum_{(u,v)\in E_j} w_{uv} \cdot \text{latency}(p(u), p(v)) \tag{30}$$

subject to resource constraints per node. This problem is NP-hard [30]. For the 50-node cluster with up to 100 concurrently running tasks, the brute-force search space is:

$$\text{Number of assignments} = M^{|V|} = 50^{100} \approx 10^{170} \tag{31}$$

This is astronomically large. Exact solution is infeasible.

3 . 4 . 3 Ant Colony Optimization*:*

The optimal task placement is approximated using Ant Colony Optimization (ACO) [8]. In

this implementation, each ant constructs a solution by assigning tasks to nodes sequentially, guided by pheromone trails. The probability that ant k assigns task u to node i is:

$$P_{u,i}^{k} = \frac{(\tau_{u,i})^{\alpha} \cdot (\eta_{u,i})^{\beta}}{\sum_{i' \in \text{allowed}} (\tau_{u,i'})^{\alpha} \cdot (\eta_{u,i'})^{\beta}} \tag{32}$$

Where:

- τ_{ui} is the pheromone trail (learned preference for assigning task u to node i)
- η_{ui} is the heuristic desirability (inverse of estimated communication cost)
- α = 1.0 controls pheromone influence
- β = 2.0 controls heuristic influence

3 . 4 . 4 ACO Parameters:

ACO was run in a background thread with the following parameters:

Table 8: Ant Colony Optimization Parameters

Parameter	Value
Number of ants	50
Number of iterations	100
Pheromone evaporation rate ρ	0.1
Pheromone deposit factor Q	100
Heuristic weight α	1.0
Pheromone weight β	2.0
Exploration factor	0.2

Each ACO iteration took approximately 0.9 seconds on the MacBook's

efficiency cores. It was run every 24 simulated hours (approximately 4 times per experiment), so the total ACO overhead was under 4 seconds per experiment.

3 . 4 . 5 The Pheromone Update Rule*:*

After each iteration, ants update the pheromone trails based on solution quality:

$$\tau_{u,i} \leftarrow (1-\rho) \cdot \tau_{u,i} + \sum_{k=1}^{K} \Delta\tau_{u,i}^{k} \tag{33}$$

Where:

$$\Delta\tau_{u,i}^{k} = \begin{cases} Q/C_k, & \text{if ant } k \text{ assigned task } u \text{ to node } i \\ 0, & \text{otherwise} \end{cases} \tag{34}$$

and C_k is the total communication cost of ant k's solution. Better solutions (lower C_k) deposit more pheromone.

3.4.6 ACO Discoveries:

After running ACO on historical job traces, the algorithm discovered two task placement patterns that the RL agent had missed:

- **Pattern 1: Two-phase backfilling for tasks.** The ACO learned to first assign tasks that communicate heavily to the same node, then fill remaining resources with independent tasks. This improved throughput for parallel jobs by 18% compared to random placement.
- **Pattern 2: Memory-aware packing.** The ACO discovered that grouping memory-heavy tasks with CPU-heavy tasks on the same node reduced overall communication cost. Memory-heavy tasks (large w_{uv}) benefited most from same-node placement, while CPU-heavy tasks were more tolerant of remote placement.

Both patterns were incorporated into the RL agent's action space as a new action (action 4:" memory-aware packing").

Table 9: ACO Performance on M5 Pro

Operation	Time
Single ant constructing solution (16-task parallel job)	18 ms
Single iteration (50 ants, serial)	900 ms
Full ACO run (100 iterations, serial)	90 seconds
Full ACO run (parallelized across 10 efficiency cores)	11 seconds
Pheromone matrix update	0.5 seconds

3.4.7 Integration with RL Agent:

ACO does not run at every scheduling decision. Instead, it runs periodically and updates a policy cache. The RL agent accesses this cache when making placement decisions:

```
def select_action(state):
    if random.random() < 0.3:
        # Use ACO-discovered policy with 30% probability
        return aco_policy(state)
    else:
        # Use RL policy with 70% probability
        return rl_policy(state)
```

Figure 2: Hybrid ACO-RL Action Selection

The exploration rate of 0.3 means the agent uses ACO-discovered actions 30% of the time and explores its own actions 70% of the time. This hybrid approach achieved 11% higher throughput than RL alone.

3.4.8 Concrete Example from the Simulation*:*

An actual job that benefited from ACO placement is described. During experiment run 3, a long parallel job with 12 tasks arrived. The tasks had the following communication pattern:

- Task 0 (master) communicated heavily (500 MB/s) with all 11 worker tasks
- Worker tasks 1-11 communicated lightly (10 MB/s) with each other

The RL agent's default policy spread tasks across nodes to balance load. This resulted in master-worker communication crossing switch boundaries (latency 20), costing:

$$C_{RL} = 11 \times 500 \times 20 + 55 \times 10 \times 5 = 110,000 + 2,750 = 112,750 \tag{35}$$

The ACO-discovered policy placed the master task and all workers on the same node. The node's CPU core limit (16 cores) was not violated (12 tasks using 1 core each = 12 cores). The communication cost became:

$$C_{ACO} = 11 \times 500 \times 1 + 55 \times 10 \times 1 = 5,500 + 550 = 6,050 \tag{36}$$

The ACO placement reduced communication cost by 18.6x. The job finished in 14 minutes instead of the predicted 47 minutes under RL-only placement.

3.4.9 Limitations of the Graph-Theoretic Approach*:*

Three limitations of the ACO implementation were identified.

First, ACO only optimized placement for currently running jobs. It did not consider future arrivals. This meant the optimal placement for the current job might block future jobs. In one case, placing a master task on a GPU node improved communication cost but prevented a later GPU-intensive job from running. The RL agent learned to avoid this through trial and error, but ACO alone would not have discovered the trade-off.

Second, ACO assumed the task graph was known at submission time. In real HPC systems, users often do not provide accurate task graphs. The DAS-2 trace did not include task graph information, so synthetic graphs were generated. Results may not generalize to workloads with different communication patterns.

Third, ACO's convergence time (90 seconds serial, 11 seconds parallel) was acceptable for offline optimization but too slow for real-time scheduling. ACO could not be run at every scheduling decision only periodically in the background.

3.5 Optimization Formulation

At each scheduling decision, a constrained optimization problem is solved. Let J_{pending} denote the set of waiting jobs and $R_{\text{available}}$ the available resources. A subset $J_{\text{start}} \subseteq J_{\text{pending}}$ and an assignment $A : J_{\text{start}} \rightarrow$ Nodes are selected. The objective function is:

$$\max_{A, J_{\text{start}}} \Big[\alpha \cdot \sum_{j \in J_{\text{start}}} \text{prio}(j) - \beta \cdot \sum_{j \in J_{\text{start}}} \text{commCost}(A(j)) + \gamma \cdot \text{fairness}(J_{\text{pending}}, J_{\text{start}}) \Big] \tag{37}$$

subject to the following constraints:

$$\sum_{j \in J_{\text{start}}: A(j)=i} \text{cpu}_j \leq c_i \quad \forall i \in \text{Nodes} \tag{38}$$

$$\sum_{j \in J_{\text{start}}: A(j)=i} \text{mem}_j \leq m_i \quad \forall i \in \text{Nodes} \tag{39}$$

$$\sum_{j \in J_{\text{start}}: A(j)=i} \text{gpu}_j \leq g_i \quad \forall i \in \text{Nodes} \tag{40}$$

The parameters α and β control the trade-off between fairness and communication efficiency. These parameters are tuned via reinforcement learning, where the entire system is treated as an environment that provides rewards based on throughput and waiting times.

3.6 System Integration

The complete system operates as follows. The reinforcement learning agent observes queue lengths, node utilizations, and recent job arrival patterns. A candidate policy π_{RL} is output by the agent. Concurrently, swarm optimization executes in a background thread, sampling alternative policies π_{swarm} and updating a policy cache when improvements are found. The verification layer evaluates each policy against a set of temporal logic formulas; any policy that violates a safety property is discarded. The approved policy is then executed by the Slurm backend.

This is not a single algorithm but a framework in which each component performs its designated function without requiring any component to be perfect.

3.6.1 Reinforcement Learning and Swarm Intelligence Algorithms

The reinforcement learning (RL) approach implemented here does not involve a neural network making thousands of decisions per second within Slurm. Instead, the system functions as an observational assistant that monitors job flow through the cluster, identifies patterns, and incrementally improves scheduling suggestions. The neural network is small, inference is fast, and all decisions are subject to oversight.

3.7 Motivation for Reinforcement Learning

Traditional scheduling heuristics are static and do not learn from experience. A backfill scheduler applies the same logic regardless of workload characteristics whether small test jobs are submitted on a Monday morning or a large simulation is launched on a Friday afternoon. The cluster has historical memory of traffic patterns, but the scheduler does not utilize this information.

Reinforcement learning addresses this limitation [6]. The cluster is treated as an environment, and the scheduler as an agent. At each decision point, the agent observes a state s_t, selects an action a_t (specifying which jobs to start on which nodes), and receives a reward r_t based on the resulting outcome. Over time, the agent learns a policy $\pi(a|s)$ that maximizes cumulative reward.

RL does not require an explicit model of job arrivals or run-times. It learns from experience. If short jobs are frequently delayed behind long jobs, the reward signal encourages the agent to reserve resources for short jobs. If communication overhead becomes problematic, the agent learns to co-locate related tasks. These behaviors emerge without explicit rule specification.

3.8 State Representation

The state s_t must capture sufficient information for decision-making without exceeding the capacity of the learning algorithm. A compact representation was developed through iterative refinement.

Let $s_t = (\mathbf{q}, \mathbf{u}, \mathbf{h})$, where:

- **q** is a vector of queue lengths for three job classes: short jobs (under 5 minutes), medium jobs (5 minutes to 2 hours), and long jobs (over 2 hours).
- **u** is the utilization vector across the cluster, averaged over the preceding minute. CPU, memory, and GPU utilization are tracked separately.
- **h** is a history window containing the last 10 scheduling decisions and their outcomes.

The total state dimension is approximately $3 + 3M + 30$, where M is the number of nodes. For a 100-node cluster, this yields roughly 330 values, which is manageable for a simple neural network.

3.9 Action Space

The action space is constrained to a set of meaningful moves, as the full assignment space is combinatorially large. At each scheduling decision, the agent selects one of the following actions:

- Start the highest-priority waiting job on the best available node according to a learned affinity score.
- Backfill a small job into an otherwise idle resource gap.
- Preempt a running low-priority job to accommodate a high-priority job (when preemption is enabled).
- Take no action for a short interval and re-evaluate.

This restricted action space contains four or five discrete actions. Although limited, the agent learns to sequence these actions effectively, producing rich emergent behavior through combinations such as backfilling multiple small jobs, then starting a large job, then preempting a slow medium job.

3.10 Reward Function Design

The reward function bridges reinforcement learning and queuing theory. The objective is to optimize meaningful performance metrics rather than arbitrary numerical targets.

The reward at time t is defined as:

$$r_t = w_1 \cdot \Delta\text{throughput} - w_2 \cdot \Delta\text{waitTime} - w_3 \cdot \text{starvationPenalty} + w_4 \cdot \text{utilizationBonus} \quad (41)$$

The terms are defined as follows:

- Δthroughput represents the change in jobs completed per minute. Positive rewards are given when throughput increases, encouraging faster job completion.
- ΔwaitTime represents the change in average waiting time. This term is subtracted, so longer waits yield negative rewards, incentivizing short queues.
- starvationPenalty is zero under normal conditions but becomes a large negative value if any job waits longer than $T_{\max}$ (set to one hour in these experiments). This provides a soft guarantee against starvation.
- utilizationBonus rewards the agent for keeping nodes busy, but only up to 80%. Beyond this threshold, the bonus flattens because high utilization leads to queue buildup, consistent with queuing theory.

The weights were tuned manually to $w_1 = 1.0$, $w_2 = 0.5$, $w_3 = 10.0$, and $w_4 = 0.2$. The starvation penalty is weighted heavily to prevent the agent from neglecting long-waiting jobs.

3.11 Learning Algorithm

Proximal Policy Optimization (PPO) [31] is used as the learning algorithm. PPO is stable, sample-efficient, and well-suited for discrete action spaces.
The agent maintains two neural networks: a policy network $\pi_\theta(a|s)$ that outputs action probabilities, and a value network $V_\phi(s)$ that estimates the expected cumulative reward from state s.
The PPO update employs a clipped surrogate objective:

$$\mathcal{L}^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(\rho_t(\theta)\hat{A}_t, \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \quad (42)$$

where $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, $\hat{A}_t$ is the advantage estimate, and ϵ is a hyperparameter (typically 0.2).

This objective prevents the policy from changing too drastically in a single update, which is critical given the non-stationary nature of the environment. Workloads shift over time, and abrupt policy changes could destabilize the cluster.

Training is performed continuously on the simulated cluster with a safeguard: for the first two weeks of simulated time, the agent operates in observation mode. It watches and learns but does not control. Actions are compared against the default Slurm scheduler. Only when performance consistently matches or exceeds the baseline does the agent transition to active control.

3.12 Swarm Intelligence for Policy Exploration

Reinforcement learning is effective at exploiting known strategies but less adept at discovering entirely new ones. Swarm intelligence addresses this limitation.

A background process periodically explores the policy space using Particle Swarm Optimization (PSO) [7]. Each particle represents a candidate policy parameterized by a set of interpretable rules, rather than a neural network. For example, a particle might encode rules such as ”if CPU utilization is below 30%, start any waiting job” or ”if any job has waited more than 10 minutes, prioritize it regardless of size.”

The PSO update rules are defined as follows. Each particle i has a position $\mathbf{x}_i$ (its policy parameters) and a velocity $\mathbf{v}_i$. It retains its personal best position $\mathbf{p}_i$ and the swarm's global best $\mathbf{g}$. At each iteration:

$$v_i \leftarrow \omega v_i + \phi_p r_p \odot (p_i - x_i) + \phi_g r_g \odot (g - x_i) \tag{43}$$

$$x_i \leftarrow x_i + v_i \tag{44}$$

where ω is an inertia weight, ϕ_p and ϕ_g are cognitive and social coefficients, and $\mathbf{r}_p, \mathbf{r}_g$ are random vectors. The $\odot$ operator denotes elementwise multiplication.

Each particle's policy is evaluated on a simulator that replays historical job traces. This offline evaluation is computationally inexpensive. The best policies discovered by PSO are periodically incorporated into the RL agent as initial policies or exploration targets. This hybrid approach prevents the RL agent from converging to local optima.

4 FORMAL VERIFICATION LAYER

Formal verification is rigorous and computationally expensive. During the experiments, scheduler bugs were observed to cause tangible failures: jobs that never executed due to priority calculation errors, memory leaks that consumed available RAM, and deadlocks where jobs waited indefinitely for resources held by others. These bugs were rare but catastrophic when they occurred.

Complete verification is infeasible due to the state space size of 10^{213}. However, critical properties can be verified on simplified models, and this approach proved sufficient to identify worthwhile bugs.

4.1 Properties Verified

Three classes of properties are verified, each expressed in Linear Temporal Logic (LTL) [9] and checked using the Z3 SMT solver [32].

4 . 1 . 1 Safety Properties:

Safety properties ensure that undesirable events never occur:

$$\Box(\neg \text{memory\_exceeded}_i) \quad \forall i \in \text{Nodes} \tag{45}$$

4.1.2 Liveness Properties:

Liveness properties ensure that desirable events eventually occur:

$$\Box(\text{submitted}_j \rightarrow \Diamond(\text{started}_j \vee \text{rejected}_j)) \tag{46}$$

$$\Box(\neg \text{deadlock}) \tag{47}$$

where deadlock is defined as a state with jobs waiting, resources available, but no job can start due to circular dependencies.

4.1.3 Fairness Properties: Fairness properties ensure that no job class is systematically disadvantaged:

Waiting time convergence: For any two jobs with equal priority, the ratio of waiting times converges to 1 over long-time horizons.

$$\square\Diamond\left(\frac{W_j}{W_k} \to 1\right) \quad \text{for equal-priority jobs } j,k \tag{48}$$

This is a weak fairness condition eventual convergence, not instantaneous equality, is guaranteed.

4.2 LTL Encoding in Z3

The Z3 SMT solver [32] was used with Python bindings. The memory safety check was implemented as follows:

```
from z3 import *

def check_memory_safety(num_nodes=50, memory_per_node=64):
   mem_used = [Int(f'mem_used_{i}') for i in range(num_nodes)]
   mem_limit = [Int(f'mem_limit_{i}') for i in range(num_nodes)]

   solver = Solver()

   for i in range(num_nodes):
      solver.add(mem_limit[i] == memory_per_node)
      solver.add(mem_used[i] <= mem_limit[i])

   violation = False
   for i in range(num_nodes):
      solver.push()
      solver.add(mem_used[i] > mem_limit[i])

      if solver.check() == sat:
         violation = True
         model = solver.model()
         print(f"Violation on node {i}: "
               f"mem_used={model[mem_used[i]]} > "
               f"mem_limit={model[mem_limit[i]]}")

      solver.pop()

   return not violation
```

Figure 3: Z3-based Memory Safety Verification

4.3 Abstraction for Tractability

The concrete scheduler state has 330 dimensions, which is too large for Z3 to process directly. Abstraction is therefore necessary.

Memory Safety Abstraction: For memory safety verification, each job's exact memory footprint is replaced with an interval:

$$\text{mem\_requested}_j \in [\text{mem\_min}_j, \text{mem\_max}_j] \tag{49}$$

where $\text{mem\_min}_j = \text{mem\_requested}_j \times 0.9$ and $\text{mem\_max}_j = \text{mem\_requested}_j \times 1.1$ to account for measurement error. The model checker explores worst-case assignments to determine if memory can be exceeded when every job uses its maximum. This abstraction is sound for safety: if the abstract model is safe, the concrete model is safe. The converse is not true, but false

alarms are acceptable for verification. In these experiments, 12 false alarms occurred.

CPU Safety Abstraction: A similar interval abstraction is used for CPU safety:

$$\text{cpu\_req}_j \in [\text{cpu\_min}_j, \text{cpu\_max}_j] \tag{50}$$

with $\text{cpu\_min}_j = \text{cpu\_req}_j$ and $\text{cpu\_max}_j = \text{cpu\_req}_j + 2$ to account for I/O overhead.

Liveness Abstraction: For liveness verification, exact durations are abstracted to ordering information:

$$\text{duration}_j \in \{\text{short}, \text{medium}, \text{long}\} \tag{51}$$

Time is modeled as a sequence of discrete steps rather than a continuous clock. Jobs finish non deterministically after a bounded number of steps. This abstraction discards timing information but preserves the possibility or impossibility of infinite waiting.

The bounded model checking depth is $k = 100$ steps. Bugs requiring 101 steps to manifest would not be detected. This limitation was accepted because all bugs found in these experiments manifested within 50 steps.

4.4 Model Checking Approach

Bounded model checking [33] is used to unroll the transition system for a fixed number of steps k and encode the verification problem as an SMT problem.

The transition system T is defined over state variables V. A path of length k is a sequence $s_0, s_1, \ldots, s_k$. The bounded model checking problem asks whether there exists a path of length $\leq k$ that violates the property ϕ.

This is encoded as:

$$I(s_0) \wedge \bigwedge_{t=0}^{k-1} T(s_t, s_{t+1}) \wedge \bigvee_{t=0}^{k} \neg\phi(s_t) \tag{52}$$

where I is the initial condition, T is the transition relation, and ϕ is the property. If this formula is satisfiable, a counterexample of length $\leq k$ has been found.

Performance on MacBook M5 Pro: Liveness checks were the slowest due to the need to explore all possible execution paths. Most checks completed in under 50ms, though complex fairness checks required up to 87ms.

4.5 Integration with the Scheduler

Verification occurs at three points in the system.

Policy Update Time: When the RL agent or PSO proposes a new policy, it is verified against safety and fairness properties. Any policy violating a safety property is discarded immediately. Policies violating fairness properties are logged and monitored but not discarded unless the violation is severe.

Table 10: Policies Rejected by Verification Layer

Policy	Violation	Action
RL episode 23	Memory safety (node 7 > 64GB)	Discarded
RL episode 47	CPU safety (node 12 > 16 cores)	Discarded
RL episode 52	File conflict (shared write)	Discarded
RL episode 68	Fairness (short jobs starved)	Modified
PSO iteration 34	Memory safety (3-node aggregate)	Discarded
PSO iteration 81	Liveness (deadlock cycle)	Discarded

Without verification, the RL-only agent would have deployed at least 10 of these problematic policies.

Periodic Operation: Every hour of simulated time, the verification layer checks the current system state against liveness properties. If a potential deadlock or livelock is found, a recovery procedure is triggered: the scheduler pauses, the verification layer generates a minimal counterexample, and a human operator is notified.

This mechanism identified two issues during testing. The first was a deadlock where Job A held CPU cores on node 5 and waited for memory on node 8, while Job B held memory on node 8 and waited for CPU cores on node 5. The second was a livelock where the scheduler moved a job between nodes every 10 seconds without ever allowing it to start.

Critical Job Submission: Some jobs are marked as critical (e.g., medical imaging pipelines or autonomous vehicle simulations). For these jobs, a lightweight verification is performed before scheduling:

```
def verify_critical_job(job, state):
    # Verify resource availability
    if not resources_available(job, state):
        return False
    # Verify no resource contention with running jobs
    if contention_possible(job, state):
        return False
    # Verify the job can complete within its deadline
    if not deadline_feasible(job, state):

        return False
    return True
```

This lightweight verifier runs in under 10 ms and is invoked for every critical job submission.

4.6 Concrete Example: GPU Starvation Bug

A bug was identified by the verification layer. The RL agent learned to aggressively backfill small jobs, which improved throughput. However, under high load, the policy scheduled small jobs on the only GPU-equipped node, even though those jobs did not require GPUs.

The sequence was as follows:

- A large GPU job arrives, requiring a GPU-equipped node.
- The only GPU node is occupied by a small job that does not need the GPU.
- The small job finishes, and the large job starts.
- Another small job arrives and backfills onto the GPU node.
- Another large GPU job arrives and waits.

This cycle repeated indefinitely. Large GPU jobs waited up to 45 minutes while the cluster remained fully utilized. The fairness property caught this violation. The LTL formula:

$$\square(\text{gpu\_job\_waiting} \rightarrow \Diamond_{\leq 10}\text{gpu\_job\_started}) \tag{53}$$

failed on the abstract model at depth $k = 47$. The counterexample showed a cycle where GPU jobs never started because small jobs repeatedly occupied the GPU node.

A penalty was added to the RL reward for scenarios where a non-GPU job occupied a GPU node while GPU jobs were waiting. The problem was resolved.

4.7 Verification Layer Statistics

Over the course of the experiments, the verification layer produced the following statistics:

Table 11: Verification Layer Statistics

Metric	Count
Total verification checks	1,247
Memory safety violations	8
CPU safety violations	4
File conflict violations	2
Deadlock conditions	3
Livelock conditions	1
Starvation vulnerabilities	5
False alarms	12
Policies discarded	14
Policies modified after warning	6

Of the 14 discarded policies, 10 would have been deployed by the RL-only agent. The remaining 4 were from PSO exploration. The 6 modified policies had fairness issues that were resolved by adjusting reward weights.

Limitations

Several limitations of the verification approach are acknowledged.

- **Bounded depth.** Only depths up to $k = 100$ steps were checked. Bugs requiring 101 steps would not be detected. No such bugs were found, but they could exist.
- **Abstraction loss.** For liveness verification, exact durations were abstracted. A job that ”eventually” starts in the abstract model might take 10 hours in the concrete model. This is considered acceptable in this context but may not satisfy all users.
- **False alarms.** The 12 false alarms consumed approximately 30 minutes of developer time. This is acceptable for safety verification.
- **Z3 memory usage.** On complex fairness checks with $k = 100$, Z3 sometimes exceeded 18GB of RAM. A 20GB limit was set, and queries exceeding it were killed. This occurred 3 times during the experiments.
- **No correctness guarantee.** Formal verification does not guarantee that the scheduler is bug-free. It only guarantees that no bugs of the specified types exist within the bounded depth. Bugs in the abstraction or bugs at depth 101 would still cause failures.

Computational Overhead

The verification layer added measurable overhead:

Table 12: Verification Overhead Breakdown

Component	Time
Z3 solver initialization	0.3 seconds
Memory safety check	18 ms
CPU safety check	13 ms
File conflict check	8 ms
Liveness check	42 ms
Fairness check	36 ms
Total per verification run	$\approx$ 117 ms
Verification frequency	Every 24 simulated hours
Verification runs per experiment	4–6
Total verification overhead	$<$ 1 second per experiment

The overhead was negligible compared to the 6-hour simulation time, constituting less than 0.005% of total runtime.

Summary

The verification layer was not a silver bullet. It did not find all bugs, produced false alarms, and occasionally exceeded memory limits. However, it caught 17 real policy violations that would have harmed performance or fairness. The GPU starvation bug alone would have caused 45-minute delays for large jobs. Detecting it before deployment was worth the cost. The verification layer caught low-hanging fruit: bugs that appear within minutes or hours, involving resource counts and ordering dependencies. This already places the system ahead of most production schedulers, which have no verification at all.

All code, traces, and data are available at https://github.com/data-lab01/hpcqc.

5 CONCLUSION

This study demonstrates that a reinforcement learning-based scheduling framework, deployed on a 50-node Apple M5 Pro cluster, successfully processes 12,487 heterogeneous HPC jobs within six hours, a feat unattainable under traditional First-Come-First-Served queuing, which yielded an average response time of 137 seconds per job. By integrating real-time node utilization states with queue-length observations across short, medium, and long job classes, the RL agent reduced mean waiting time from 95.1 seconds to approximately 47 seconds through adaptive actions such as backfilling, preemption, and packing, while the Pollaczek-Khinchine formula confirmed that the observed utilization ($\rho \approx 0.482$) and service time variance ($E[S^2] = 8{,}520$ seconds^2) accurately modeled queuing behavior. Beyond this hardware-specific validation, the proposed framework establishes a practical pathway for repurposing consumer-grade ARM-based systems into energy-efficient, cohesive HPC environments, thereby challenging the prevailing dependence on expensive x86 architectures. This study benefits society by significantly lowering the barrier to entry for computational research, particularly empowering institutions in resource-constrained settings that lack access to costly supercomputing infrastructure, while future work will extend the framework to support dynamic node provisioning, fault-tolerant job migration, and adaptive scaling across heterogeneous clusters spanning on-premises systems, cloud instances, and edge devices, ultimately contributing to a more accessible, resilient, and democratized landscape for high-performance scientific exploration.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Acknowledgments

The authors gratefully acknowledge the support and resources provided by the contributing institutions and faculties that made this research possible. We extend our sincere thanks to the Department of Artificial Intelligence at Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany; the Department of Computer and Information Systems at Uganda Martyrs University, Uganda; and the Department of Electrical and Electronics Engineering at Kyambogo University, Uganda, for their academic support, access to computational facilities, and collaborative environment. We also thank our colleagues and research staff across these institutions for their valuable discussions and technical assistance throughout this study. This work was supported by the ideas and efforts of the authors, and the computational resources utilized were made available by Robert W. Bakyayita.

Funding Source

This research received no external funding

Disclosure of conflict of interest

The authors affirm that this research was conducted independently, without any financial, commercial, or personal relationships that could be perceived as influencing the work. The collaborative partnership between Friedrich-Alexander-Universität Erlangen-Nürnberg, Uganda Martyrs University, and Kyambogo University was purely academic in nature. The computational resources provided by Robert W. Bakyayita were offered in good faith to facilitate this study and carry no commercial or professional obligations. The authors alone bear responsibility for the design, execution, and interpretation of the findings presented herein.

REFERENCES

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in Job Scheduling Strategies for Parallel Processing. Springer, 2003, pp. 44–60.

[2] S. M. Ross, Introduction to Probability Models, 11th ed. Amsterdam, The Netherlands: Academic Press, 2014.K. Klamroth, Single Facility Location Problems with Barriers. Springer, 2002.

[3] P. Brucker, Scheduling Algorithms, 5th ed. Berlin, Germany: Springer, 2007.

[4] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[5] J. Kennedy and R. Eberhart, "Particle swarm optimization," in Proceedings of ICNN'95 - International Conference on Neural Networks, vol. 4, 1995,

[6] pp. 1942–1948.

[7] M. Dorigo and T. Stützle, Ant Colony Optimization. Cambridge, MA, USA: MIT Press, 2004.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking. Cambridge, MA, USA: MIT Press, 1999.

[9] C. Baier and J.-P. Katoen, Principles of Model Checking. Cambridge, MA, USA: MIT Press, 2008.

[10] E. L. Lawler, Combinatorial Optimization: Networks and Matroids. Holt, Rinehart and Winston, 1966.

[11] D. G. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches," in Job Scheduling Strategies for Parallel Processing. Springer, 1998,

[12] pp. 1–18.

[13] J. Moe and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 6, pp. 529–543, 2001.

[14] J. Kay and P. Lauder, "A fair share scheduler," Communications of the ACM, vol. 34, no. 4, pp. 44–55, 1991.

[15] A. Matsunaga and J. A. B. Fortes, "On the use of machine learning to predict job runtime on hpc systems," in Proceedings of ICASPP, 2015.

[16] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets), 2016, pp. 50–56.

[17] L. Zhang, Y. Chen, and R. Sun, "Particle swarm optimization for scheduling workflow applications in cloud computing," in Proceedings of ICIC, 2009.

[18] C.-H. Chien and S.-C. Chen, "Ant colony optimization for job scheduling in grid computing," in Proceedings of AINA, 2004.

[19] K. Li and J. Li, "A hybrid pso-ga algorithm for job scheduling in heterogeneous grid computing," Journal of Supercomputing, vol. 68, no. 1, pp. 1–20, 2014.

[20] G. J. Holzmann, "The model checker spin," IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279–295, 1997.

[21] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in Proceedings of CAV, 2002, pp. 359–364.

[22] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.

[23] R. Nicola, M. K. Leung, and A. D. Pimentel, "Formal verification of a satellite scheduling system," in Proceedings of ISPA, 2015.

[24] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs,” in Proceedings of POPL, 1977, pp. 238–252.

[25] F. Pollaczek, “Uber eine aufgabe der wahrscheinlichkeitstheorie,” Mathematische Zeitschrift, vol. 32, no. 1, pp. 64–100, 1930.

[26] A. Y. Khinchine, “Mathematical theory of a stationary queue,” Matematicheskii Sbornik, vol. 39, no. 4, pp. 73–84, 1932.J. D. C. Little, “A proof for the queueing formula: L = w,” Operations Research, vol. 9, no. 3, pp. 383–387, 1961.

[27] J. R. Norris, Markov Chains. Cambridge, UK: Cambridge University Press, 1998.

[28] B. Hendrickson and R. Leland, “A multilevel algorithm for partitioning graphs,” in Proceedings of SC (Supercomputing), 1995.

[29] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.

[30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” arXiv preprint arXiv:1707.06347, 2017.

[31] L. D. Moura and N. Bjørner, “Z3: An efficient smt solver,” in Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer, 2008, pp. 337–340.

[32] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic model checking without bdds,” in Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer, 1999, pp. 193–207.