



Efficient Job Scheduling Using Reinforcement Learning Approach in Cluster Computing

Inam Ullah*, Abdul Haseeb Malik, Qazi Ejaz Ali, Waheed Ur Rehman, Syed Muhammad Asim Jalal, Najeebullah

Department of Computer Science, University of Peshawar, Peshawar, Pakistan

ARTICLE INFO

Article History:

Received:	May	28, 2026
Revised:	June	16, 2026
Accepted:	June	22, 2026
Available Online:	June	30, 2026

Keywords:

Job Scheduling
Reinforcement Learning
Machine Learning
GPU Cluster Computing
Proximal Policy Optimization
XGBoost
HPC

Classification Codes:

Funding:

This research received no specific grant from any funding agency in the public or not-for-profit sector.

ABSTRACT

Job scheduling in high-performance computing (HPC) clusters directly affects user waiting times and GPU hardware utilization. Conventional heuristic schedulers such as First Come First Serve (FCFS), Shortest Job First (SJF), and backfilling depend on static rules that cannot adapt to dynamic workloads. While machine learning (ML) prediction models and reinforcement learning (RL) schedulers have each shown capability, neither fully resolves the problem: prediction models cannot respond to real-time cluster state changes, while RL agents explore vast action spaces without prior experience. This paper offers a hybrid ML+RL scheduling framework for GPU cluster computing. An XGBoost regression model is trained offline on real job execution records from the MIT SuperCloud dataset, incorporating NVIDIA DCGM GPU hardware data logging. The accomplished predictor is implanted as a live per-step component inside a custom Gymnasium simulation environment, generating node-specific runtime predictions at each scheduling decision. A Proximal Policy Optimization (PPO) agent receives the full live state of all 225 cluster nodes together with top 4 predicted priority nodes, allowing the agent to balance ML supervision with real-time cluster awareness. Evaluated on 11,001 held out test jobs, the proposed scheduler reduces average waiting time by 98.9% (from 121.5 hours to 1.3 hours), decreases makespan by 84.6% (from 3,453 hours to 533 hours), and improves GPU utilization from 7.24% to 46.90%, outperforming FCFS, backfilling and an SJF oracle with perfect runtime knowledge in a trace driven simulation. Because utilization is measured over each scheduler's own makespan, the utilization gain mirrors the makespan gain. These results come from a single training seed, and the separate contribution of the ML hints is analysed from the training reward but not yet isolated by ablation; the paper states these limits explicitly.



© 2026 The authors published by JCIS. This is an Open Access Article under the Creative Common Attribution Non-Commercial 4.0

Corresponding Author's Email: inamkhattak1999@uop.edu.pk

1. Introduction

The growing proliferation of data-intensive applications such as deep learning, large-scale analytics, and scientific simulation has made high-performance computing (HPC) and cluster computing systems indispensable infrastructure. These systems run thousands of concurrent tasks that require shared memory, CPUs, and GPUs. The main factor controlling the distribution of resources is the scheduler, which chooses which jobs execute, when, and on which node. These choices significantly impact cluster performance in terms of resource usage, throughput, and job waiting times [1, 2].

Due to their ease of use and minimal computational overhead, traditional scheduling methods, including First Come, First Served (FCFS), Shortest Job First (SJF), and backfilling, are widely used. Nevertheless, these techniques are controlled by static rules that are unable to take advantage of trends from past execution data or adjust to changing

workloads. Such strict rule-based methods result in low utilization and long wait times in heterogeneous GPU clusters, where jobs span widely disparate resource requirements and nodes differ in GPU capacity [1, 3].

Runtime prediction models have been used to apply machine learning (ML) to HPC scheduling. A model can help make more intelligent scheduling decisions if it can understand the relationship between job properties, node characteristics, and actual execution durations [4, 5]. Nevertheless, a prediction model by itself is not a scheduler; it cannot adjust to the current cluster state during scheduling and does not directly improve scheduling objectives. By teaching an agent to learn scheduling choices through interaction with the cluster environment, reinforcement learning (RL) overcomes this constraint [2, 6]. Despite their flexibility, pure RL schedulers have poor early performance and slow convergence since they begin exploration from scratch without any past knowledge from previous cluster logs.

In contrast to previous hybrid approaches that loosely couple the two, the proposed design calls the predictor at every scheduling step and embeds its output directly in the PPO agent's observation, allowing the agent to combine ML guidance with real-time cluster awareness for better scheduling decisions. This paper proposes a hybrid ML+RL scheduling framework that tightly integrates an XGBoost runtime predictor as a live per-step component of the RL scheduling environment.

This paper makes three contributions. First, it describes a scheduling environment in which an XGBoost runtime predictor, trained only on the earliest 70% of jobs, is queried for every node at every decision, and its four best suggestions enter the PPO observation. Second, it compares the resulting policy with FCFS, backfilling and an SJF oracle on 11,001 later jobs replayed from the MIT SuperCloud trace, under one fully specified simulation and evaluation protocol. Third, it analyses which parts of the observed gain the present experiments can and cannot attribute to the ML hints, so that the results are read at their proper strength.

2. Literature Review

FCFS assigns jobs in arrival order, ensuring fairness but allowing large jobs to block shorter ones, leading to significant resource wastage. SJF prioritises shorter jobs to minimise average waiting time, but requires accurate runtime estimates that users rarely provide, and is susceptible to starvation of long jobs. Backfilling improves resource efficiency by allowing shorter jobs to advance in the queue provided they complete before blocking longer jobs, but its effectiveness depends on workload characteristics and runtime estimation accuracy [3]. All three methods are fundamentally static: they do not learn from past workloads and cannot adapt to real-time cluster conditions.

Gradient boosting methods have been most widely applied for HPC job runtime prediction due to their effectiveness with mixed-type and skewed data. Hou et al. demonstrated that broad learning and gradient boosting-style regression can significantly improve runtime prediction accuracy on HPC clusters, resulting in better scheduling decisions [5]. Deep learning approaches have also been applied: Bao et al. proposed deep learning-based placement for distributed ML clusters with heterogeneous workloads [4], and Peng et al. developed DL2 for runtime-aware queue management in deep learning clusters [1]. A consistent limitation across ML approaches is that they operate offline and cannot adapt recommendations to the live cluster state at scheduling time.

RLScheduler by Zhang et al. was an early demonstration that RL agents can learn competitive HPC batch scheduling policies outperforming FCFS and backfilling on real workload data [2]. DRAS by Fan et al. scaled deep RL to large heterogeneous HPC environments [6]. Mondal et al. demonstrated that RL agents adapt to time-varying workloads where fixed schedulers fail [7]. Multi-agent approaches have been explored for scalability [8], and recent work has addressed topology-aware scheduling [9], network contention [10], and co-scheduling across workload types [11]. Despite these advances, pure RL schedulers begin exploration without leveraging rich historical execution data, leading to slow convergence and costly early poor decisions.

Hybrid frameworks integrating ML prediction with RL decision-making have shown improved results over either approach alone. Hu et al. proposed interpretable ML-guided RL schedulers for data processing clusters [12], and Luo et al. developed prediction-assisted online scheduling for GPU clusters [13]. However, most hybrid approaches maintain loose coupling: the ML model operates independently and provides limited input, rather than being called at every scheduling step. The key research gap is the absence of frameworks that embed a dedicated telemetry-aware runtime predictor as a live per-step observation component of the RL environment, validated on real large-scale GPU cluster production workloads.

3. Framework and Methodology

The ML runtime predictor and the RL scheduling agent function as a single cohesive system in Figure 1, which depicts the suggested framework. The framework overcomes a basic drawback of current methods, such that RL agents can view live conditions but begin learning from scratch with no past information, whereas ML prediction models are aware of which nodes are historically fast but are unable to see the present cluster state. The ML component offers well-informed guidance from the first training step, and the RL component adjusts that guidance to current conditions at each scheduling choice, therefore simultaneously resolving both shortcomings.

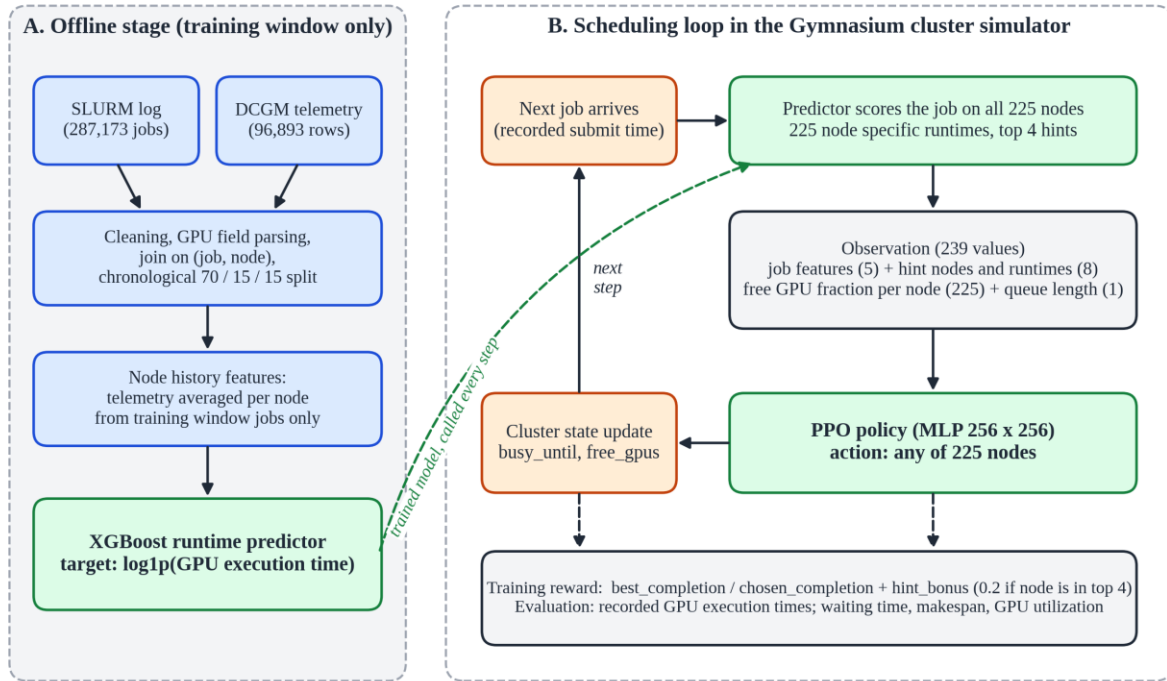


Figure 1: Proposed hybrid ML+RL scheduling framework. (A) Offline stage using the training window only. (B) Scheduling loop: at each decision the predictor scores the arriving job on all 225 nodes, the top four hints and the live cluster state form the PPO observation, and the chosen node updates busy_until and free_gpus. The reward is used only in training; evaluation uses recorded execution times.

3.1 Dataset

The MIT SuperCloud workload dataset, sourced from an operational GPU cluster managed by the Massachusetts Institute of Technology, is utilised [14]. The dataset consists of two complementary files. The scheduler log comprises 287,173 job records documented by the SLURM workload manager, detailing resource requests (CPU cores, RAM, GPU quantity and type), submission and execution timestamps, allocated nodes, and job priority. The Data Centre GPU Manager (DCGM) telemetry log comprises 96,893 entries of GPU hardware sensor data recorded by NVIDIA's Data Centre GPU Manager during job execution, encompassing GPU memory utilisation, streaming multiprocessor (SM) utilisation, PCIe receive and transmit bandwidth, power consumption in watts, total energy consumed in joules, and actual GPU execution time in seconds. The cluster comprises 225 nodes outfitted with NVIDIA Volta GPUs.

The two files are connected by a common node name and job identifier. The dataset is very useful for this work because of this combination, since most HPC datasets merely document what a job requested and how long it ran. In addition, the DCGM telemetry documents the hardware's real experiences during execution, allowing the predictor to learn node-specific behaviour instead of just job-level patterns.

3.2 Data Preprocessing and Feature Engineering

Before training, raw cluster logs must be thoroughly cleaned. Fourteen scheduling-relevant columns remain after the removal of thirteen administrative columns (array job metadata, exit codes, SLURM accounting strings, and flag fields). Only legitimate finished GPU runs, such as those with a non-empty nodelist, a non-null GPU request field, and a positive computed runtime ($\text{time_end} - \text{time_start}$), are kept in the filter. 201,922 valid scheduler records are produced by this filtering. There are two SLURM forms for the GPU request field: `gpu:volta:N` (GPU type and count) and `gpu:N` (count only). Both are correctly parsed, as 39% of GPU jobs would be silently discarded if the second format were ignored.

DCGM records are inner-joined with the scheduler data after being aggregated using column-wise mean from one row per GPU to one row per (task, node) pair. Each training sample is guaranteed to contain both job-level and hardware-level data due to the inner join. 73,335 combined records from 225 nodes are the end result. The dataset median was substituted for the SLURM memory sentinel value (maximum uint64, indicating no memory limit specified), which is present in about 39% of rows.

A feature set divided into four groups is the result of feature engineering. Resource requests (CPU cores, RAM, GPU count, priority, waiting time, GPU type, job type) are recorded by job-level features. GPU capacity per node is captured by node-level characteristics, which are empirically obtained from the data. Energy consumption, SM utilisation, PCIe bandwidth in both directions, power consumption, and GPU memory usage are all captured by GPU telemetry features, which are averaged per node from past runs. Workload intensity relationships that cannot be expressed by raw request numbers are captured by engineered ratio features (`cpu_mem_ratio`, `gpu_cpu_ratio`, `energy_per_gpu`, PCIe bandwidth ranges, power range, and SM utilisation range).

The regression goal, which is the actual GPU execution time from DCGM, covers seven orders of magnitude, ranging from 0.19 seconds to 2,895,390 seconds. A few extreme outlier tasks would dominate the loss function in a direct regression on this range, effectively eliminating short jobs. To ensure that the model learns equally from short and lengthy jobs, this range is compressed to [0.174, 14.879] using a $\log_{1p}$ transformation.

Availability of features at decision time

Because the predictor is queried inside the scheduler, every input must be known at the moment a job is placed. Table 1 lists each feature group, where it comes from and how its value is obtained during scheduling. Job request fields are fixed at submission. Waiting time enters the predictor as the time the job has already spent in the queue at the decision instant, which the simulator knows exactly. Node capacity comes from the static node inventory. Node telemetry features are per node means computed over jobs in the training window only; they are frozen before any validation or test job is processed and are never updated with telemetry from a job that is being scheduled. The regression target is never an input [20].

Feature group	Examples	Source	Value used at a scheduling decision
Job request	CPU cores, RAM, GPU count, GPU type, priority, job type	SLURM submission record	Known at submission
Queue time	waiting time	Simulator clock	Time already waited: current time minus submission time
Node capacity	GPUs per node	Node inventory in the trace	Static
Node telemetry	energy, SM utilisation, PCIe rx and tx, power, GPU memory	DCGM, training window jobs only	Frozen per node means
Engineered ratios	<code>cpu_mem_ratio</code> , <code>gpu_cpu_ratio</code> , <code>energy_per_gpu</code> , ranges	Derived from the rows above	Computed from the values above
Target	GPU execution time	DCGM	Never an input; label in training, true duration in evaluation

Table 1: Feature Availability at Scheduling Time

One caution applies to the offline accuracy figures. In the training table built by the (job, node) join, the telemetry columns of a row are measured during that job's own execution, and quantities such as `energy_consumed_joules` grow with execution time. Offline scores computed on these rows (Table 3) therefore describe how well runtime can be recovered when a job's own telemetry is known, which is an optimistic upper bound on accuracy at decision time. The scheduler does not use this information: each of the 225 feature rows pairs the current job with the training window aggregates of one node, as described in Section 3.3. The scheduling results in Section 4 are therefore not affected by this form of leakage, but the R^2 values in Table 3 should not be read as decision time accuracy.

3.3 Model Training

XGBoost naturally handles mixed feature types; that's why it is resilient to the highly skewed distributions typical of HPC job logs, and achieves strong regression accuracy on tabular data without requiring the extensive architecture search required by deep learning models. It is chosen for runtime prediction [15]. Because the model must generalise across a temporal distribution shift, later jobs in the dataset often run longer than early ones; its built-in L1 and L2 regularisation parameters are especially helpful in this situation.

The dataset is divided into 70% training (51,334 jobs), 15% validation (11,000 jobs), and 15% test (11,001 jobs) chronologically rather than at random. Random splitting would make it possible for future job information to show up during training, leading to evaluation findings that are overly hopeful. A true scheduler must train on previous jobs in order to generalise to future ones, and this direction must be reflected in the assessment. After sorting by submission time, jobs 1 to 51,334 form the training set, jobs 51,335 to 62,334 the validation set and jobs 62,335 to 73,335 the test set, so every validation and test job was submitted after the last training job. The same boundaries govern every later stage: the node telemetry aggregates and the XGBoost model use the training jobs only, early stopping uses the validation jobs, PPO training episodes are drawn from the training jobs only, and the test jobs are used once, for the final comparison in Section 4.1.

Learning rate 0.05, maximum tree depth 7, row and column subsampling 0.8, L1 regularisation 0.1, and L2 regularisation 1.0 are important hyperparameters. Early halting automatically determines the ideal number of trees without the need for manual adjustment by monitoring validation RMSE with a patience of 50 rounds. MAE, RMSE, and R^2 are used to report performance in both log space (which displays learning quality) and seconds space (which can be interpreted in real-time units).

Crucially, the trained predictor does not generate a single prediction per job when utilised within the RL environment. Rather, it creates a feature matrix where each row matches the attributes of the current job with a separate node's past telemetry record to produce 225 node-specific forecasts every scheduling step. With distinct values out of 225 nodes and forecasts ranging from 46.5 seconds to 20,353.5 seconds across nodes for the same job, this per-node inference is what gives the ML hints of their significance. The RL agent receives actionable guidance instead of generic, uninformative recommendations because of this distinction.

3.4 Cluster Simulation Environment

The 225-node MIT SuperCloud cluster is modelled by the RL agent within a specially designed environment based on the Gymnasium framework, with GPU capacities taken straight from the dataset. Two dynamic data structures monitor the live cluster state during the simulation: the `busy_until`, which records when each node completes its current task, and `free_gpus`, which records the GPUs that are currently accessible per node. Every clock advance and scheduling decision results in an update to both.

Instead of using absolute Unix timestamps, time is managed in relative seconds from the beginning of each episode. Because consecutive jobs in the real dataset often have the same submission timestamp, this design decision is required. The simulation clock would never move past the massive absolute `busy_until` values if absolute timestamps were used, preventing nodes from ever being released. Relative time guarantees that the simulation accurately captures the dynamics of node release and job arrival. The environment chooses a random starting position in the job queue at the beginning of each episode. This forces the learned strategy to generalise over the whole range of workload patterns in the 51,334 job training split by exposing the agent to a distinct window of 500 real jobs per episode.

There are four components in the observation vector that the agent is shown at each phase. The CPU request, RAM request, GPU count, job priority, and waiting time are the first five dimensions that represent normalised job features. The ML top-4 hints are encoded in the next eight dimensions: four values for the relevant node indices and four values

for the anticipated runtimes of the top-4 nodes. The free GPU fraction for each node at that precise instant is encoded in the following 225 dimensions, where 1.0 denotes complete freedom, and 0.0 denotes complete busyness. Normalised queue length is encoded in the final dimension.

The fundamental concept of the hybrid framework is reflected in this observation design as the agent concurrently sees what the ML model suggests (hints) and what the cluster can now support (cluster state), providing it with the information required to make decisions that neither component could support on its own.

The agent is free to choose any node in the Discrete (225) action space. Instead of being restrictive, the ML top-4 indications are informative. When cluster conditions make a different node more suitable, such as when all four hint nodes are now busy and a less-predicted node is instantly available, this maintains the agent's capacity to override ML suggestions. The activity is immediately rerouted to the capable node with the earliest busy_until value when the chosen node's GPU capacity is insufficient for a task. The simulation clock advances to its free time when the chosen node is busy, and the work is scheduled there. The reward implicitly penalises the subsequent delay.

Simulation assumptions

The simulator is trace driven and rests on the following explicit assumptions. (i) Arrivals: jobs are released in recorded submission order at their recorded submission times, measured relative to the first job of the window. (ii) Placement: every job runs on a single node; a job that requests several GPUs must fit within the GPU capacity of that node, and if the chosen node is too small the job is moved to the capable node with the earliest busy_until, as described above. (iii) Node timeline: each node holds one busy_until value, so a job starts at the later of the current time and busy_until of its node, and free_gpus records the GPUs held by the running job. (iv) Durations: the recorded DCGM execution time is the job's duration; slowdown from colocated work and network contention is not modelled. (v) Failures: node failures, maintenance, preemption and cancellations are not simulated; jobs that did not finish are removed during preprocessing. (vi) Queue: at each step one arrived job is placed, and the queue length in the observation counts jobs that have arrived but are not yet placed. (vii) Priority is an input feature only and is not enforced by the simulator. These assumptions apply identically to PPO and to every baseline.

Choosing nodes with short anticipated completion times and utilising the ML hints when appropriate are two behaviours that the reward is intended to jointly incentivise. For every choice made about scheduling:

$$\text{Reward} = (\text{best_completion} / \text{chosen_completion}) + \text{hint_bonus}$$

where chosen_completion is the same amount for the selected node, best_completion is the lowest value of busy_until[n] + predicted_runtime[n] across all competent nodes, and hint_bonus is 0.2 if the selected node was in the ML top-4 and 0.0 otherwise. For the best option, the ratio is 1.0, and as the decision deteriorates, it approaches 0. A maximum episode reward of 600 across 500 steps is possible with a maximum payout per step of 1.2. Instead of a binary good/bad signal, this continuous gradient offers insightful feedback at each stage that directs PPO updates toward better policies.

Both completion terms are computed with the predictor's runtimes, because these are the only runtimes known when the decision is taken. The recorded runtime is used only to advance busy_until after the job is placed, so the dynamics of the environment follow the real trace while the reward uses decision time information. During evaluation (Section 4.1) the reward plays no role, and every metric is computed from recorded runtimes. The hint bonus is a fixed shaping term, and it can pull the policy toward hinted nodes even when a node outside the top four would finish earlier; Section 4.2 bounds how much of the observed reward it can explain.

Three factors lead to the selection of Proximal Policy Optimisation (PPO) [16]. First, when the action space covers 225 nodes and the observation space, its clipped surrogate objective avoids the huge destabilising policy updates that simple policy gradient approaches generate. Second, it increases sample efficiency by reusing every batch of experience that has been gathered over several gradient update epochs. Third, it prevents early convergence to a suboptimal policy by maintaining exploration during training because of its entropy bonus.

With 256 neurons per layer, the agent's two-layer MLP policy network has enough ability to handle the 239-dimensional observation without adding needless processing cost. The following are important hyperparameters: batch size 64, n_steps 2,048, n_epochs 10, learning rate 0.01 (1×10^{-2}), entropy coefficient 0.01, discount factor 0.99, and clip range 0.2. Stable-Baselines3 [17] is used to implement the agent, and it is trained over 500,000 timesteps. Because PPO shows entropy-driven reward erosion in subsequent training, a special RewardTrackingCallback logs the mean

episode reward every 2,000 steps and automatically saves the best model checkpoint. So that the best checkpoint is utilised for evaluation instead of the final model.

The learning rate of 0.01 was chosen in preliminary runs. It is larger than the Stable-Baselines3 default of 3×10^{-4} , and together with the entropy bonus it is a likely cause of the reward decline after 288,000 steps (Table 4), which is why the best checkpoint rather than the final model is evaluated.

Reproducibility. Table 2 lists the complete agent and environment configuration. The random number generators of Python, NumPy, PyTorch and the environment are seeded with one fixed seed for training, and the PPO policy is evaluated deterministically, taking the most probable action at every step. The best checkpoint is selected by mean training episode reward, logged every 2,000 steps, so the test set plays no part in that choice. The experiments were implemented in Python with XGBoost [15], Gymnasium, PyTorch and Stable-Baselines3 [17] on a single machine. The source code, preprocessing scripts and configuration file are available from the corresponding author on request.

Setting	Value
Algorithm	PPO [16], Stable-Baselines3 [17] implementation
Policy and value networks	MLP, two hidden layers of 256 units
Observation	239 values: job features 5, hint node indices 4, hint runtimes 4, free GPU fraction 225, queue length 1
Action space	Discrete(225): any node
Episode	500 consecutive jobs from a random start inside the training split
Reward	best_completion / chosen_completion + hint_bonus (0.2 if the node is in the ML top 4, else 0)
Total timesteps	500,000
n_steps, batch size, epochs	2,048, 64, 10
Learning rate	0.01 (1×10^{-2}), constant
Discount factor, GAE λ	0.99, 0.95
Clip range, entropy coefficient	0.2, 0.01
Value loss coefficient, max gradient norm	0.5, 0.5
Checkpoint selection	Best mean episode reward, checked every 2,000 steps
Evaluation policy	Deterministic; one continuous replay of the 11,001 test jobs

Table 2: PPO Agent and Environment Configuration

4. Results and Discussion

Early stopping causes training to converge at tree 113 (out of a maximum of 3,000), indicating successful regularisation. The predictor's performance across data splits is shown in Table 3.

Split	MAE (log)	RMSE (log)	R ² (log)	MAE (sec)	RMSE (sec)	R ² (sec)
Train	0.1799	0.3913	0.9863	9,889	64,940	0.4769
Validation	0.3441	0.6533	0.9572	17,473	75,230	0.2295
Test	0.4177	0.7299	0.9476	12,218	47,013	0.1980

Table 3: XGBoost Runtime Predictor Results

The model explains 94.76% of the variance in real GPU execution times over 225 heterogeneous nodes, according to the test set R² of 0.9476 in log space. Regularisation inhibits overfitting, as evidenced by the stable training-validation gap. The log-space metric accurately captures the model's ability to rank nodes by expected performance—the operationally relevant capability—while the low seconds-space R² (0.1980) is expected and matters less for the scheduling application: a small number of very long jobs produce large squared residuals in seconds' space. As explained in Section 3.2, these offline scores use telemetry measured during each job's own run and are an upper bound on accuracy at decision time.

Energy-related variables (`energy_consumed_joules`, `energy_per_gpu`), which indicate the physical relationship between execution time and energy consumption, are the most predictive, according to feature importance analysis. The difference between compute-intensive and I/O-intensive workloads is captured by the PCIe transmit bandwidth range (`pcie_tx_range`), which raw resource request features are unable to convey. The majority of the predictive signal is driven by telemetry-enriched data, as evidenced by the comparatively low value of job request features (GPU count, CPU count). This dominance is consistent with the caution in Section 3.2: the energy a job consumes grows with its duration, so this feature carries information about the target that is unavailable before the job runs, which is why the scheduler uses node level historical means instead.

PPO training milestones across 500,000 timesteps are reported in Table 4.

Timestep	Mean Episode Reward	Note
2,000	61.2	Random policy baseline
60,000	86.4	Policy begins to stabilize
200,000	174.4	Strong learning from ML hints
288,000	187.1	Best model saved
500,000	124.5	Final model (entropy decay)

Table 4: PPO Agent Training Milestones

There are three different stages in the learning curve. Phase 1 (0–60K steps): the agent explores the 225-node action space without a cohesive policy, resulting in a slow improvement. Phase 2 (60K–288K steps): the `hint_bonus` signal drives the agent's quick development as it learns to link ML hint information to greater rewards. Phase 3 (288K–500K steps): a slow decline in rewards follows, consistent with policy drift under the entropy bonus and the relatively large learning rate. At step 288,000, the optimal model obtains a mean episode reward of 187.1, which is 3.1× better than the random baseline (61.2). The quick Phase 2 improvement is consistent with the ML hints accelerating learning in a broad action space, although the reward curve alone cannot separate the two reward terms (Section 4.2).

4.1 Evaluation

Evaluation protocol. The 11,001 test jobs are replayed as one continuous sequence rather than in 500 job episodes, so no episode boundaries are involved. The cluster starts empty, with `busy_until` equal to zero and all GPUs free on every node, and it is not reset during the replay. Every scheduler, PPO included, receives the same jobs in the same order at the same submission times, on the same 225 nodes with the same GPU capacities, under the assumptions of Section 3.4. Durations are the recorded GPU execution times, never predicted ones. The PPO policy acts deterministically and the three baselines are deterministic, so each scheduler produces exactly one schedule for this trace.

Metrics. The waiting time of a job is its start time minus its submission time, averaged over all test jobs. Makespan is the time from the first submission to the last completion. GPU utilization is the GPU time consumed by all jobs, the sum of GPU count times recorded runtime, divided by the GPU time available, which is the total number of GPUs multiplied by the makespan of the scheduler being measured. All times are reported in hours.

Baseline configurations. FCFS places jobs strictly in submission order. Backfilling follows the EASY scheme [18]: when the job at the head of the queue cannot start, a later job may start first provided it does not delay the reservation of the head job. SJF (Oracle) sorts the whole test set once by recorded runtime, which no real scheduler knows, and places jobs in that order. The three baselines share one node placement rule, which assigns each job to a node able to hold its GPU request without considering how soon that node becomes free, whereas the PPO agent observes the free GPU fraction of every node. This difference in placement information is part of what the comparison measures and is discussed in Section 4.3.

The identical 11,001 held-out test jobs from the chronologically last 15% of the dataset, such that these jobs were never seen during training, are used to evaluate each of the four schedulers using real recorded GPU execution times. Table 5 shows the outcomes.

Scheduler	Avg. Waiting Time (h)	Makespan (h)	GPU Utilization
FCFS	121.5 h	3,453.1 h	7.24%
SJF (Oracle)	180.1 h	3,636.5 h	6.88%
Backfilling	121.5 h	3,453.1 h	7.24%
PPO Hybrid (Proposed)	1.3 h	533.4 h	46.90%

Table 5: Scheduler Comparison on 11,001 Test Jobs

The average waiting time is reduced by 98.9% with the PPO hybrid, from 121.5 hours (FCFS, backfilling) to 1.3 hours. The reduction is 99.3% when compared to the SJF oracle. Backfilling and FCFS focus jobs on a small number of nodes while others are idle; the PPO agent uses ML predictions to more fairly distribute workload, allowing for earlier job initiation. Despite having complete runtime knowledge, SJF's average waiting time (180.1 hours) is noticeably worse than FCFS's because of how the oracle is implemented. It sorts the whole test set by runtime once, and a job cannot start before it is submitted, so a short job submitted late is placed ahead of long jobs submitted early. The node then idles until the short job arrives while the earlier long jobs wait, which raises both waiting time and makespan. A queue level SJF that orders only the jobs already waiting would avoid this effect and is a stronger baseline (Section 4.3). By taking both availability and anticipated runtime into account, the PPO hybrid overcomes this pitfall.

Compared to FCFS, the PPO hybrid reduces the makespan by 84.6%, from 3,453.1 hours to 533.4 hours. This saves almost 120 days of cluster time by finishing the 11,001-job test batch in 533 hours as opposed to 3,453. Backfilling produces exactly the same schedule as FCFS, job for job, and identical values on all three metrics follow from that single fact. The metrics are not independent: waiting time and makespan are read off the same schedule, and utilization is a fixed quantity divided by makespan (Table 6). The schedule is identical because of the way the simulator dispatches jobs. Each job is committed to a node timeline in submission order and starts at the later of its submission time and

busy_until of that node. An idle interval on a node can only open before a job that arrived after the node became free, and every job considered afterwards arrived later still, so it cannot move into that interval. No job can therefore start earlier without delaying a job placed before it, and the EASY condition [18] is never satisfied. This is a property of the dispatch model and of the shared placement rule rather than evidence that backfilling is ineffective in general [3].

The PPO hybrid outperforms the best baseline by 39.66 percentage points, achieving 46.90% GPU utilisation as opposed to 7.24% with FCFS and backfilling and 6.88% under SJF. A 7.24% utilization rate indicates that only 7.24% of the evaluation time was spent by cluster GPUs actively executing tasks; the remaining 92.76% reflects idle GPU-hours. SJF's slightly lower figure (6.88%) follows directly from its longer makespan. Utilization must be read together with its denominator, which is each scheduler's own makespan (Section 4.1). Every scheduler runs the same jobs for their recorded durations, so the numerator is the same for all four and utilization is inversely proportional to makespan. Table 6 confirms this: utilization multiplied by makespan equals 250.0 to 250.2 hours for every scheduler, the small spread being rounding in the reported values. The 6.5× utilization gain is therefore the makespan reduction ($3,453.1 / 533.4 = 6.47$) expressed in another unit, not a separate improvement. Over a shared window, such as the FCFS makespan of 3,453.1 hours, every scheduler would show the same 7.24%, because the same GPU hours are consumed. We keep the metric for comparability with earlier work and treat waiting time and makespan as the primary results.

Scheduler	Makespan (h)	GPU utilization (%)	Utilization × makespan (h)
FCFS	3,453.1	7.24	250.0
SJF (Oracle)	3,636.5	6.88	250.2
Backfilling	3,453.1	7.24	250.0
PPO Hybrid (Proposed)	533.4	46.90	250.2

Table 6: GPU Utilization Is Inversely Proportional to Makespan (values from Table 5)

Conventional schedulers use only one criterion to make decisions: backfilling relies on availability with little runtime awareness, SJF on runtime alone, and FCFS on arrival order. None adapts its approach based on feedback. The PPO agent simultaneously learns a multi-dimensional policy that takes into account node availability (cluster state vector), expected runtime quality (ML hints), and their interaction as demonstrated by previous incentives. The outcome against the SJF oracle shows that, in this setting, perfect runtime knowledge applied through a fixed ordering rule is not enough, and that placement decisions which account for node state matter more. It does not by itself show that the ML hints are the cause of the gain, which is examined next.

4.2 Contribution of the ML Hints and the Reward Components

The reward of Section 3.4 is bounded by 1.2 per decision, of which at most 0.2 comes from the hint bonus. Table 7 converts the milestones of Table 4 into mean reward per decision and bounds the share of each term. At the best checkpoint the policy earns 0.374 per decision. Even if every choice had been a hinted node, the completion ratio would still contribute at least 0.174 per decision; conversely, the hint bonus can account for at most 53% of the best reward. The reward curve therefore cannot separate learning to place jobs well from learning to collect the bonus, which is the concern that a fixed hint bonus raises.

Timestep	Episode reward	Reward per decision	Completion ratio term	Hint bonus term
2,000	61.2	0.122	0 to 0.122	0 to 0.122
60,000	86.4	0.173	0 to 0.173	0 to 0.173
200,000	174.4	0.349	0.149 to 0.349	0 to 0.200
288,000 (best)	187.1	0.374	0.174 to 0.374	0 to 0.200
500,000 (final)	124.5	0.249	0.049 to 0.249	0 to 0.200

Table 7: Bounds on the Two Reward Terms (500 decisions per episode; hint bonus at most 0.2 per decision)

The same bound gives a useful reference point. A policy that always picks the node with the smallest predicted completion time scores exactly 1.0 on the ratio term at every decision, plus the bonus whenever that node is hinted, so its episode reward lies between 500 and 600, against 187.1 for the best PPO checkpoint. PPO therefore does not reproduce this greedy choice, and its gains in Table 5 cannot be credited to that choice either. Two readings are possible: the policy gives up some immediate completion time for placements that keep nodes free for later arrivals, which the myopic reward does not credit, or the reward is only loosely aligned with the evaluation metrics. Telling them apart needs the controlled comparisons listed in Section 4.3, and until they are run the claim that per step ML hints drive the gain remains a hypothesis supported by the design, not a measured effect.

4.3 Threats to Validity and Limitations

Statistical evidence. All PPO results come from one training run with one seed and one deterministic test replay. The baselines are deterministic, so they have no seed variance on a fixed trace, but PPO is known to vary noticeably across seeds [21, 22], and no confidence interval can be given for it here. The size of the gap, for example 1.3 against 121.5 hours of mean waiting time, makes a reversal of the ranking against these particular baselines unlikely, but the exact percentages should be read as the result of a single run. The first extension of this work is to report the mean, standard deviation and bootstrap confidence intervals over at least five seeds and several disjoint test windows [22].

Baseline strength and ablation. The baselines use a placement rule that ignores node availability, while PPO sees the free GPU fraction of every node. Part of the gap is therefore likely to come from spreading load across idle nodes, which a simple rule such as earliest available node or smallest predicted completion time would also capture. Isolating the hints requires PPO without hints in the observation, PPO without the hint bonus, PPO with other rewards such as negative waiting time, the greedy smallest predicted completion heuristic of Section 4.2, and FCFS, EASY backfilling and queue level SJF with earliest available node placement, all run under the protocol of Section 4.1.

Simulator fidelity. Arrivals, requests and durations come from the real SLURM and DCGM logs [14, 23], but the simulator treats each node as one sequential timeline, places every job on one node, and ignores failures, preemption and interference (Section 3.4). It has not yet been validated against a recognised simulator such as Batsim [19] or against the queue times recorded by SLURM for the same jobs, so the results compare policies inside this model rather than predicting production behaviour.

Deployment. No experiment here runs online, on a second cluster or with measured overhead. Each decision needs 225 XGBoost predictions and one forward pass of a small MLP, which is inexpensive in principle, but this cost has not been measured. We therefore make no claim about production deployment until these steps are completed.

5. Conclusion and Future Work

A hybrid machine learning and reinforcement learning scheduler for GPU cluster computing was introduced in this paper to address the fundamental drawbacks of current methods: pure RL agents explore inefficiently without prior knowledge, while prediction models lack real-time adaptability. The suggested framework embeds the XGBoost predictor as a live per-step component inside a customised Gymnasium environment, trains a PPO agent to combine ML guidance with real-time cluster awareness, and trains an XGBoost predictor offline on actual MIT SuperCloud workload data incorporating NVIDIA DCGM GPU telemetry ($R^2 = 0.9476$ in log space on held out test data, an offline figure computed with job telemetry available).

The suggested scheduler outperforms FCFS, backfilling, and the SJF oracle with perfect runtime knowledge on all three metrics when 11,001 held out test jobs are replayed in the trace driven simulator. It achieves a 98.9% reduction in average waiting time (121.5 hours $\rightarrow$ 1.3 hours), an 84.6% reduction in makespan (3,453 hours $\rightarrow$ 533 hours), and a 6.5 $\times$ improvement in GPU utilisation (7.24% $\rightarrow$ 46.90%). Because utilization is measured over each scheduler's own makespan, the utilization gain restates the makespan gain. These results come from a single training seed, in a simulator with explicit simplifying assumptions, against baselines whose placement rule ignores node availability, and the reward analysis shows that they do not yet isolate the contribution of the ML hints. They show that the tightly coupled hybrid design is promising; they do not yet establish that per step coupling is the cause of the gain.

Future research will focus on the following areas: (i) evaluation over multiple seeds and disjoint test windows with confidence intervals, together with the ablations and stronger baselines listed in Section 4.3; (ii) validation of the simulator against Batsim and against the queue times recorded by SLURM; (iii) cross cluster generalization experiments; (iv) energy aware reward functions that incorporate GPU power consumption; (v) extension to multi node job allocation; and (vi) online retraining with measured decision overhead, as a precondition for any production use. These additions would increase the framework's practical usefulness and explore the potential benefits of predictive intelligence and adaptive scheduling.

References

- [1] **Journal Article** Peng, Y., Bao, Y., Chen, Y., Wu, C., Meng, C., & Lin, W. (2021). DL2: A deep learning-driven scheduler for deep learning clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(8), 1947–1960. <https://doi.org/10.1109/TPDS.2021.3052895>
- [2] **Conference Paper** Zhang, D., Dai, D., He, Y., Bao, F. S., & Xie, B. (2020). RLScheduler: An automated HPC batch job scheduler using reinforcement learning. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15. <https://doi.org/10.1109/SC41405.2020.00035>
- [3] **Journal Article** Mu'alem, A. W., & Feitelson, D. G. (2001). Utilisation, predictability, workloads and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6), 529–543. <https://doi.org/10.1109/71.932708>
- [4] **Journal Article** Bao, Y., Peng, Y., & Wu, C. (2022). Deep learning-based job placement in distributed machine learning clusters with heterogeneous workloads. *IEEE/ACM Transactions on Networking*, 31(2), 634–647. <https://doi.org/10.1109/TNET.2022.3203314>
- [5] **Journal Article** Hou, Z., Shen, H., Feng, Q., Lv, Z., Jin, J., Zhou, X., & Gu, J. (2024). Optimising job scheduling by using broad learning to predict execution times on HPC clusters. *CCF Transactions on High Performance Computing*, 6(4), 365–377. <https://doi.org/10.1007/s42514-023-00137-z>
- [6] **Journal Article** Fan, Y., Li, B., Favourite, D., Singh, N., Childers, T., Rich, P., & Lan, Z. (2022). DRAS: Deep reinforcement learning for cluster scheduling in high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(12), 4903–4917. <https://doi.org/10.1109/TPDS.2022.3205325>
- [7] **Conference Paper** Mondal, S. S., Sheoran, N., & Mitra, S. (2021). Scheduling of time-varying workloads using reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10), 9000–9008. <https://doi.org/10.1609/aaai.v35i10.17088>
- [8] **Journal Article** Zhao, X., & Wu, C. (2021). Large-scale machine learning cluster scheduling via multi-agent graph reinforcement learning. *IEEE Transactions on Network and Service Management*, 19(4), 4962–4974. <https://doi.org/10.1109/TNSM.2021.3138906>
- [9] **Journal Article** Ayadi, H., An, A., Shao, Y., Pourmedheji, H., Deng, J., Huang, J. X., & Zhou, H. (2025). Topology-aware GPU job scheduling with deep reinforcement learning and heuristics. *Journal of Parallel and Distributed Computing*, 105138. <https://doi.org/10.1016/j.jpdc.2025.105138>
- [10] **Conference Paper** Ryu, J., & Eo, J. (2023). Network contention-aware cluster scheduling with reinforcement learning. *IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, 2742–2745. <https://doi.org/10.1109/ICPADS60453.2023.00370>
- [11] **Book Chapter** Souza, A., Pelckmans, K., & Tordsson, J. (2021). A HPC co-scheduler with reinforcement learning. In D. Klusáček, W. Cirne, & G. P. Rodrigo (Eds.), *Job Scheduling Strategies for Parallel Processing* (pp. 126–148). Springer. https://doi.org/10.1007/978-3-030-88224-2_7
- [12] **Preprint** Hu, Z., Wang, C., Shu, Y., & Zhu, L. (2024). *Learning interpretable scheduling algorithms for data processing clusters*. arXiv. <https://doi.org/10.48550/arXiv.2405.19131>

- [13] **Conference Paper** Luo, Z., Liu, J., Lee, M., & Shroff, N. B. (2025). Prediction-assisted online distributed deep learning workload scheduling in GPU clusters. *IEEE INFOCOM 2025*, 1–10. <https://doi.org/10.1109/INFOCOM55648.2025.11044761>
- [14] **Conference Paper** Tang, B. J., Chen, Q., Weiss, M. L., Frey, N. C., McDonald, J., Bestor, D., & Samsi, S. (2022). The MIT supercloud workload classification challenge. *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 708–714. <https://doi.org/10.1109/IPDPSW55747.2022.00122>
- [15] **Conference Paper** Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. <https://doi.org/10.1145/2939672.2939785>
- [16] **Preprint** Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal policy optimisation algorithms*. arXiv. <https://doi.org/10.48550/arXiv.1707.06347>
- [17] **Journal Article** Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268), 1–8. <http://jmlr.org/papers/v22/20-1364.html>
- [18] **Book Chapter** Lifka, D. A. (1995). The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing* (Lecture Notes in Computer Science, Vol. 949). Springer. https://doi.org/10.1007/3-540-60153-8_35
- [19] **Book Chapter** Dutot, P.-F., Mercier, M., Poquet, M., & Richard, O. (2017). Batsim: A realistic language-independent resources and jobs management systems simulator. In *Job Scheduling Strategies for Parallel Processing* (Lecture Notes in Computer Science, Vol. 10353, pp. 178–197). Springer. https://doi.org/10.1007/978-3-319-61756-5_10
- [20] **Journal Article** Kaufman, S., Rosset, S., Perlich, C., & Stitelman, O. (2012). Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data*, 6(4), Article 15. <https://doi.org/10.1145/2382577.2382579>
- [21] **Conference Paper** Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep reinforcement learning that matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1). <https://doi.org/10.1609/aaai.v32i1.11694>
- [22] **Conference Paper** Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A., & Bellemare, M. G. (2021). Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34. <https://arxiv.org/abs/2108.13264>
- [23] **Preprint** Samsi, S., Weiss, M. L., Bestor, D., Li, B., Jones, M., Reuther, A., Edelman, D., Arcand, W., Byun, C., Holodnack, J., Hubbell, M., Kepner, J., Klein, A., McDonald, J., Michaleas, A., Michaleas, P., Milechin, L., Mullen, J., Yee, C., . . . Gadepally, V. (2021). *The MIT Supercloud dataset*. arXiv. <https://doi.org/10.48550/arXiv.2108.02037>