Hybrid Learning and Optimization-Based Dynamic Scheduling for DL Workloads on Heterogeneous GPU Clusters

Shruti Dongare
Virginia Tech, USA
dshruti20@vt.edu

Redwan Ibne Seraj Khan
Virginia Tech, USA
redwan@vt.edu

Hadeel Albahar
Kuwait University, Kuwait
hadeel.albahar@ku.edu.kw

Nannan Zhao
Northwestern Polytechnical University, China
nannanzhao@nwpu.edu.cn

Diego Meléndez-Maita
Virginia Tech, USA
dmelendezmaita@vt.edu

Ali R. Butt
Virginia Tech, USA
butta@cs.vt.edu

Abstract

Modern cloud platforms increasingly host large-scale deep learning (DL) workloads, demanding high-throughput, low-latency GPU scheduling. However, the growing heterogeneity of GPU clusters and limited visibility into application characteristics pose major challenges for existing schedulers, which often rely on offline profiling or application-specific assumptions. We present RLTune, an application-agnostic reinforcement learning (RL)–based scheduling framework that dynamically prioritizes and allocates DL jobs on heterogeneous GPU clusters. RLTune integrates RL–driven prioritization with MILP–based job-to-node mapping to optimize system-wide objectives such as job completion time (JCT), queueing delay, and resource utilization. Trained on large-scale production traces from Microsoft Philly, Helios, and Alibaba, RLTune improves GPU utilization by up to 20%, reduces queueing delay by up to 81%, and shortens JCT by as much as 70%. Unlike prior approaches, RLTune generalizes across diverse workloads without requiring per-job profiling, making it practical for cloud providers to deploy at scale for more efficient, fair, and sustainable DL workload management.

CCS Concepts

• **Computing methodologies** → **Planning and scheduling**; **Machine learning**; **Reinforcement learning**; • **Software and its engineering**;

Keywords

Cluster Management, Workload Scheduling, Reinforcement Learning, Deep Learning, Traces

ACM Reference Format:
Shruti Dongare, Redwan Ibne Seraj Khan, Hadeel Albahar, Nannan Zhao, Diego Meléndez-Maita, and Ali R. Butt. 2025. Hybrid Learning and Optimization-Based Dynamic Scheduling for DL Workloads on Heterogeneous GPU Clusters. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772257

1 Introduction

Cloud providers increasingly support large-scale deep learning (DL) workloads in shared GPU clusters, powering applications from scientific computing to commercial AI services [14, 20, 29, 35, 42, 54]. These workloads consume a significant share of data center resources [1, 23] due to their compute intensity and prolonged runtimes, prompting the development of specialized DL infrastructure [2–4, 6]. The extended runtimes and computational intensity of such workloads demand high-performance GPUs for massive parallel processing and memory efficiency [9].

Modern cloud-scale DL clusters often comprise a mix of GPU generations, as operators incrementally upgrade hardware while keeping legacy nodes online to amortize costs and maintain capacity [16, 48, 58, 65]. This evolution results in a highly heterogeneous scheduling landscape, where architectural disparities and user preference for faster GPUs [27] create imbalances in resource demand and utilization. In multi-tenant cloud environments, this leads to prolonged queueing delays, degraded performance for certain job classes, and inefficient GPU provisioning at scale. Addressing these challenges is particularly difficult because batch job scheduling is NP-hard, and static policies fail to adapt to the dynamic interplay of job characteristics, cluster state, and workload churn.

Traditional schedulers like Slurm [5] provide scalable, application-agnostic job dispatching and have powered both TOP500 HPC systems [7] and commercial cloud backends such as AWS. However, DL workloads introduce new scheduling pressures, e.g., iterative execution [46, 51], gang scheduling [24], intra-node GPU fragmentation [18, 62], and resource sharing [60] that demand more adaptive and workload-aware mechanisms. These new characteristics has led to DL-focused schedulers such as Gavel [49] and Sia [33], which use predictive, preemptive strategies to optimize throughput and fairness. Yet, these systems rely on profiling, matching new jobs to previously seen ones based on model architecture, dataset, or runtime behavior. In practice, cloud platforms lack detailed metadata due to user privacy, model diversity, and rapidly changing workloads, as seen in real-world traces like Microsoft Philly [34] and Alibaba PAI [59]. As a result, profiling-dependent methods limit scalability, and yield unreliable approximations. Many DL jobs are black boxes (e.g., proprietary models or novel architectures), making profiling infeasible and motivating learning-based alternatives that operate effectively in production clouds. While profiling already faces limitations in diverse clusters, its relevance further reduces as emerging deployment trends increasingly dedicate separate pools to specialized workloads such as LLM serving or DLRM training to improve stability. This specialization narrows cross-application diversity, leaving fewer opportunities for profiling-based schedulers to provide benefit. In contrast, dynamic, profiling-free schedulers that adapt to runtime signals remain effective across both mixed and dedicated environments.

SoCC '25, November 19–21, 2025, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11.
https://doi.org/10.1145/3772052.3772257

Table 1: *Comparison of Existing ML/DL Job Schedulers and RL-Based CPU Schedulers with RLTune.*

Scheduler	Scheduling Strategy	GPU Het.	App-Agn.	Offline Prof. free	Dyn. Policy	Preempt.	Elastic
Slurm	Multi-factor	✓	✓	✓	limited	configurable	✗
QSSF[30]	Historic data	✓	✓	✗	✗	✗	✗
Gavel[49]	Gavel	✓	✗	✗	✓	✓	✗
Pollux[52]	Pollux	✗	✗	✗	✓	✗	✓
Sia[33]	Sia	✓	✗	✗	✓	✓	✓
RLTune (Ours)	RL + MILP	✓	✓	✓	✓	✗	✗
SchedInspector[64]	RL	✗ (CPU)	✓	✓	limited	✗	✗
RLScheduler[63]	RL	✗ (CPU)	✓	✓	limited	✗	✗

Although Gavel and Sia rely on performance prediction, RLTune instead follows an application-agnostic design, in the DL scheduling domain, which means that their decisions do not depend on model semantics such as architecture, dataset, optimizer, batch size, or training objective. RLTune is also profiling-free as it does not build or depend on per-job performance models, instead using only user-submitted metadata and queue-level information to guide scheduling in real time. Table 1 summarizes the design trade-offs across representative DL schedulers. Application-agnostic methods scale to real-world traces lacking detailed metadata, but require more intelligent mechanisms to adaptively prioritize jobs and make fine-grained allocation decisions in heterogeneous, resource-constrained clusters.

We argue for a complementary learning-driven yet application-agnostic approach to DL job scheduling. Reinforcement learning (RL)[36] is well suited for this goal. It learns from experience rather than static heuristics, operates in partially observable environments, and optimizes long-term outcomes like JCT and resource utilization. Prior works demonstrated the effectiveness of RL in CPU scheduling (e.g., RLScheduler [63], SchedInspector [64]), applying RL to GPU scheduling introduces new challenges such as co-allocation constraints, heterogeneity, and fragmentation arising from the hierarchical, multi-node nature of GPU clusters. These factors fundamentally change the learning problem and reshape the reward dynamics, thus requiring novel design. To address these, we present RLTune, a dynamic scheduler that couples RL-based dynamic prioritization (DP) with mixed-integer linear programming (MILP)–based resource allocation. RLTune leverages job-level and system-level signals (e.g., user metadata, resource availability, queue state) to construct engineered features and select the most relevant ones as input to the RL agent for Dynamic Prioritization (DP), while MILP performs multi-dimensional allocation across GPUs, CPUs, and memory in alignment with cluster-level objectives. While RL and MILP have been individually applied to scheduling, RLTune uniquely separates and couples them using RL for proactive prioritization and MILP for multi-dimensional look-ahead allocation forming a unified hybrid framework that learns without per-application profiling and generalizes across heterogeneous clusters.

To evaluate RLTune, we use three diverse, publicly available DL workload traces: Philly [34], Helios [30], and Alibaba'20 [59], and assess performance using metrics waiting time, JCT, bounded slowdown (BSLD), and resource utilization. Our evaluation examines how RLTune captures fine-grained opportunities for long-term efficiency through safe, reward-driven trade-offs in job prioritization and allocation. We compare the trained RL policy against state-of-the-art baselines [5, 30, 63, 64], demonstrating improvements across multiple performance objectives.

Specifically, RLTune makes the following contributions:

- It captures key scheduling challenges in heterogeneous GPU clusters such as application diversity, resource fragmentation, and profiling infeasibility, motivating an application-agnostic learning approach.
- Couples RL-based dynamic prioritization with MILP-based multi-resource allocation, leveraging runtime features for long-term optimization without per-application profiling.
- We train and evaluate RLTune on real-world traces, measuring queueing delay, JCT, BSLD, and resource utilization.
- RLTune achieves up to 81% lower queueing delay, 70% shorter JCT, and 20% higher GPU utilization than state-of-the-art schedulers, across diverse workloads and cluster setups.
- We deploy RLTune on a heterogeneous Slurm cluster, demonstrating its effectiveness in improving end-to-end scheduling under real-world conditions.

2 Background and Motivation

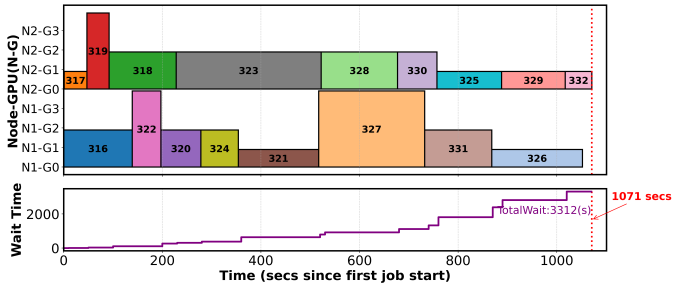


Figure 1: *Slurm's Multi-Factor Priority scheduling (per-job-per-node allocation)*

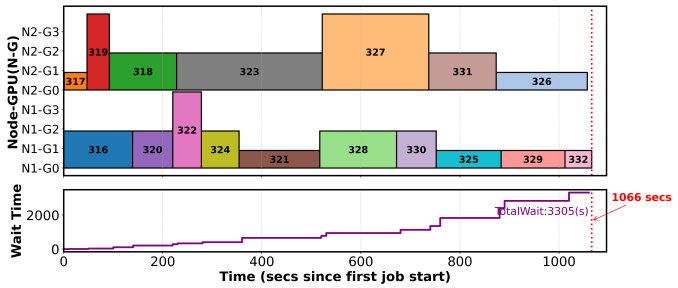


Figure 2: *Jobs scheduled with Priority shuffling (per-job-per-node allocation)*

Empirical Gaps in De Facto Production Scheduling. In this section, we examine current production scheduling to identify remaining improvement opportunities, focusing on Slurm which is the de facto foundation for modern GPU clusters. While prior schedulers employ predictive or profiling-based strategies, we rethink how production systems can operate under profiling-free, dynamically evolving conditions. To this end, we empirically analyze where even this mature baseline leaves room for adaptive improvement through controlled experiments on a Slurm-based cluster. Our goal is to observe how its multifactor priority plugin and allocation mechanisms perform under realistic DL workloads. We deployed Slurm (21.08.5) on two P100 nodes (4 GPUs each), configured with *SchedulerType=sched/backfill*, *PriorityType=priority/multifactor*, and *SelectType=select/cons_tres*. We submitted a mix of DL fine-tuning and inference workloads, including LLM jobs requesting single-GPU, multi-GPU, and multi-node execution.

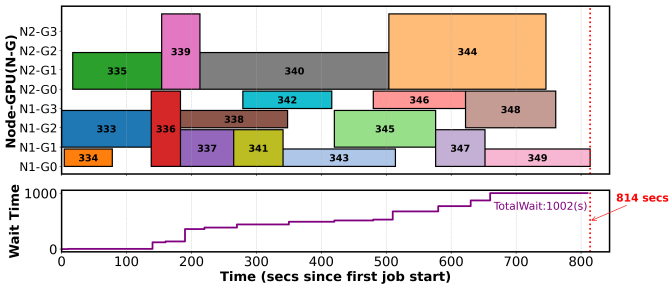


Figure 3: *Slurm's Multi-Factor Priority scheduling (packing allocation)*

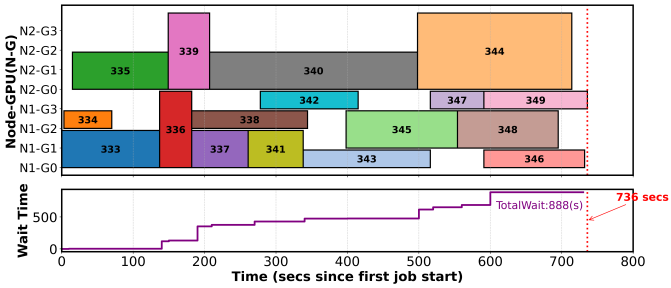


Figure 4: *Jobs scheduled with Priority shuffling (packing allocation)*

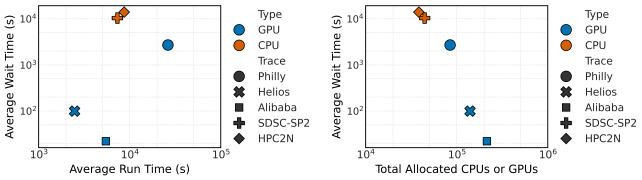
Fig. 1 and Fig. 3 illustrate Slurm's scheduling behavior under two configurations. In Fig. 1, Slurm operated in its default per-job-per-node mode with *OverSubscribe=No*, which prevents multiple jobs from sharing a node and causes jobs to be distributed across separate nodes, leaving several GPUs idle. As a result, resource utilization decreased and cumulative waiting time rose sharply. When *OverSubscribe=Yes* was enabled (Fig. 3), analogous to a gang-scheduling scenario, Slurm packed multiple jobs per node, improving GPU utilization but introducing intra-node contention that could degrade performance. These observations highlight the limited flexibility of existing policies in balancing the classical spread–versus–pack trade-off. We next examined job prioritization, determined in Slurm by its multi-factor priority plugin. To test whether small priority adjustments can yield benefits, we smartly altered job priorities. In the per-job-per-GPU case (Fig. 2), swapping the priorities of Jobs 327 and 328, and in the packed case (Fig. 4), raising Job 348 above 346 and 347, both led to noticeable reductions in cumulative waiting time and makespan. The dotted red lines (JCT) and purple regions (waiting time) show that even minor, context-driven priority changes can produce measurable gains.

These experiments reveal two key gaps in current scheduler behavior: (1) the absence of dynamic, context-aware priority adjustment, and (2) limited flexibility in deciding when to isolate or pack jobs. While Slurm's heuristics are robust and general-purpose, they lack the adaptive intelligence to exploit small but high-impact scheduling opportunities, especially at scale. Recognizing and addressing these gaps at runtime could yield substantial performance gains. Our findings highlight opportunities to introduce context-aware control into an otherwise mature field of GPU scheduling, enabling systems like Slurm to adapt intelligently using runtime feedback instead of per-job profiles. This naturally raises the question of what mechanism can learn and react to system feedback quickly enough to guide scheduling decisions in real time.

Investigating Learning-Driven Scheduling. Building on our observation in prior subsection, we next explore whether learning-based mechanisms can enable the level of adaptivity required for profiling-free GPU scheduling. RL is a natural candidate because it can learn directly from runtime feedback and optimize long-term performance. RL has already proved successful in CPU scheduling [22, 63, 64]. This prior success reveals an opportunity, however, GPU clusters present fundamentally different scheduling challenges, making it non-trivial to apply existing RL policies directly.

Table 2: *Summary of CPU and GPU traces comparing wait time, run time, job arrival rate, and total allocated resources (CPUs or GPUs).*

Trace	Type	Jobs	Avg Wait (s)	Avg Run (s)	Arrival Rate (jobs/s)	Total Alloc (CPUs/GPUs)
SDSC-SP2	CPU	2887	10403.6	7319.6	0.001123	44416
HPC2N	CPU	1768	13985.4	8699.6	0.000868	38403
Philly	GPU	60k	2703.3	26299.2	0.022333	84602
Helios	GPU	85k	99.3	2481.4	0.032919	139996
Alibaba	GPU	200k	22.1	5466.3	0.077136	214372



(a) *Average run time vs. average wait time.* **(b)** *Allocated resources vs. wait time.*

Figure 5: *Comparison of CPU and GPU workload traces*

To empirically validate these differences, we analyzed two CPU traces (SDSC-SP2, HPC2N) [26] and three GPU traces (Philly, Helios, Alibaba). Table 2 reports average wait time, run time, arrival rate, and total allocated resources, normalized over one month. Fig. 5 provides a complementary view, comparing (a) run time vs. wait time and (b) allocated resources vs. wait time. The results reveal distinct scheduling dynamics: GPU traces exhibit much higher job counts, faster arrivals, and greater aggregate resource demand.

The run–wait relationship also diverges where CPU jobs experience long waits even for short runtimes, whereas GPU workloads vary by cluster (Philly: long runs with moderate waits; Helios and Alibaba: short runs with minimal waiting). These patterns highlight different queuing and contention behaviors in multi-GPU, multi-tenant environments. Our experiments further confirmed that CPU-trained RL models failed to converge on heterogeneous GPU clusters. This shift fundamentally redefines the learning problem by changing what RL predicts, how rewards evolve, and which signals matter. Resource needs in CPU scheduling are uniform, however, in GPU clusters, scheduling extends to multi-dimensional allocation across GPUs, CPUs, memory, and interconnects, as well as the spread–versus–pack trade-off. To address this, RLTune employs RL for dynamic prioritization while delegating multi-resource allocation to a complementary solver. Even with this reformulation, a key question remains: can it remain effective under the bursty, non-stationary conditions of real GPU workloads?

Workload Variations and Scheduling Stability. The effectiveness of any scheduler depends on how well it adapts to evolving job arrivals and queue dynamics. To examine this, we analyzed scheduling trajectories, i.e., the temporal sequences of job arrivals, queue responses, and cumulative waiting times, from two large GPU traces, Philly and Helios. For 100 consecutive batches of 256 jobs each, we measured (1) the number of jobs per batch exceeding the global median wait time and (2) the total cumulative wait within the same window. As shown in Fig. 6, these trajectories vary sharply: some batches remain nearly flat with few jobs waiting and minimal cumulative wait time, while others exhibit severe congestion where most jobs wait, accumulating hundreds of hours of total delay. Such non-stationary behavior reveals that workload pressure in GPU clusters is highly bursty and unpredictable. Therefore, Workload characterization alone is insufficient.

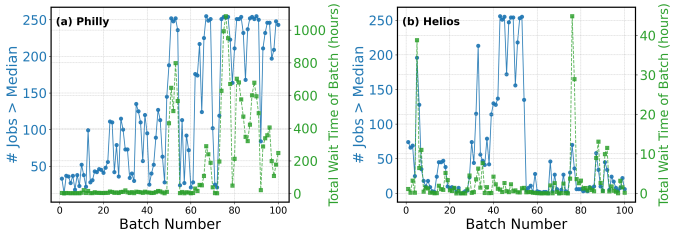


Figure 6: *Batch-wise Analysis: Jobs > Median Wait vs Total Wait Time*

Such variability directly affects a learning-based scheduler's ability to generalize. A policy that performs well during steady phases may fail under sudden congestion or resource fragmentation. For instance, if a scheduler encounters consecutive trajectories where most jobs experience low waiting times, it may appear effective even when its decisions have little real impact and overfitting to easy scenarios and underperforming once the system becomes bursty or imbalanced. Reinforcement learning, however, can leverage this variability when guided by timely feedback to learn stable trade-offs across changing workload conditions. These insights motivate our subsequent design, where we encode workload dynamics into the reward formulation to sustain performance under bursty, dynamic, and unpredictable environments.

Toward a Feasible Hybrid Scheduling Framework. To translate insights from the preceding analysis into practice, we must verify whether dynamic prioritization and adaptive allocation remain feasible at scale under real system constraints. In real deployments, GPU scheduling becomes a multi-dimensional decision problem: when multiple jobs are colocated on a node, GPUs alone are not the bottleneck but CPU, memory, and intra-node bandwidth can introduce interference and affect runtime. In our experiments, Fig. 3 and Fig. 4, we adopt a GPU proportionate CPU allocation strategy and a relaxed memory policy, allowing dynamic use up to a safe threshold. While adequate for feasibility testing, this approach reveals the need for fine-grained modeling of CPU and memory coupling in job-to-GPU mapping. Solver-based, look-ahead allocation can address this challenge by anticipating contention and jointly optimizing GPU, CPU, and memory placements. This motivates extending the scheduling formulation through a Mixed-Integer Linear Programming (MILP)-based allocation framework.

Given this complexity, reinforcement learning remains well suited to control the priority function, dynamically adjusting to workload and system feedback. However, RL alone cannot guarantee global resource-level optimality across dimensions. We therefore adopt a hybrid strategy that couples RL-based dynamic prioritization with MILP-based multi-resource allocation. This separation allows RL's adaptability to complement MILP's stability, improving interpretability, reducing RL training complexity, and ensuring consistent performance across heterogeneous clusters. These insights motivate RLTune 's design, which we describe in the next section.

3 Design

3.1 RLTune System Overview and Job Lifecycle

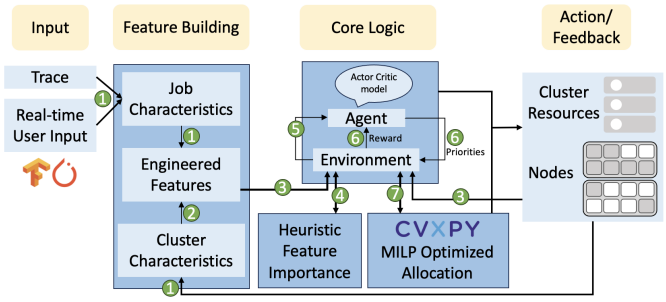


Figure 7: *System Overview and life cycle of Job in RLTune*

We integrate RL-based prioritization with resource-aware placement in RLTune, a hybrid learning-and-optimization scheduler designed for heterogeneous GPU clusters. Fig. 7 presents an overview of RLTune, highlighting its components and the job life-cycle under its operation. The system is organized into three key modules: Feature Building, Core Logic (RL+MILP), and Action/Feedback Network. The Feature Building module extracts job- and cluster-level characteristics, while Feature Sampling dynamically selects the most informative features at runtime based on the current cluster state. The RL agent, implemented using an Actor–Critic architecture with PPO, assigns job priorities adaptively according to real-time

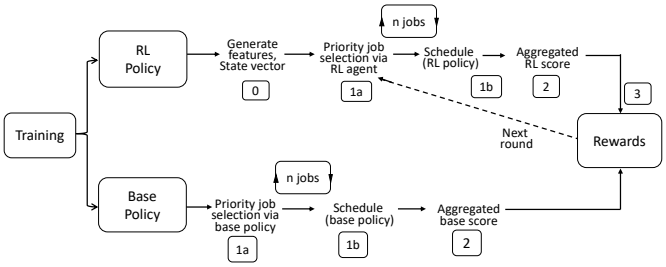


Figure 8: *Training pipeline a in simulated environment for one batch*

cluster conditions. The Allocation Optimization module, formulated as a Mixed-Integer Linear Program (MILP), then determines the most efficient job-to-node mapping given available resources. RLTune operates in two phases: training and evaluation. We describe the step-by-step workflow of both phases and the contribution of each system component in the following sections. We use (①) to denote steps corresponding to those in Fig. 7.

3.1.1 ***Training Phase and Workflow.*** Training an RL agent requires extensive iterations and large volumes of data; hence, we conduct training in an RL environment that mimics a Slurm simulator using large trace inputs. We allocate 90% of the trace data for training and perform periodic evaluations after every N batches (N ranges from 100–1000). The remaining 10% of the trace data is held as unseen data for evaluation.

Fig. 8 illustrates the step-by-step workflow of how jobs traverse each component during training. We train the RL agent in batches of 256 jobs, with one epoch consisting of 100 such batches. Each batch is processed through two pipelines: the base policy pipeline and the RL policy pipeline. Following an application-agnostic approach, the Feature Building Module (FBM) (see Fig.7) in both pipelines scans only visible job features such as job ID, submit time, requested resources, etc.(①). Simultaneously, the FBM scans the cluster state to extract system-level characteristics (①). In Fig. 8, step 0 of the RL pipeline combines these scanned features to construct comprehensive engineered features for each incoming job (②), explained more in § 3.2. Core logic component handles steps 1a, 1b of RL pipeline which differ from 1a, 1b of base pipeline. The core logic includes the RL environment, RL agent and feature sampling module. RL environment continuously monitor arrived jobs and adds them to *job_queue* of both pipelines. The base pipeline uses one of several scheduling policies (e.g., FCFS, SJF, WFP, UNI-CEP, F1, QSSF). Each policy's priority function is derived from one or more job features (e.g., Submit Time (ST), Requested or Run Time (RT), Wait Time (WT), Requested Resources (N)). During execution of 1a, 1b, top priority job from *job_queue* is selected for scheduling, depending on the selected base policy's priority function. In the RL pipeline, the environment takes the feature set from the FBM, applies feature sampling to select a fixed number of important features for each job, and aggregates them across all jobs in the *job_queue* to form the state matrix S_t (③),(④); details are in 3.2. The state matrix S_t is forwarded to the RL agent as input (⑤). The agent outputs a corresponding priority vector A_t (⑥), ranking jobs in real time. The top-K jobs from this ranked list are forwarded to the MILP-based Allocation Optimizer, which evaluates multi-dimensional, look-ahead strategies to select the optimal job-to-node mapping under current GPU, CPU, and memory constraints(⑦). The simulator tracks resource availability, moves jobs from *job_queue* to *running_queue*, and maintains allocation and release records. Upon successful scheduling, a score (base or RL) is computed using a chosen performance metric such as wait time, completion time, bounded slowdown, or resource utilization representing the base score in the base pipeline and the RL score in the RL pipeline. This completes the execution of step 1b in both pipelines. In step 2 of both base and RL pipelines, individual job scores are aggregated to compute the Aggregated Base Score (ABS) and the Aggregated RL Score (ARS) separately. Next, we calculate reward by subtracting ARS from ABS 3. Details of the reward function are provided in § 3.2. During the next iteration in the RL pipeline, the rewards are fed back to reinforce or adjust the agent's actions(⑥). Each batch completes a full feedback loop, and an epoch consists of 100 such batches. Training typically spans epochs until the policy converges.

3.1.2 ***Evaluation Phase and Workflow.*** During the evaluation phase, the base and RL pipelines run independently to compare performance between the baseline policy and RLTune. Evaluations are conducted either in simulation or on a live Slurm deployment. In RL pipeline, steps 0,1a,1b,2 mirror those of the training phase, reusing the same Feature Building and Feature Sampling modules to construct the state matrix S_t. The RL agent then produces a ranked list of job priorities based on S_t, and the top-K prioritized jobs are forwarded to the MILP optimizer for job-to-node placement under current cluster constraints. In real-time Slurm evaluation, the job queue is scanned every minute to generate S_t, capturing both waiting and newly arrived jobs. The RL agent updates job priorities, which are applied directly to Slurm using the *scontrol –priority=* command and allocation flexibility is managed through Slurm's *–oversubscribe* flag as per the solver's guidance. This ensures that updated priority and allocation decisions are refreshed for submitted jobs before they start running, avoiding any data movement across GPUs, nodes, or storage systems. During evaluation, the aggregated score computed as the sum of individual job score (e.g., per-job wait time) directly represents batch-level performance (e.g., total or average wait time). The batch size can be tuned based on the workload arrival rate. RLTune also handles SLA-bound or high-priority jobs that cannot tolerate its operational overhead through the baseline scheduler, ensuring fairness and compliance.

3.2 System Components

Feature Building and Feature Sampling for Importance. To help the RL agent make effective scheduling decisions, we design a feature-building module that captures both job and cluster characteristics. This structured feature set enables the agent to learn key patterns and dependencies in the scheduling environment. For every incoming job, the module constructs a set of engineered features from runtime attributes, resource demands, and cluster status indicators that collectively describe the current scheduling context. The complete list of features is summarized in Table 3.

The primary features such as requested GPUs, submit time, and job duration are obtained directly from trace data during training or scanned from the job in real time. Cluster features are extracted from the current system state, including the number of free and

Table 3: *Feature Categories and Corresponding Features.*

Feature Category	Features
Visible Job Features	job ID, user info, requested GPUs, virtual cluster, GPU type, requested time, submit time, req_CPU, req_mem.
Cluster Characteristics	free nodes, can_schedule_now, num_ways_to_schedule.
Engineered Features	Demand-Supply Ratio (DSR), Job Size, Job Urgency Score, Future Availability, Cluster Fragmentation Factor (CFF).

used GPUs or nodes. Computed features like can_schedule_now and num_ways_to_schedule capture job feasibility under current resource constraints. From primary features, we derive engineered features by combining or transforming base features to express more actionable scheduling insights. We first describe three complex engineered features through equations, followed by the remaining ones. The first engineered feature, Demand–Supply Ratio (DSR), is defined as follows:

$$\text{demand\_supply\_ratio} = \left(\frac{[\text{req. gpus}]_{\text{type}}}{[\text{free gpus}]_{\text{type}}} \right)_{\text{norm}} \quad (1)$$

Demand Supply Ratio captures scaled measure of the relationship between the demand for GPUs of a specific type and the current availability of GPUs of the same type. This feature provides insight for keeping balance between cluster load and resource contention. Second feature is Future Availability defined as follow:

$$\text{future\_avail} = \left[\sum_{\text{type}} \left(\text{free\_gpus} - (j_{\text{curr}})_{\text{req\_gpus}} - \sum_{j \in \text{nodes}} (j_i)_{\text{req\_gpus}} \right) \right]_{\text{norm}} \quad (2)$$

The Future Availability estimates the expected number of free GPUs in the cluster after accounting for the GPUs requested by the current job and those already allocated to other jobs on the same nodes. By incorporating this forward-looking estimate, the RL agent gains a predictive view of resource usage if the current job is scheduled. Next Cluster Fragmentation Factor (CFF) defined as follow:

$$\text{CFF} = \left(1 - \frac{\sum_{\text{nodes}} (\text{free\_gpus})^2}{\text{total\_free\_gpus}} \right)_{\text{norm}} \quad (3)$$

CFF measures how evenly free GPUs are distributed across the cluster. A higher CFF indicates greater fragmentation, where GPUs are scattered across multiple nodes, making it harder to schedule large multi-GPU jobs on the same node.

Not all features listed in Table 3 are directly used for state construction. In total, we maintain 17 features across the system: some are used purely for metadata tracking (e.g., Job Id, user info, virtual cluster), and such as Req_CPU and Req_mem are used only when explicitly provided, otherwise inferred from GPU share. For forming the state vector input to the RL agent, we employ heuristic-based feature sampling to select a subset of eight key features from the complete set. This selection avoids redundancy by choosing either raw or engineered variants based on situational relevance. For instance, when CFF is high, the sampler select and weights feature `Job_size = normalized(requested_gpus × requested_time)` to favor short jobs that can efficiently fill fragmented nodes. Under low-fragmentation conditions, the job urgency score is emphasized to boost the priority of aged jobs while avoiding penalties for large or long-running jobs. When a job can be scheduled in multiple valid ways, importance is assigned to the num_ways_to_schedule feature. This increases the likelihood that such flexible jobs receive higher priority, allowing the scheduler to exploit placement opportunities and improve overall cluster utilization.

We selected 8 features as a balanced trade-off between model simplicity and expressive capacity. A preliminary sensitivity analysis comparing feature sets of sizes 7, 8, and 9 showed that 8 features consistently delivered stable and efficient learning, with no significant benefit from including more. This compact yet expressive representation reduces runtime overhead while preserving enough information to support robust policy learning.

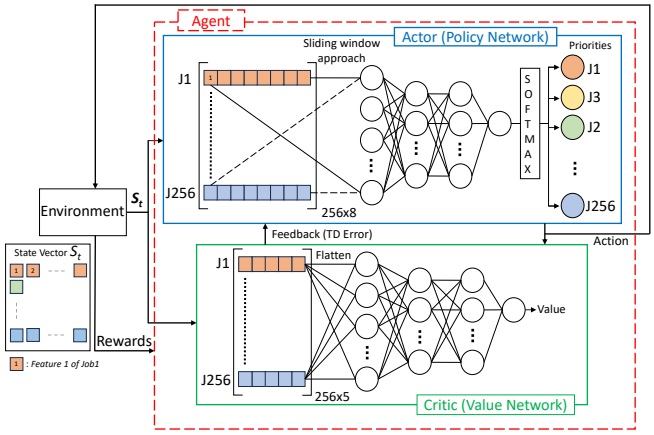


Figure 9: *Actor-Critic Architecture*

Actor Critic. The goal of RLTune is to enable dynamic prioritization by learning and adapting scheduling decisions to changing cluster availability. To achieve this, we employ the Proximal Policy Optimization (PPO) algorithm [55] with an Actor–Critic framework. The RL agent, modeled as an Actor–Critic network, consists of an actor that assigns priorities to queued jobs and a critic that evaluates the quality of these decisions to guide policy updates.

The environment provides a State matrix S_t, composed of a feature vector that we then split into Observation vector (OV) of fixed size ($batchsize \times 8 job features$), and a critic vector (CV) of size ($batchsize \times 5 job features$). The OV contains normalized values of the eight key features selected through heuristic-based sampling. The CV includes five core features such as submit time, run time, and can_schedule_now. CV estimate long-term value and stability of scheduling actions. To bound inference complexity and maintain scalability, RLTune evaluates at most `MAX_QUEUE_SIZE = 256` jobs per decision trajectory. The RL agent operates on fixed-size OV and CV embeddings (256×8 and 256×5), applying zero-padding when fewer jobs are present. This design ensures that both the state and action spaces remain constant, allowing the Actor–Critic model stable maintain inference latency across diverse workloads with varying queue lengths.

Algorithm 1 Dynamic Resource Allocation using CVXPY and Mixed-Integer Linear Programming

Require: cluster_status, gpus_per_node,cpu_per_gpu,mem_per_gpu
Ensure: Decision: way1 (spreading) vs. way2 (packing)

```
// Binary variable: 0 for way1, 1 for way2
x ← Variable(boolean=True)
dims ← (len(cluster_status), gpus_per_node)
// Occupancy matrix
CJO ← Variable(dims, boolean=True)
constraints ← []
if way1 is list or way2 is list then
    for all (way, val) in [(way1, 1 − x), (way2, x)] do
        for all (node, gpu_count) in way do
            if node in valid_nodes then
                for g ← 1 to gpu_count do
                    idx ← valid_nodes[node]
                    constraint ← CJO[idx][g] == val
                    constraints.append(constraint)
for all (i, (node_num, available_gpus)) in enumerate(cluster_status) do
    for g ← 1 to min(available_gpus, gpus_per_node) do
        total_occupancy ← CJO[i][g]
        constraints.append(tot_occup ≤ avail_gpus)
        constraints.append(∑ CJO[i]× cpu_per_gpu ≤ avail_cpus)
        constraints.append(∑ CJO[i]× mem_per_gpu ≤ avail_mem)
// Maximize GPU occupancy
objective ← Maximize(sum(CJO))
prob ← Problem(objective, constraints)
prob.solve(solver=GLPK_MI, verbose=False)
if x.value < 0.5 then
    selected_way ← way1
else
    selected_way ← way2
        return selected_way
```

Fig. 9 illustrates the architecture of the Actor–Critic networks along with their inputs and outputs. Both networks are implemented using TensorFlow [10]. The actor network is a three-layer MLP [57] that receives the Observation Vector (OV) as input and outputs an action vector representing job priorities. Each job's feature set is processed in a sliding-window fashion, enabling the actor to evaluate jobs individually while maintaining global context. After passing the OV through the network and a softmax layer, the resulting normalized priority scores are returned to the environment, which schedules jobs and computes rewards. The critic network, also a three-layer MLP, processes the Critic Vector (CV). The CV is flattened to include all jobs simultaneously, allowing the critic to estimate the expected cumulative reward for the current job sequence under the actor's policy. The critic thus provides a scalar value representing the quality of the actor's decisions. The actor and critic are trained jointly: after each scheduling decision, the environment returns a reward signal. This reward is used to update the critic's value estimation and, through backpropagation, to refine the actor's policy. In this way, the critic guides the actor's learning, enabling continuous improvement of scheduling decisions across varying workload conditions.

Reward Function. The "Score", a job-level reward is calculated based on the targeted optimization goal. For example, if the goal is to minimize wait time, the score is the job's wait time (scheduled – submitted), and the reward is defined as reward = $-wait\_time$. So the RL agent maximizes reward by reducing wait time. In our design, individual job scores are aggregated because metrics such as average waiting time or bounded slowdown can only be computed after all jobs in a batch are scheduled. Hence, job-level scores are not directly fed to the RL agent. Once the final action in the batch is produced, we compute the performance gap between the baseline (without RLTune) and RLTune 's scheduling outcomes. The normalized difference serves as the reward signal. Feeding rewards as normalized performance gaps reduces variance from sudden fluctuations in input features [63, 64]. This formulation also prevents overfitting to easy scenarios as discussed in Section 2, ensuring that improvements reflect genuine scheduling gains even under skewed workload patterns and congested queue conditions.

Allocation Optimization Module. The Dynamic Resource Allocation component optimizes job-to-node assignments by selecting the best allocation strategy in real time. When multiple placement options exist (e.g., pack, spread, or hierarchical), this module evaluates the impact of each choice on future cluster performance. As discussed in Sec. 2, spreading and packing represent key trade-offs between utilization and contention. Conventional schedulers such as Slurm rely on static heuristics and cannot adapt between these strategies under changing conditions. To overcome this limitation, we model job allocation as a Mixed-Integer Linear Program (MILP) implemented using the GLPK_MI solver [47] within the CVXPY framework [21]. In each iteration, the RL agent outputs a real-time job priority vector. The top-K jobs are passed to the MILP-based optimizer, which selects the optimal job-to-node mapping under current GPU, CPU, and memory constraints. Using look-ahead strategies, it dynamically chooses between spreading and packing to improve long-term cluster performance.

Algorithm 1 defines a binary variable x selects between *way1* (spreading) and *way2* (packing). The occupancy matrix CJO represents GPU usage per node, with constraints enforcing GPU, CPU, and memory limits. The objective maximizes total GPU occupancy while satisfying per-node resource constraints. After solving, the value of x determines the selected strategy. As MILP operates on the top-K jobs prioritized by the RL agent, the formulation considers both current and future job requirements: the top-K high-priority upcoming jobs in the queue are monitored to explicitly model their resource constraints, start times, and potential usage across multiple time slots. K is a tunable parameter that can be adjusted according to job burst intensity. The Feature Sampling module serves as a bridge between the RL and MILP components, aligning their decisions to complement each other. Feature sampling ensures coordination by emphasizing cues such as future availability and jobs with multiple allocation options. This design allows the RL agent to promote jobs where the optimizer can be most effective, enabling both components to work in tandem toward higher utilization and lower waiting times.

Table 4: *Trace Summary.*

Trace Name	Time	Total Jobs	GPUs, Number of Nodes	GPU type	Users	Run time (avg, max)	Scheduler	Scheduling Algorithm	Network (same-rack)	Network (cross-rack)
Philly'17	Oct'17-Dec'17 (75 days)	96260	2490, 552	P100(2-GPU), P100(8-GPU)	319	28329 sec, 60 days	Apache YARN (FIFO)	Locality-aware	100-Gbps(Infini-Band)	Ethernet
Alibaba'20	July'20-Aug'20 (60 days)	1.2 million	6.5K, 1.8K	T4, Misc, P100, V100(16), V100(32)	1242	4456 sec, 30 days	Fuxi (FIFO)	GPU Sharing, reserving-packing	V100/V100M32: NVlink, rest all: PCIe	Not Allowed
Helios'21	April'20-Sep'20	1,753K	2096, 262	P100(8-GPU), V100(8-GPU)	277	6652 sec, 50 days	Slurm (FIFO)	Quasi-Shortest-Service-First	Intra-Node:PCIe(Pascal), NVLink(Volta); Inter-node: Infiniband	Not Allowed

Table 5: *Scheduling Policies. Characteristics: submit time (ST), Requested or run time (RT), wait time (WT), Requested Resources (N), and throughput (T).*

Scheduling Policy	Characteristics	Equation or Strategy
FCFS	ST	St
SJF	RT	Rt
WFP3[56]	ST, RT, WT, N	$-\left(\frac{wt}{rt}\right)^3 \times nt$
UNICEP[56]	ST, RT, WT, N	$-\frac{wt}{\log_2(nt)\times rt}$
F1[17]	ST, RT, N	$\log_{10}(rt) \times nt + 870 \times \log_{10}(st)$

4 Experimental Setup

In this section, we describe the simulated environment, workload traces, scheduling policies, and performance metrics used in our experiments and analysis.

4.1 Scheduling Environment

Reinforcement learning requires large amounts of data and interaction, making real-time training impractical. We therefore use a trace-driven simulated environment that enables repeated iterations. We implemented this environment using OpenAI Spinning Up [11] and adapted it from RLScheduler [63], extensively modifying it to support heterogeneous GPUs and fundamentally redefine the learning problem to align with our resource allocation and prioritization logic. The environment loads a job trace and simulates scheduling from an idle cluster. Whenever a job arrives or completes, it selects the next action using our proposed mechanisms. If resources are insufficient, it applies backfilling by placing smaller jobs without delaying higher-priority ones or disrupting running jobs. During training, RLTune uses ground-truth runtimes from traces, consistent with prior RL schedulers [30, 63], to provide accurate reward signals and ensure stable learning. During evaluation, only user-provided (potentially noisy) runtime estimates are used, reflecting realistic conditions. Despite this uncertainty, RLTune performs robustly across all baselines. Improved runtime predictors could further enhance overall performance but are orthogonal to the core profiling-free scheduling contributions of this work. Evaluation does not support elastic resource modification (GPU type or count) or preemption within the scope of this work.

4.2 Workloads/Traces

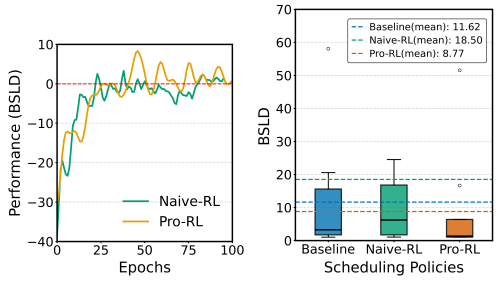
We evaluate RLTune on heterogeneous clusters using real production deep learning (DL) traces: Philly [34], Helios [30], and Alibaba PAI [59]. Unlike prior approaches [33, 49, 52] that replay a small subset of pre-profiled jobs, we use the full traces to drive simulation. Although systems such as [33, 49, 52] are strong baselines, direct comparison is difficult since their evaluations rely on mapping trace entries to pre-profiled workloads. Their mapped trace version exclude job identifiers and retain only limited metadata (timestamps, durations, GPU counts, GPU time), making it infeasible to map them back to the original traces. During RL training, each trace is simulated on a representative cluster slice chosen to maintain realistic contention based on job arrivals and runtimes. For example, in Helios, we model five virtual clusters (VC1–VC5) with 16, 12, 10, 8, and 8 nodes (each with 8 GPUs), following prior work [33, 41, 61]. Table 4 summarizes key trace characteristics, highlighting workload heterogeneity and diversity. Evaluating on a single trace limits generality [13, 50]. Earlier frameworks (e.g., [28, 46, 49, 62, 67]) relied solely on Philly, then the only public dataset. In contrast, we employ multiple diverse traces, producing policies that generalize across heterogeneous clusters.

4.3 Scheduling Policies

We compare RLTune against several scheduling policies, including FIFO, SJF, WFP3, UNICEP [56], F1 [17], Slurm Multifactor Priority. These policies rely on different job characteristics, summarized in Table 5. FIFO schedules jobs by submission order, while SJF favors shorter runtimes. WFP3 and UNICEP [56] combine multiple factors, prioritizing jobs with short runtimes, small resource requests, or long waits, often tuned with expert knowledge. The F1 scheduler [17] is ML-based, using brute-force simulation and non-linear regression to minimize target metrics.

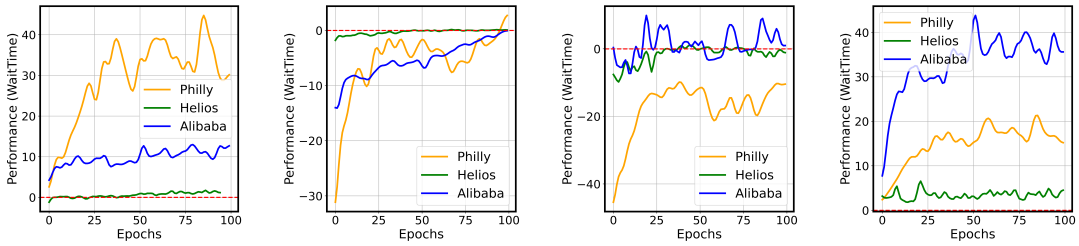
4.4 Performance Metrics

We evaluate four key performance metrics: (1)**Wait Time**: The time between a job's submission and its start. (2) **Job Completion Time (JCT)**: The mean duration from submission to completion, equal to wait time plus runtime averaged across all jobs. (3) **Average bounded slowdown (bsld)**: Measures slowdown relative to runtime, balancing penalties for both long-waiting and long-running jobs. Introduced in [25]. (4) **Resource utilization**: The mean percentage of allocated GPUs, normalized by the total GPUs in the cluster over time.



(a) *Training* **(b)** *Evaluation*

Figure 10: *Comparison between performance of naive-RL and pro-RL*



(a) *FIFO* **(b)** *SJF* **(c)** *F1* **(d)** *WFP3*

Figure 11: *Training curves of RLTune on three real-world traces for the performance metric Wait Time, using four different base policies. Subfigures highlight the respective base policy, and the y-axis shows the performance difference between RLTune and the corresponding base policy.*

5 Evaluation

We present evaluation of RLTune under various scenarios.

5.1 Design Choice Justification.

This section evaluates the key design choices in RLTune, focusing on the impact of feature construction, sampling strategy, and CVXPY-based allocation. We first analyze how these components affect learning efficiency and overall scheduling performance.

Naive-RLTune vs Pro-RLTune. We evaluate two variants of RLTune: naive-RLTune and pro-RLTune. In naive-RLTune, raw trace features are fed directly to the RL agent without feature construction or allocation optimization. Since MILP-based allocation is disabled, job placement follows Slurm's default (OverSubscribe=No). Pro-RLTune adds two enhancements, feature sampling and MILP-based allocation optimization. The engineered features capture key aspects of cluster state, job queue, and job characteristics, as described in § 3.2, while the solver ensures resource-aware placement. Figure 10a shows their training curves using Slurm as a baseline on the Philly trace, and Figure 10b presents the evaluation results. Pro-RLTune achieves a 52.59% improvement in BSLD over naive-RLTune. This result highlights that directly feeding raw features to RL and expecting it to learn everything performs poorly in complex GPU environments. However, structured feature design and solver-guided allocation significantly enhance learning efficiency and scheduling performance. In all subsequent experiments, we refer to pro-RLTune simply as RLTune.

5.2 Generalization Across Diverse Job Traces and Base Scheduling Policies

We evaluate RLTune on three heterogeneous traces, Philly, Helios, and Alibaba, each representing a distinct cluster configuration with unique workload characteristics (Tables 2, 4). This diversity ensures that RLTune is tested across varied workload patterns and cluster conditions. In each training epoch, the RL agent processes 100 batches of 256 jobs, updating the actor–critic policy and value networks after every batch. The agent is trained to enhance performance over four base scheduling policies: FIFO, SJF, F1, and WFP3. Figure 11 shows the training curves for 100 epochs using wait time as the performance metric. Each curve corresponds to a specific policy, with results from the three traces normalized to the range −50 to +50 for consistent scaling and compact visualization.

Figure 11 illustrates the training progress of RLTune, showing the normalized difference between the RL agent's performance and each corresponding base scheduling policy (FIFO, SJF, F1, and WFP3) for the wait time metric. A rising curve indicates that the RL agent is learning to reduce average wait time more effectively than the baseline policy. The eventual flattening of the curve signifies that training reaches convergence and the policy stabilizes. Across traces, the vertical scale differs because of workload diversity.

After training, we evaluate RLTune 's performance against the four base scheduling policies as shown in Fig. 12. For each workload trace, jobs are scheduled using both the original policy and its RL-enabled counterpart (e.g., FIFO vs. RL-FIFO). Each experiment runs ten times with random sequences of 1,024 jobs, and the average wait time is reported. Overall, RLTune consistently lowers average wait time relative to its base policies. For instance, in the Philly trace under FIFO, RL-FIFO achieves an 87.5% reduction in wait time, demonstrating a substantial gain in scheduling efficiency. In the Alibaba trace under SJF, the results show only a small change, reflecting limited room for improvement in a workload already suited to SJF. These results confirm that RLTune effectively adapts across diverse policies and traces, enhancing multiple scheduling strategies under a unified framework.

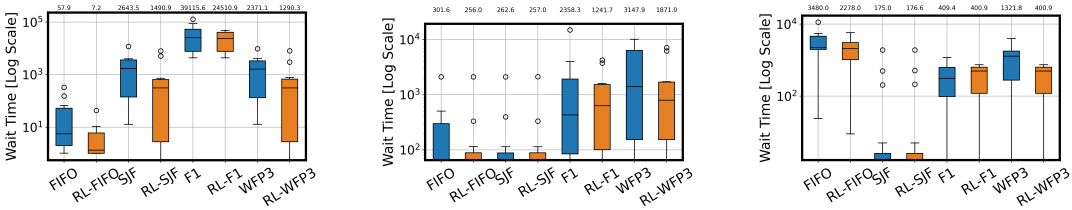
5.3 Performance on Multiple Metrics

This section evaluates RLTune across several metrics, bounded slowdown (BSLD), job completion time (JCT), and GPU utilization. While waiting-time results were discussed earlier, here we focus on the remaining metrics and compare RLTune against three base scheduling policies. Results for additional policies are omitted due to space limits. The training and evaluation setup remains identical to the previous section. Figures 13a and 13b show training curves of RLTune when trained with FIFO and F1 as base policies for the BSLD metric. Figures 13c and 13d present the corresponding results of RLTune for JCT under FIFO and SJF as base policies.

Table 6: *Utilization improvement across policies and traces*

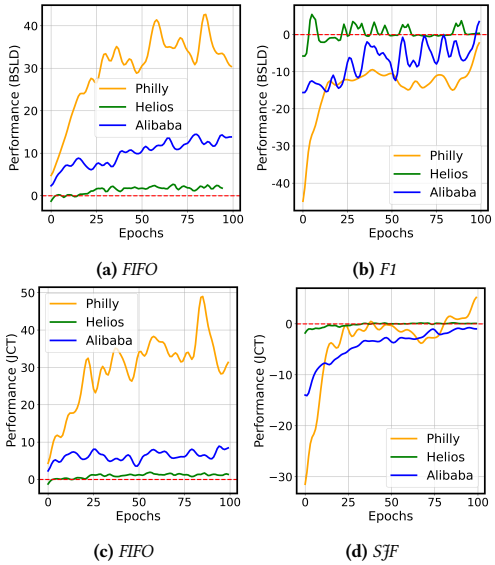
	FIFO	SJF	F1
Philly	4.83%	3.44%	13.62%
Helios	1.00%	19.71%	1.95%
Alibaba	2.40%	11.30%	1.28%

Figures 14 and 15 present evaluation results of RLTune compared to base scheduling policies for the BSLD and JCT metrics across the three traces. As shown in Table 6, RLTune improves cluster utilization across all workloads relative to the baseline policies. Across workloads, RLTune reduces BSLD by at least 5.28% over F1 on Helios and up to 72.32% over SJF on Alibaba. For JCT, it



(a) *Philly* **(b)** *Helios* **(c)** *Alibaba*

Figure 12: *Waiting time distribution across base policies and RL policy for three traces. Mean values of the performance metric are annotated on top of each graph.*



(a) *FIFO* **(b)** *F1*

(c) *FIFO* **(d)** *SJF*

Figure 13: *Training curves of RLTune for BSLD and JCT across three traces, trained using two base scheduling policies.*

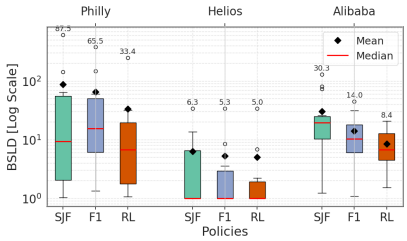


Figure 14: *BSLD distribution across different base policies and the RL policy for three traces. Lower values indicate better performance.*

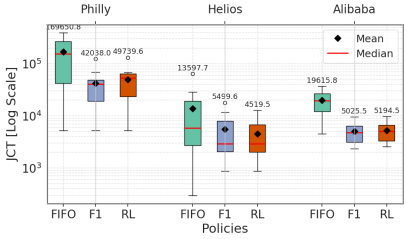


Figure 15: *Job Completion Time distribution across different base policies and the RL policy for three traces. Lower values indicate better performance.*

achieves up to 70.68% improvement over FIFO on Philly, with a modest 1.3% degradation relative to F1 on Alibaba, likely because F1 already attains near-optimal job placements in that trace. RLTune increases resource utilization by 13.62% and 19.71% on Philly and Helios, respectively, relative to the F1 and SJF policies.

The observed trends align with workload characteristics and the nature of baseline policies. Philly's mix of long, multi-GPU jobs allows RLTune to learn more effective prioritization, yielding substantial gains in wait time, BSLD, and utilization. Helios and Alibaba, dominated by short jobs, offer limited headroom since heuristics like SJF and F1 already perform near-optimally. While SJF assumes perfect knowledge of job runtimes and F1 relies on static log-scaled features, both lack adaptability to evolving queue states and heterogeneous resources. In contrast, RLTune learns directly from runtime signals, generalizing across traces and achieving consistent improvements in BSLD and utilization even when JCT gains are modest. These results reflect the complementary nature of the evaluated metrics, where wait time and JCT capture per-job responsiveness, BSLD and utilization reveal system-level efficiency. Nevertheless, RLTune underscores its versatility and effectiveness across multiple performance dimensions.

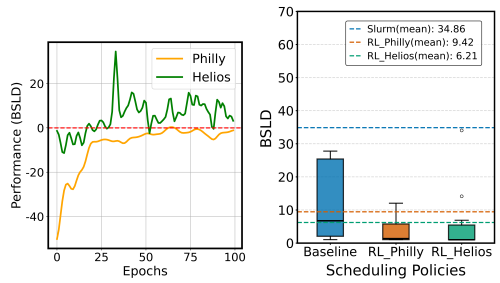
Table 7: *Wait time improvement on the Helios trace using cross-policy models.*

trained on	tested on			
	FIFO	SJF	F1	WFP3
FIFO	14.95%	2.29%	4.73%	17.32%
SJF	14.22%	2.22%	4.70%	15.23%
F1	14.91%	1.66%	4.73%	13.81%
WFP3	3.76%	-4.01%	-4.31%	11.83%

5.3.1 Transfer Learning. Our training setup is not limited to a specific policy or metric; instead, we assess how a model trained under one policy performs when used to make scheduling decisions under another. Table 7 reports percentage improvements in wait time on the Helios trace when each model is trained on one base policy and tested against all others. The results show that RLTune generalizes effectively across policies within the same trace, reducing the need for frequent retraining. Performance degradation is observed when trained on WFP3 and tested on other policies like SJF, and F1. SJF and F1 prioritize short, early jobs, which sometimes aligns with WFP3's fairness model. However, WFP3-trained agents emphasize long waiting jobs, even if they can be large or late, leading to mismatches under SJF and F1 (linear and log-scaled score) and may have reduced performance. Thus, training cost can be reduced by leveraging transfer, showing potential to lower the overhead of training in dynamic environments.

5.4 Slurm in Simulated Environment

Until now, we have examined the effectiveness and generality of RLTune based on various heuristic scheduling policies. The Slurm multifactor priority plugin adjusts job order using a weighted blend of age_factor, fair_share, job_attributes, and partition_factor, QoS_factor. age_factor is waiting time, fairshare_factor by mapping CPU-based fair-share math to GPU. We use the job's requested run time as the job_attribute_factor, and the partition_factor represents the priority assigned to each queue in the system. In our study, we approximate each factor for GPUs and set all weights to 1000 to ensure equal contribution. Figure 16 presents training curves on the Helios and Philly traces, comparing bounded slowdown (BSLD) against Slurm's baseline. RLTune consistently outperforms Slurm, reducing BSLD by 71.54% on Philly and 81.18% on Helios, demonstrating that it can surpass Slurm's multifactor priority mechanism in simulation.



(a) *Training* **(b)** *Evaluation*

Figure 16: *Results on Philly and Helios trace using SLurm as a base scheduler.*

5.5 Comparison against State-of-the-Art Systems

Finally, we compare RLTune with SOTA scheduler Quasi-Shortest-Service-First Scheduler (QSSF) [30]. The QSSF policy uses history-based job priority predictions, and the authors have made these predictions publicly available for the Philly trace. Hence we can compare our system against QSSF scheduler for Philly trace. The results are presented in Table 8. For all four performance metrics, we see significant improvement with RLTune as compared to QSSF. RLTune brings 25% improvement in wait time 3.25× better performance on the BSLD metric.

To verify the robustness of RLTune, we conduct a large-scale experiment using 10,000 consecutively executed jobs, comparing RLTune against QSSF. In this setup, Fig. 17 presents the job completion time (JCT), which better reflects long-term system efficiency. The results show a 48.43% improvement in JCT with RLTune. As noted in Table 2, the Philly trace exhibits high wait and run times, showing the benefits of adaptive scheduling under heavy and long-running workload.

Performance	QSSF	RLTune
Wait Time	3748.14	2830.01
BSLD	28.17	20.11
JCT	35567.97	33199.58
Utilization	4.72	4.97

Table 8: *Comparison of QSSF and RLTune performance (with backfilling).*

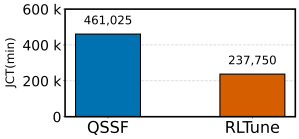


Figure 17: *Comparison of JCT for QSSF and RLTune on 10k jobs.*

Table 9 presents a comparative analysis of multiple scheduling policies across three large-scale GPU traces. We evaluate each policy using five key metrics BSLD, wait time, JCT, GPU utilization, and time which is a scheduling overhead for 10 batches of 256 jobs. The FIFO policy is just for the baseline. We compare RLTune against two most recent state-of-the-art RL-based schedulers: RLScheduler and SchedInspector allowing a direct comparison of performance and execution time. Notably, the original simulated environments used in these papers do not support GPU workloads. To ensure a fair and meaningful comparison, we reimplemented the core RL mechanisms of each scheduler and adapted them to run on GPU traces. Across all three traces, RLTune delivers the best overall performance, achieving the lowest BSLD, wait time, and JCT while maintaining high utilization. On Philly, it reduces BSLD to 232.82 and JCT to 85.7k s, outperforming RLScheduler (491.92) and SchedInspector (1114.52). On Helios, it attains a BSLD of 75.24 with 43.3% lower wait time than RLScheduler, maintaining comparable utilization (4.09) and moderate execution time (102 s). On Alibaba, RLTune achieves BSLD (44.05) and low wait time (8726 s), surpassing all baselines. While RLScheduler shows reasonable performance, its runtime overhead (181 s) is the highest. Overall, RLTune achieves an effective balance between scheduling performance and runtime efficiency across heterogeneous GPU workloads.

5.6 Real Slurm Deployment

To evaluate end-to-end system performance, we conducted an experiment using a heterogeneous Slurm cluster comprising two P100 nodes (each with 4 GPUs), two K80 nodes (each with 2 GPUs), and one M40 node (with 1 GPU). We used *Slurm v21.08.5* configured with *SchedulerType=sched/backfill*, *SelectType=select/cons_tres*, and *PriorityType=priority/multifactor* with equal weights assigned to all priority factors. We generated a synthetic trace of 1,024 ML/DL jobs on a heterogeneous Slurm cluster, consisting primarily of deep learning fine-tuning and inference workloads. These included large language model (LLM) jobs requiring a single GPU, multiple GPUs, or multi-node execution. The dynamically arriving batch was submitted to both Slurm's weight-tuned Multi-factor Priority scheduler and the RLTune-equipped scheduler under identical conditions. RLTune adjusted job priorities at 1-minute intervals and applied a custom allocation mechanism that dynamically toggled the *OverSubscribe=NO* flag.

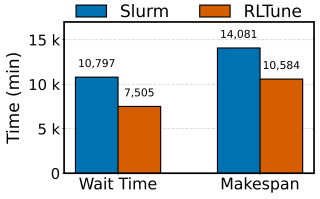


Figure 18: *End-to-end performance of RLTune vs slurm.*

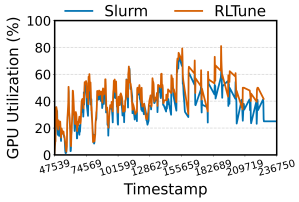


Figure 19: *End-to-end performance: utilization RLTune vs slurm.*

Table 9: *Comparison of scheduling policies across three traces: Philly, Helios, and Alibaba. WT (s) = Wait Time, JCT (s) = Job Completion Time, BSLD = Bounded Slowdown, Utilization (%) = GPU utilization. "Time" denotes total elapsed time in seconds.*

	Philly				Helios				Alibaba				Time
Policy	BSLD	WT	JCT	Util	BSLD	WT	JCT	Util	BSLD	WT	JCT	Util	
FIFO	1298.06	142391.50	169650.84	4.63	161.77	7264.58	10483.16	4.03	88.32	14306.14	19219.08	0.08	86.76
SchedInspector	1114.52	120623.63	150347.88	4.85	152.16	7187.54	10715.42	4.00	89.52	14470.00	19382.94	0.09	96.13
RLScheduler	491.92	86512.85	112022.08	4.64	117.83	6178.47	9211.55	4.01	52.18	9641.81	14554.75	0.06	181.27
RLTune	232.82	34199.93	85761.51	4.85	75.24	4335.87	7541.40	4.09	44.05	8726.45	13639.38	0.09	102.38

The results, summarized in Fig. 18, 19, show that RLTune achieved a makespan of 10,584 mins compared to 14,081 mins for the Multi-factor Priority Slurm baseline (a 24.8% reduction), increased overall GPU utilization by 3.9%, and reduced average wait time by 30.5%.

5.7 Operation Costs

Training RLTune takes between 3–8 hours, depending on the trace length and job diversity. During inference, the system to maintain inference latency of $\sim$ 0.7 ms (including state construction and RL forward pass), and the `CVXPY` solver adds only $\sim$ 0.2 ms, which is acceptable for batch-scheduling workloads. To evaluate scalability, we stress-tested RLTune with up to 10,000 concurrent job arrivals. Decision latency increases sub-linearly from 7.8 s, 9.8 s, 14.3 s, and 22.8 s for queue sizes of 128, 256, 512, and 1024, respectively corresponding to 1.26×–1.59× growth as the queue doubles. The MILP solver is triggered only when multiple placements exist for high-priority jobs (typically $H = 8$–16). Under bursty arrivals, this parameter can be reduced to meet latency budgets.

6 Related Work

Scheduling in Heterogeneous Environments. Scheduling in heterogeneous GPU clusters has been approached in various ways. Systems such as Gavel [49] and Pollux [52] optimize throughput by leveraging job-level profiling to predict performance under different resource allocations, while Sia [33] extends this approach to heterogeneous GPUs through co-adaptive tuning. However, their dependence on pre-profiled workloads and application-specific metadata limits adaptability and fair trace comparison. In contrast, RLTune learns scheduling policies dynamically from real-time job and cluster states, generalizing to unseen workloads without profiling overhead. Other schedulers address orthogonal objectives. Lucid [32] focuses on scheduling interpretability, and Lyra [43] explores elastic scheduling by loaning idle inference GPU servers for elastic training jobs. Shockwave [68] targets job progress fairness using stochastic dynamic programming, explicitly handling temporal variations in job throughput. Distributed LLM serving systems like FairServe [38] and training schedulers like like Optimus [51], Tiresias [28], and Gandiva [61] aim to improve JCT, efficiency, and fairness on heterogeneous setups. Others further consider JCT fairness for training jobs [19, 46, 67], Antman [62] for co-location on homogeneous GPUs, while Allox [41] for CPU-GPU interchangeability. SchedTune [12] incorporates ML-based predictions using historical data. Recent efforts expand to network-, storage-, and elasticity-aware scheduling. CASINNI [53] mitigates communication bottlenecks, Easyscale [44] supports elastic training, FDG [60] reduces GPU fragmentation, and SiloD [66], SHADE [40], and FedCaSe [39] integrate caching with scheduling. Acme [31] characterizes large-scale LLM workloads, motivating adaptive and fault-tolerant scheduling.

Existing schedulers rely on profiling, heuristics, or narrow objectives. RLTune combines RL-based prioritization and MILP-based allocation for profiling-free scheduling across heterogeneous GPUs.
Reinforcement Learning for Task Scheduling. While RLScheduler [63], SchedInspector [64], MARS [15], [45], and DRAS [22] advance RL-based scheduling for CPU workloads, they operate under static or workflow-oriented models. Their learning focuses on CPU-centric environments. In contrast, RLTune targets GPU-intensive ML/DL workloads to extend this line of work toward GPU-aware, multi-resource scheduling, where learning-based prioritization and solver-based optimization shows potential to address fine-grained resource contention and dynamic workload diversity.

7 Conclusion

In this work, we addressed the challenge of scheduling ML/DL workloads on large GPU clusters through RLTune, an application-agnostic RL+MILP-based dynamic scheduling policy. Unlike prediction-based schedulers, RLTune jointly optimizes prioritization and allocation without per-application profiling, adapting to dynamic workloads. Evaluations on real-world traces show up to 81% lower queueing delay, 70.8% shorter JCT, and 20% higher GPU utilization, along with runtime cost savings. Compared with state-of-the-art RL schedulers, RLTune achieves 1.2× faster job completions, up to 35% lower waiting times, and 20% higher utilization across diverse traces. On a heterogeneous Slurm cluster with a 1,024-job synthetic trace, it further reduces makespan by 24.8%, improves utilization by 3.9%, and lowers wait time by 30.5%. Overall, RLTune generalizes across cluster types and objectives, providing a robust and adaptive solution for heterogeneous GPU scheduling. RLTune Github Link [8].

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. Some results were obtained using the Chameleon testbed [37], supported by the NSF. This work is supported in part by NSF grants CSR-2106634, CSR-2312785, National Natural Science Foundation of China, Grant no. 62202382.

References

[1] 2024. 2024 Data Center Trends: A Glimpse into the Future of Rack Density, AI, and Workload Migration. https://tinyurl.com/datacentertrend.
[2] 2024. Announcing A3 supercomputers with NVIDIA H100 GPUs, purpose-built for AI. https://cloud.google.com/blog/products/compute/introducing-a3-supercomputers-with-nvidia-h100-gpus.
[3] 2024. Introducing the AI Research SuperCluster — Meta's cutting-edge AI supercomputer for AI research. https://ai.meta.com/blog/ai-rsc/.
[4] 2024. Microsoft announces new supercomputer, lays out vision for future AI work. https://news.microsoft.com/source/features/ai/openai-azure-supercomputer/.

[5] 2024. Slurm Workload Manager. https://slurm.schedmd.com. Accessed: 2024.

[6] 2024. Tesla's Dojo Supercomputer: A Paradigm Shift In Supercomputing? https://www.forbes.com/sites/stevendickens/2023/09/11/teslas-dojo-supercomputer-a-paradigm-shift-in-supercomputing/.

[7] 2024. TOP500 Supercomputer Sites. https://www.top500.org. Accessed: 2024.

[8] 2025. RLTune GitHub Link. https://github.com/dshruti20/RLTune.

[9] Marcel Aach, Eray Inanc, Rakesh Sarma, Morris Riedel, and Andreas Lintermann. 2023. Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks. *Journal of Big Data* 10, 1 (2023), 96.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[11] Joshua Achiam. 2018. Spinning up in deep reinforcement learning.

[12] Hadeel Albahar, Shruti Dongare, Yanlin Du, Nannan Zhao, Arnab K Paul, and Ali R Butt. 2022. Schedtune: A heterogeneity-aware gpu scheduler for deep learning. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 695–705.

[13] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 533–546.

[14] Muhammad Aqib, Rashid Mehmood, Aiiad Albeshri, and Ahmed Alzahrani. 2017. Disaster management in smart cities by forecasting traffic plan using deep learning and GPUs. In *International Conference on Smart Cities, Infrastructure, Technologies and Applications*. Springer, 139–154.

[15] Betis Baheri, Jacob Tronge, Bo Fang, Ang Li, Vipin Chaudhary, and Qiang Guan. 2022. MARS: Malleable actor-critic reinforcement learning scheduler. In *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 217–226.

[16] S. Banerjee et al. 2021. Handling the Challenge of Heterogeneous Clusters for Deep Learning: A Case Study of AlphaFold. *J. Parallel and Distrib. Comput.* (2021).

[17] Danilo Carastan-Santos and Raphael Y De Camargo. 2017. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[18] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[19] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[20] Angel Cruz-Roa, Hannah Gilmore, Ajay Basavanhally, Michael Feldman, Shridar Ganesan, Natalie NC Shih, John Tomaszewski, Fabio A González, and Anant Madabhushi. 2017. Accurate and reproducible invasive breast cancer detection in whole-slide images: A Deep Learning approach for quantifying tumor extent. *Scientific reports* 7 (2017), 46450.

[21] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.

[22] Yuping Fan, Zhiling Lan, Taylor Childers, Paul Rich, William Allcock, and Michael E Papka. 2021. Deep reinforcement agent for scheduling in HPC. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 807–816.

[23] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey Fox, David Kanter, Thorsten Kurth, Peter Mattson, et al. 2021. MLPerf™ HPC: A holistic benchmark suite for scientific machine learning on HPC systems. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 33–45.

[24] Dror G Feitelson. 1996. Packing schemes for gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 89–110.

[25] Dror G Feitelson and Larry Rudolph. 1998. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'98 Workshop Orlando, Florida, USA, March 30, 1998 Proceedings 4*. Springer, 1–24.

[26] Dror G Feitelson, Dan Tsafrir, and David Krakov. 2014. Experience with using the parallel workloads archive. *J. Parallel and Distrib. Comput.* 74, 10 (2014), 2967–2982.

[27] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2022. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *arXiv preprint arXiv:2205.11913* (2022).

[28] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.

[29] M Shamim Hossain and Ghulam Muhammad. 2018. Environment classification for urban big data using deep learning. *IEEE Communications Magazine* 56, 11 (2018), 44–50.

[30] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[31] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[32] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 457–472.

[33] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[34] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[35] Vanessa Isabell Jurtz, Alexander Rosenberg Johansen, Morten Nielsen, Jose Juan Almagro Armenteros, Henrik Nielsen, Casper Kaae Sønderby, Ole Winther, and Søren Kaae Sønderby. 2017. An introduction to deep learning on biological sequence data: examples and solutions. *Bioinformatics* 33, 22 (2017), 3685–3690.

[36] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[37] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, and Dan Stanzione et al. 2020. Lessons learned from the chameleon testbed.. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 219–233.

[38] Redwan Ibne Seraj Khan, Kunal Jain, Haiying Shen, Ankur Mallick, Anjaly Parayil, Anoop Kulkarni, Steve Kofsky, Pankhuri Choudhary, Renee St Amant, Rujia Wang, et al. 2024. Ensuring Fair LLM Serving Amid Diverse Applications. *arXiv preprint arXiv:2411.15997* (2024).

[39] Redwan Ibne Seraj Khan, Arnab K Paul, Yue Cheng, Xun Steve Jian, and Ali R Butt. 2024. FedCaSe: Enhancing Federated Learning with Heterogeneity-aware Caching and Scheduling. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 52–68.

[40] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K. Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R. Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 135–152. https://www.usenix.org/conference/fast23/presentation/khan

[41] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[42] He Li, Kaoru Ota, and Mianxiong Dong. 2018. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. *IEEE network* 32, 1 (2018), 96–101.

[43] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 835–850.

[44] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2023. EasyScale: Elastic training with consistent accuracy and improved utilization on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[45] Tiangang Li, Shi Ying, Yishi Zhao, and Jianga Shang. 2023. Batch jobs load balancing scheduling in cloud computing using distributional reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems* 35, 1 (2023), 169–185.

[46] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient {GPU} cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[47] Andrew Makhorin. 2008. GLPK (GNU linear programming kit). *http://www. gnu. org/s/glpk/glpk. html* (2008).

[48] A. Mathuriya, A. Bard, P. Mendygral, L. Meadows, J. Arnemann, et al. 2021. Towards an optimized distributed deep learning framework for a heterogeneous multi-GPU cluster. *Cluster Computing* (2021).

[49] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI 20). 481–498.

[50] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. 2020. Job characteristics on large-scale systems: long-term analysis, quantification, and implications. In *SC20: International conference for high performance computing, networking, storage and analysis*. IEEE, 1–17.

[51] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[52] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.

[53] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1403–1420.

[54] Viktor Rausch, Andreas Hansen, Eugen Solowjow, Chang Liu, Edwin Kreuzer, and J Karl Hedrick. 2017. Learning a deep neural net policy for end-to-end control of autonomous vehicles. In *2017 American Control Conference (ACC)*. IEEE, 4914–4919.

[55] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[56] Wei Tang, Zhiling Lan, Narayan Desai, and Daniel Buettner. 2009. Fault-aware, utility-based job scheduling on blue, gene/p systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–10.

[57] Hind Taud and Jean-Franccois Mas. 2018. Multilayer perceptron (MLP). *Geomatic approaches for modeling land change scenarios* (2018), 451–455.

[58] F. Wang, G. Yang, H. Xu, X. Hu, and Y. Zhou. 2020. Optimizing Distributed Training Deployment in Heterogeneous GPU Clusters. In *The 34th ACM International Conference on Supercomputing (ICS)*.

[59] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[60] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 995–1008. https://www.usenix.org/conference/atc23/presentation/weng

[61] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[62] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[63] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. 2020. RLScheduler: an automated HPC batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[64] Di Zhang, Dong Dai, and Bing Xie. 2022. Schedinspector: A batch job scheduling inspector using reinforcement learning. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 97–109.

[65] Z. Zhang, Y. Wang, et al. 2020. Optimal Resource Efficiency with Fairness in Heterogeneous GPU Clusters. *arXiv preprint arXiv:2403.18545* (2020).

[66] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, et al. 2023. Silod: A co-design of caching and scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 883–898.

[67] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. 2020. {HiveD}: Sharing a {GPU} cluster for deep learning with guarantees. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 515–532.

[68] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. 2023. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 703–723.