# A Semantic-Aware Reinforcement Learning Scheduler for Parameterized Deep Learning Jobs on Heterogeneous Multi-GPU Clusters

**Zehao Zhou**

**A Thesis Submitted to the Faculty of Graduate Studies**
**In Partial Fulfillment of the Requirements**
**for the Degree of Master of Computer Science**

**Graduate Program in Electrical Engineering and Computer Science**

**York University**
**Toronto, Ontario**

**April 2026**

# Abstract

Efficient scheduling of deep learning workloads in heterogeneous GPU clusters is increasingly challenging due to diverse workload characteristics and stringent memory constraints. Existing approaches often rely on coarse-grained metrics such as runtime or resource demand, which fail to capture workload semantics and lead to resource misallocation, where high-memory GPUs are occupied by low-demand jobs while memory-intensive workloads remain blocked. To address this limitation, we first develop a semantic-aware workload simulator that models deep learning jobs using intrinsic attributes such as model type, dataset size, and training configuration, and estimates execution behavior based on hardware-agnostic computational workloads. This enables more realistic modeling of heterogeneous performance and system-level behaviors. Building on this foundation, we propose SARL, a Semantic-Aware Reinforcement Learning scheduler that incorporates workload semantics and hardware heterogeneity into scheduling decisions. We formulate the scheduling problem as a Markov Decision Process and introduce a model-based GPU allocation mechanism to enable efficient decision making under large action spaces. Experimental results demonstrate that SARL consistently outperforms strong baselines, achieving a 29.6% reduction in deadline miss rate and a 14.4% reduction in job completion time (JCT).

# Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Aijun An, for her continuous guidance and support throughout this research.

I also thank my committee members, Prof. Ping Wang and Prof. Zhenhao Li, for their valuable feedback and suggestions, which have helped me improved this work.

I acknowledge IBM for providing the opportunity and support that enabled this study, as well as for offering valuable practical insights and experience with real-world GPU cluster systems.

Finally, I would like to thank my senior teammate Yiming Shao for his encouragement and support. I also thank Hajer Ayadi for her helpful suggestions on topic selection and for her guidance during the early stages of building the simulation environment.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

Since the introduction of the Transformer architecture [2] and the subsequent emergence of GPT-3 [3], the field of deep learning has become one of the most prominent focuses in computer science, achieving remarkable success across numerous real-world applications. In particular, computationally intensive tasks such as graph analysis and natural language processing (NLP) have greatly benefited from these advancements. However, when dealing with large-scale datasets and complex model architectures, the computational demands become prohibitively high, often requiring cutting-edge GPU hardware for execution.

For example, in a preliminary experiment using PyTorch, we simulated a GPT-3–style model with a sequence length of 128, batch size of 32, and 50 epochs on a dataset containing 50,000 samples. The estimated resource consumption covered the entire training process—forward pass, backward propagation, and parameter updates using the Adam optimizer. Under these conditions, the model occupied approximately 79.3 GB of GPU memory and required at least $1.7 \times 10^{18}$ floating-point operations (FLOPs). Given that the state-of-the-art NVIDIA A100 GPU delivers a peak throughput of 312 TFLOPS (trillion floating-point operations per second) [4], the theoretical lower bound for completing this workload is approximately 1.5 hours. In practice, however, the actual training time is typically much longer due to memory limitations, communication overhead, and suboptimal hardware utilization. Consequently, training models of this scale on a single GPU is increasingly impractical in terms of both cost and time, particularly in real-world scenarios [5]. This has led to distributed learning over a cluster of machines becoming the standard approach, with individual machines equipped with several GPUs[6, 7].

Meanwhile, with the rapid advancement of AI and the ever-increasing scale of model architectures—from the 175 billion parameters in GPT-3 (2020) to the 671 billion parameters in DeepSeek-R1 (2025)[3, 8]—the memory required to store and train such models has grown dramatically. As a result, memory bottlenecks have become a critical challenge in large-scale training workloads, directly limiting system throughput and overall efficiency [6, 9].

Yet ironically, despite this growing demand, resource utilization in production environments remains strikingly inefficient [10]. Under the commonly adopted data-parallel training paradigm, lightweight jobs may occupy high-memory GPUs despite requiring only a fraction of their capacity. Meanwhile, a more complex job waiting in the queue may require substantially larger GPU memory. In such cases, although lower-tier GPUs may still be idle, their memory capacity may be insufficient to satisfy the job's requirements. As a result, the job must remain in the waiting queue until the high-end GPUs become available, which increases the risk of significant deadline overruns.

This situation commonly appears in modern deep learning workloads. For example, models such as ResNet-style architectures used in large-scale vision tasks often require relatively moderate GPU memory per device but may run for extended periods due to extremely large datasets. The ImageNet dataset alone contains tens of millions of images and is widely used for training large-scale models [11, 12]. Consequently, these jobs can occupy powerful GPUs for long durations. In contrast, large language models (LLMs) such as GPT-style represent another extreme. In many practical scenarios, these models are fine-tuned rather than trained from scratch. Fine-tuning refers to adapting a pretrained model using a smaller task-specific dataset (for example, sentiment analysis on the IMDB 50K dataset [13]). Such workloads typically have shorter training durations but require significantly larger GPU memory footprints, creating strong memory pressure during scheduling.

Recent large-scale models impose increasingly stringent compute demands, often exceeding the capacity of a single accelerator. In practice, modern systems adopt distributed training (e.g., Distributed Data Parallel, DDP) to scale such workloads. However, under this paradigm, the coexistence of heterogeneous GPUs and memory-constrained workloads introduces new scheduling challenges. Efficiently allocating limited high-memory GPUs while maintaining overall system performance becomes increasingly difficult.

In particular, practical clusters must handle diverse workloads with varying urgency and resource requirements, where naive scheduling policies can lead to deadline violations or suboptimal job completion time (JCT). Therefore, this work focuses on improving scheduling intelligence at the system level.

To formalize this problem, we consider the scheduling of deep learning training jobs in a heterogeneous GPU cluster, where multiple jobs with diverse resource demands and semantic characteristics arrive over time. Each job is associated with workload-specific attributes, such as model type, dataset size, and training configuration, which influence its computational and communication cost. At each decision point, the scheduler must select the most suitable job(s) from the pending queue and allocate appropriate GPU resources, including the type of GPUs within the cluster.

The objective is to minimize the deadline miss rate while maintaining efficient JCT. The scheduling process is inherently dynamic and must account for system-level factors such as communication overhead, resource fragmentation, and reconfiguration costs. Due to the tightly coupled interactions among workload semantics, hardware heterogeneity, and cluster state, designing effective scheduling policies remains a challenging problem.

Designing an effective scheduler for deep learning workloads presents several unique challenges. Unlike traditional HPC jobs, which are often characterized by runtime and resource requests, deep learning workloads exhibit rich semantic properties that directly determine their execution behavior. For instance, memory consumption depends on model architecture and batch size, while job duration is influenced by dataset scale and training

configuration.

However, such semantic information is unavailable to the scheduler in practice. As a result, scheduling decisions are often based on coarse signals which are insufficient for accurately estimating resource requirements or predicting execution behavior.

This limitation is further exacerbated by the lack of semantic information in publicly available cluster traces. Widely used datasets, such as Alibaba cluster traces [14], typically record only job runtime without exposing key workload attributes. Consequently, many prior works approximate job length solely based on observed runtime [15, 16, 17].

Such abstractions introduce several critical challenges. First, they implicitly assume homogeneous hardware environments, where runtime is comparable across devices. In contrast, modern GPU clusters are increasingly heterogeneous [18], making runtime an unreliable proxy for workload complexity. Second, in online scheduling settings, the absence of semantic information prevents schedulers from accurately estimating job characteristics, forcing reliance on indirect signals such as GPU demand or historical statistics. Third, it obscures important execution factors that depend on workload semantics, such as communication overhead and preemption cost. While communication behavior is closely tied to model structure and parameter scale [19], it cannot be reliably inferred from runtime alone. More importantly, preemption overhead can vary significantly across jobs—interrupting some workloads may incur minimal cost, while others may require nearly restarting training from scratch [20]. Without semantic context, such differences remain invisible, leading to suboptimal scheduling decisions.

Fortunately, recent studies indicate that workload semantic attributes are becoming increasingly available in modern production GPU clusters. For example, Wang et al. [21] show that commercial cluster traces can include rich metadata such as model types and training configurations, making semantic-aware scheduling practically feasible. Furthermore, Lee et al. [22] demonstrate that such semantic features can be leveraged to accurately predict training time, achieving prediction errors within a small range.

At the same time, distributed training introduces significant communication overhead that depends on both workload semantics and resource placement. Prior work shows that communication cost is highly sensitive to GPU locality and scales with model size [23, 24, 25]. In large-scale models, communication can dominate total training time [19], making topology-aware scheduling essential.

Taken together, these challenges highlight the need for scheduling policies that jointly consider workload semantics, hardware heterogeneity, and communication locality. The interactions among these factors are highly coupled and dynamic, making it difficult to design effective rules or heuristics that generalize across workloads.

This naturally aligns with the strengths of reinforcement learning, which can learn adaptive scheduling policies by capturing complex system-level interactions through experience.

To address these challenges, we propose a Semantic-Aware Reinforcement Learning Scheduler (SARL), a learning-based scheduling framework that incorporates workload semantics and hardware heterogeneity into decision making. The main contributions of this work are summarized as follows:

- **Semantic-Aware Deep Learning Workload Simulator**

  We propose a semantic-aware workload simulator that represents each job using a compact set of semantically meaningful attributes, including model type, dataset size, batch size, number of epochs, and GPU demand. Based on these attributes, job execution is modeled using hardware-agnostic computational workloads (measured in TFLOPs) rather than fixed runtime. This allows the simulator to adapt execution time according to the computational capability of different GPUs.

  We further incorporate model-aware performance estimation. For convolutional neural networks (e.g., ResNet, VGG, Inception), job length is estimated based on model size, number of epochs and samples in the dataset. For large language models, we adopt scaling-law-based estimation and calibrate predictions using prior performance modeling techniques (e.g., NeuSight [22]). Memory consumption is estimated using a profiling-based approach combined with regression modeling. Specifically, we first collect empirical memory usage data through offline profiling across different batch sizes for each model. Based on these measurements, we fit regression models to estimate memory consumption for unseen batch sizes of the same model.

  In addition, the simulator models system-level behaviors that depend on workload semantics, including communication overhead and preemption cost. Different model families exhibit distinct communication patterns and checkpointing overheads, which are explicitly captured in the simulation.

  Finally, we design a deadline generation mechanism that reflects job urgency rather than purely random scaling of runtime. Deadlines are derived from estimated execution time based on cluster performance, with different urgency levels assigned to jobs to better mimic real-world usage.

  By integrating semantic attributes with model-aware execution modeling, the proposed simulator generates workloads that more closely resemble real-world deep learning training jobs, enabling more realistic evaluation of scheduling policies.

- **Topology- and Heterogeneity-Aware Scheduling Representation**

  To enable effective scheduling in realistic GPU clusters, we design a topology- and heterogeneity-aware representation of the system state.

  First, we explicitly model the hierarchical structure of the cluster (rack $\rightarrow$ machine $\rightarrow$ GPU) to capture communication locality. Since distributed training performance

is sensitive to inter-GPU communication, the scheduler must be able to distinguish whether GPUs are allocated within the same machine, within the same rack, or across racks.

Second, we incorporate hardware heterogeneity into the state representation. Modern GPU clusters often consist of multiple GPU types with varying computational capabilities and memory capacities. As a result, scheduling decisions must consider not only resource availability but also the performance implications of assigning different GPU types to a job.

This representation enables the scheduler to jointly reason about resource availability, communication cost, and hardware capability, providing a more informative state for decision-making.

- **Single-Agent Model-Based GPU Allocation RL Scheduler**

  We propose a single-agent reinforcement learning scheduling framework with model-based GPU allocation for heterogeneous clusters.

  Prior work has explored both single-agent and dual-agent designs. For example, Ryu et al. adopts a single-agent formulation that jointly determines job selection and fine-grained GPU allocation [26]. Similar single-agent designs have also been explored in systems such as DeepGJS [27]. However, such joint decision-making leads to a large action space and degraded learning efficiency in complex environments [15].

  More recent approaches employ dual-agent or decoupled strategies with cooperative training or heuristic components [28, 15], which reduce the action complexity but introduce additional training overhead and coordination challenges, especially in heterogeneous clusters.

  To address these limitations, we introduce a hierarchical decision mechanism. After selecting a job, the RL agent selects a GPU model, corresponding to a group of GPUs with identical hardware characteristics (e.g., computational capability and memory capacity). A lightweight heuristic is then used to assign specific GPUs within the selected model.

  This design significantly reduces the action space compared to fine-grained allocation while preserving sensitivity to hardware heterogeneity. As a result, it achieves a practical balance between scalability and scheduling effectiveness.

- **Performance Improvements**

  Experimental results demonstrate that our approach is highly effective. Compared to strong baseline methods, **SARL** achieves a 29.6% reduction in average deadline miss rate,and a 14.4% improvement in average JCT. These results highlight the

effectiveness of incorporating workload semantics and hardware heterogeneity into scheduling decisions.

The remainder of this paper is organized as follows.

Chapter 2 provides background on deep learning workloads and GPU cluster systems, covering distributed training paradigms, cluster architectures, communication characteristics, preemption mechanisms, and representative scheduling approaches in multi-GPU environments.

Chapter 3 presents the methodology of the proposed framework. It first formulates the scheduling problem in heterogeneous GPU clusters, and then describes the overall system design, including the semantic-aware simulation environment and the reinforcement learning–based scheduler. The chapter further details the formulation of the scheduling process as a Markov Decision Process, including the state representation, action space, and reward design, as well as the training methodology.

Chapter 4 evaluates the proposed approach through comprehensive experiments. It describes the experimental setup, including baseline methods and cluster configurations, and compares the proposed scheduler with existing approaches using multiple performance metrics. The chapter further analyzes performance under different resource conditions and presents ablation studies to examine the contribution of key design components.

Finally, Chapter 5 concludes the thesis and discusses potential directions for future work.

# 2 Literature Review

Deep learning has become a fundamental component of modern software systems, with applications ranging from everyday tasks such as face recognition and image analysis to advanced domains including autonomous driving and large language models such as GPT. These applications rely heavily on large deep learning models to achieve high prediction accuracy. Improving model performance typically requires both large-scale training datasets and increasingly complex model architectures. For example, Kaplan et al. observed that increasing the number of model parameters can systematically reduce training loss and improve model performance [29].

In a typical modern deep learning training process, model parameters are iteratively optimized using gradient-based learning algorithms. Each training iteration consists of three main stages: the forward pass, the backward pass, and the parameter update. During the forward pass, the model processes a mini-batch of input data and produces predictions. The backward pass then computes gradients of the loss function with respect to model parameters using the backpropagation algorithm. Finally, the model parameters are updated according to an optimization rule such as stochastic gradient descent (SGD) [30] or the Adam optimizer [31]. Formally, the parameter update at iteration $t$ can be expressed as [32]

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$$

where $\theta$ denotes the model parameters, $\eta$ is the learning rate, and $\mathcal{L}$ represents the training loss function.

The computational cost of the training procedure scales with both the size of the training dataset and the number of model parameters. Larger datasets require more training steps to process all samples, while larger models increase the computational workload and memory footprint of each forward and backward pass.

## 2.1 Estimating job length from Job Semantics

However, existing publicly available cluster traces typically use the observed job runtime as a proxy for job length [14, 33], while omitting detailed semantic information about the jobs themselves. Such information includes model architecture, batch size, dataset characteristics, and operator-level configurations. The absence of these semantics significantly limits the ability of schedulers to make informed decisions, particularly for learning-based approaches such as reinforcement learning (RL), where the environment is often modeled as a Markov Decision Process (MDP). Without access to intrinsic job properties, the scheduler effectively treats jobs as opaque entities, making it difficult to generalize across workloads or adapt to unseen scenarios.

In practice, however, job-level semantic information is often accessible. For instance, modern machine learning platforms and APIs require users to explicitly specify model configurations and training parameters. A representative example is the OpenAI fine-tuning API, where users must provide details such as the base model (e.g., GPT variants), batch size, and training dataset [34]. This suggests that job semantics are not inherently unavailable, but rather underutilized in existing cluster datasets and scheduling frameworks.

Recent studies have begun to explore how such semantic information can be leveraged. Wang et al. [21] point out that conventional schedulers "do not understand the jobs," and propose extracting semantic features from job logs using LLMs. Their work demonstrates that meaningful job representations can be derived from textual and structural information, which in turn can enhance scheduling decisions.

Beyond semantic extraction, another line of work investigates how job semantics can be used to quantitatively estimate job execution time. In particular, Lee et al. propose NeuSight [22], a performance forecasting framework that predicts the latency of deep learning workloads on GPUs without requiring actual execution. NeuSight takes as input both job-level semantics (e.g., operator types and tensor dimensions) and GPU hardware characteristics (e.g., memory bandwidth, peak FLOPS, and cache size). Instead of directly regressing end-to-end runtime, it models GPU execution at a finer granularity by decomposing each operator into smaller computational tiles, estimating device utilization using a machine learning model, and aggregating these estimates under fundamental performance constraints. Through this hybrid approach, NeuSight achieves high prediction accuracy even for unseen models and GPUs.

Importantly, NeuSight provides strong empirical evidence that job semantics, when combined with hardware-aware modeling, are sufficient to accurately predict job runtime on a single GPU, even across heterogeneous hardware platforms. This establishes a reliable foundation for reasoning about job length beyond raw runtime observations. In particular, it enables the possibility of generalizing job length across different hardware environments by relating execution time to normalized compute metrics such as FLOPS or throughput. This insight motivates our approach to redefine job length in a more portable and hardware-aware manner, rather than relying solely on historical runtime measurements.

While these methods demonstrate that job semantics can be effectively used to estimate execution time on a single GPU, modern deep learning workloads are rarely confined to a single device. In practice, large-scale models and datasets often exceed the computational and memory capacity of individual GPUs, necessitating the use of multiple devices for training. As a result, job length is no longer solely determined by single-device execution, but also by the parallelization strategy and inter-device communication overhead.

## 2.2 Extending job length Estimation to Distributed Training

Consequently, as both dataset size and model complexity continue to grow, the computational and memory demands of deep learning training have increased dramatically. To address these challenges, developers distribute the training workload across multiple computing devices, a paradigm known as distributed training [1]. Today, distributed training has become a standard feature in modern deep learning frameworks, including TensorFlow [35] and PyTorch [36].

The growing importance of distributed training is also reflected in developer communities. For instance, the number of questions related to distributed deep learning on Stack Overflow has increased steadily over the past decade, as illustrated in Figure 1.

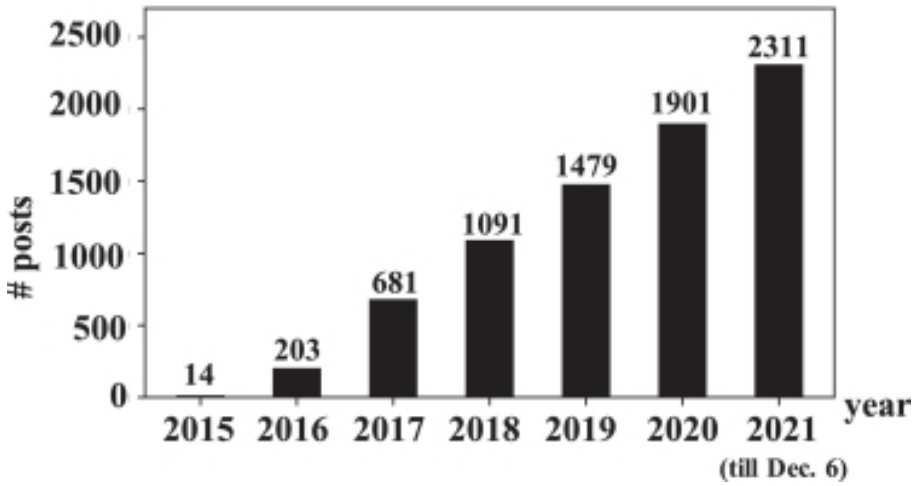



Figure 1: Trend of Stack Overflow questions related to distributed deep learning over time (adapted from [1]).

Compared with training deep learning models on a single GPU, distributed training typically imposes stricter requirements on the underlying computing environment. First, provisioning and maintaining multiple high-performance GPUs can be costly for individual users. In practice, large-scale deep learning training often requires clusters equipped with many GPUs, which are typically available only in dedicated servers or data centers. Second, distributed training generally benefits from homogeneous hardware configurations. A typical training job is executed across multiple GPUs of the same model to ensure balanced computational performance and efficient communication. Using heterogeneous GPUs may lead to performance imbalance, where slower devices become bottlenecks during synchronization, thereby degrading the overall training efficiency. Consequently, large-scale distributed training is commonly performed in specialized computing clusters or data centers designed to support such workloads.

To better understand how distributed training workloads are executed in these environments, it is useful to first review the basic workflow of deep learning training.

## 2.3 Workflow of Distributed Deep Learning Training

In distributed deep learning systems, a training task is first decomposed into multiple subtasks that can be executed in parallel across different computing devices. Several parallelization strategies have been proposed to support distributed training, among which the most widely adopted approaches include data parallelism and model parallelism [1]. In addition, recent studies have explored hybrid approaches that combine multiple parallelization techniques to further improve scalability and efficiency [37].

### 2.3.1 Data Parallelism

Data parallelism is the most commonly used distributed training paradigm [38] and is widely supported by modern deep learning frameworks such as PyTorch. In this approach, the training dataset is partitioned into multiple non-overlapping subsets, each of which is processed by a separate GPU. Meanwhile, every GPU maintains a replica of the same deep learning model [37, 1, 39]. Frameworks such as PyTorch provide built-in modules (e.g., *DistributedDataParallel*) that enable data-parallel training with minimal modifications to existing training code.

This strategy demonstrates strong scalability in large-scale training tasks. For example, when training ResNet-50 on the ImageNet dataset using large mini-batches, data-parallel training has been shown to achieve near-linear scaling across multiple GPUs [40].

However, due to the need for gradient synchronization among GPUs, data-parallel training inevitably introduces communication overhead, which can become a significant performance bottleneck in large-scale distributed systems [37].

### 2.3.2 Model Parallelism

In contrast to data parallelism, model parallelism partitions a deep learning model into multiple components that are executed on different devices. Each GPU stores and computes only a subset of the model parameters, allowing the training of models whose size exceeds the memory capacity of a single device [41, 1, 37].

During training, a mini-batch of data must pass through these model partitions sequentially. As a result, some devices may remain temporarily idle while waiting for the next portion of the mini-batch to arrive from preceding stages. This idle period is commonly referred to as the *pipeline bubble*, which reduces overall hardware utilization [37].

Another challenge of model parallelism lies in determining how to partition the model computation across devices. Unlike data parallelism, where each worker simply maintains a replica of the entire model, model-parallel training requires careful design of the model partitioning strategy to balance computation and communication across devices [42].

### 2.3.3 Hybrid Parallelism

Hybrid parallelism combines both data parallelism and model parallelism within the same training system. In this approach, different parallelization strategies are applied to different parts of the neural network according to their computational and memory characteristics [37].

Empirical studies have shown that well-designed hybrid parallel strategies can achieve lower communication overhead than pure data-parallel training under the same hardware conditions [43].

However, hybrid parallelism also introduces additional system complexity. In addition to determining how the model should be partitioned, the system must decide which parallelization strategy should be applied to each component of the model. Furthermore, hybrid approaches inherit the challenges associated with both data and model parallelism, such as gradient synchronization overhead and pipeline inefficiencies. As a result, hybrid parallelism is often viewed as a trade-off between communication overhead and computational efficiency in distributed systems [43].

Among these approaches, data parallelism remains the dominant strategy in many practical deep learning systems due to its relative simplicity and scalability. However, the efficiency of data-parallel training is highly influenced by communication overhead and resource allocation across GPUs, which motivates the need for effective scheduling and resource management in GPU clusters.

## 2.4 GPU Cluster Architecture

Modern GPU clusters are typically organized in a hierarchical structure. A cluster consists of multiple racks, each rack contains several machines (or nodes), and each machine is equipped with multiple GPUs, commonly ranging from four to eight devices per node. In many production environments, GPUs within the same machine are usually of the same model to ensure balanced computational performance.

Within a single machine, GPUs are typically interconnected using high-speed links such as NVLink [7]. Compared with traditional PCIe interconnects, NVLink provides substantially higher bandwidth and lower latency, often achieving several times the bandwidth of PCIe connections [44]. This high-speed interconnect enables efficient communication among GPUs located on the same node.

Across different machines within the same rack, communication is commonly supported by high-performance networking technologies such as InfiniBand. Although the bandwidth of InfiniBand is generally lower than that of NVLink, it remains significantly faster than conventional Ethernet-based connections and is widely used in high-performance computing clusters [45, 46].

Communication between machines located in different racks typically relies on Ethernet-

based networks, which usually provide lower bandwidth and higher latency compared with NVLink and InfiniBand. Consequently, the physical topology of the cluster can have a substantial impact on communication performance in distributed deep learning workloads.

It is worth noting that in Yang et al.'s work [7], the term *cluster* is used to refer to a group of machines that are physically organized within a single rack. In this thesis, we adopt the more commonly used hierarchical terminology in large-scale computing infrastructures and treat such a group as a *rack* within a larger GPU cluster.

This terminology is consistent with observations from real-world production systems. For example, Kamra et al. study the infrastructure of a commercial IBM compute cluster and describe a hierarchical organization consisting of clusters, racks, and machines [47]. Such hierarchical architectures naturally introduce multiple levels of communication locality within GPU clusters.

Consequently, for distributed deep learning workloads—particularly those using data parallelism—the communication cost between GPUs generally follows an increasing order based on their physical distance in the cluster topology. Communication among GPUs within the same machine is typically the fastest due to high-speed interconnects such as NVLink, followed by communication across machines within the same rack, while communication across different racks often incurs the highest latency and lowest bandwidth.

Understanding these communication hierarchies is essential for analyzing the performance of distributed training systems. Therefore, the next subsection discusses the communication mechanisms used in distributed deep learning workloads.

## 2.5 Communication in Deep Learning Models

As discussed earlier, data parallelism dominates distributed deep learning training due to its conceptual simplicity, wide applicability, and strong framework support. However, the primary challenge associated with data-parallel training is the communication overhead introduced by gradient synchronization.

In data-parallel training, each worker computes gradients locally using its assigned subset of data. After the backward pass, these gradients must be synchronized across all workers through collective communication operations such as *AllReduce* in order to ensure that all model replicas maintain consistent parameters [48].

In modern frameworks such as PyTorch's DistributedDataParallel (DDP), gradient synchronization is typically implemented using a technique known as *bucketed AllReduce* [48]. Instead of synchronizing gradients parameter by parameter, multiple gradient tensors are grouped into communication buckets. Once all gradients in a bucket become available during the backward pass, an AllReduce operation is launched to aggregate them across workers. This design allows communication to overlap with ongoing gradi-

ent computation while also amortizing the overhead of communication across multiple tensors [48].

However, the choice of bucket size introduces an important trade-off. Larger buckets may introduce additional communication latency, since an AllReduce operation can only start after all gradients in the bucket become available during the backward pass. This may lead to an increase of time in each iteration. Conversely, smaller buckets allow earlier communication but may incur higher communication overhead due to more frequent collective operations [48].

As the scale of distributed training increases, particularly with a growing number of GPUs, the cost of gradient synchronization can become a dominant component of the total training time [48]. Therefore, modeling communication overhead is essential when analyzing the performance of data-parallel workloads in GPU clusters, especially when taking the physical topology of GPUs into account. Since GPUs located within the same machine, rack, or across racks exhibit significantly different communication characteristics, the placement of GPUs allocated to a training job can greatly affect its performance.

In practice, deep learning training jobs often run for extended periods of time and are executed in shared GPU clusters where workloads arrive dynamically. As a result, cluster resource management mechanisms must support flexible runtime adjustments in order to maintain efficient utilization of the available GPUs. One commonly used mechanism to enable such flexibility is *preemption*, which allows the system to interrupt running workloads and reallocate resources when necessary.

## 2.6 Preemptions

Preemption refers to the ability of a system scheduler to interrupt a currently running job, release its allocated resources, and return the job to the waiting queue so that higher-priority workloads can be executed. This mechanism is commonly used in shared computing environments where multiple jobs compete for limited resources. For example, a scheduler may preempt a running job in order to allocate GPUs to a newly arrived task with a tighter deadline.

For non-distributed training workloads that execute on a single device, the preemption process is relatively straightforward. The system only needs to save the current training state, including model parameters and optimizer states, and release the occupied resources. When the job resumes in the future, the training process can simply reload the saved model state from storage and continue execution.

However, preemption becomes significantly more complicated in distributed training scenarios. In distributed settings such as data parallelism, training involves frequent communication among multiple GPUs. For instance, gradient synchronization is typically

performed through collective communication operations such as *AllReduce*. If a job is interrupted during such communication phases, the participating workers may end up with inconsistent states, which can lead to instability or incorrect model updates when training resumes.

To avoid such inconsistencies, a safe approach is to allow preemption only at well-defined synchronization boundaries, such as the end of a training iteration. In practice, however, cluster schedulers often cannot precisely guarantee that preemption occurs at these safe points due to the unpredictable nature of system events.

To address this challenge, two major mechanisms are commonly adopted in practice: *checkpointing* [49, 16, 20] and *context-based suspension* [50, 51, 52]. Checkpointing is a well-established technique that periodically saves the training state to local storage(e.g., SSD) so that the job can later be resumed from the saved state. This mechanism is widely supported in modern deep learning frameworks such as PyTorch and TensorFlow [53, 54]. In practice, implementing checkpointing is relatively straightforward. For example, in PyTorch-based training pipelines, users can periodically save model states using a checkpoint callback mechanism, as illustrated in the following simplified example:

```
Procedure Checkpointing
    Every N training steps do
        Save(model_parameters)
        Save(optimizer_state)
        Write state to local storage
    End
```

The primary limitation of checkpoint-based preemption lies in the overhead associated with periodically saving the training state. In particular, checkpointing introduces additional I/O costs because model parameters and optimizer states must be written to persistent storage. This overhead becomes especially significant for large-scale models. For example, modern large language models such as GPT-3 or GPT-4 may contain hundreds of gigabytes to several terabytes of parameters. Consequently, in practical deployments, checkpoints are typically saved only every few hours in order to reduce the I/O burden [20]. As a result, a preemption event may cause the loss of several hours of training progress.

Moreover, some production GPU clusters do not provide built-in checkpointing support, which means that the training state cannot be safely preserved when preemption occurs. In such cases, a preempted job may lose its entire training progress and must restart from the beginning [52].

An alternative approach is *context-based suspension*, commonly referred to as GPU context switching. This method is conceptually similar to process context switching in CPU scheduling. Instead of terminating the training process or writing model states to

disk, the system temporarily pauses the GPU execution state and saves the GPU context to host memory, after which the GPU can be reassigned to execute another task [50]. Compared with checkpointing, this approach avoids the large I/O overhead associated with periodically saving and restoring model parameters.

In practice, context-based suspension can be significantly faster. For example, assuming a PCIe bandwidth of approximately 20 GB/s, transferring tens of megabytes of GPU context to host memory may take only a few milliseconds, whereas restoring model checkpoints of tens of gigabytes may take tens of seconds. However, this approach also faces several challenges. First, GPU contexts are substantially larger than CPU contexts, leading to higher context-switch overhead (e.g., approximately 44 MB for GPU contexts compared with roughly 1 KB for CPU contexts) [50]. More importantly, modern GPUs generally lack a user-facing interface for explicit context switching, making it difficult for cluster schedulers to directly control this mechanism [50].

## 2.7 Task Scheduling in Multi-GPU Clusters

Broadly, scheduling strategies can be categorized into three classes: heuristic-based methods, optimization-based approaches, and learning-based methods, particularly those leveraging deep reinforcement learning (DRL). These categories differ in how scheduling decisions are derived—whether through predefined rules, explicit objective optimization, or data-driven policy learning.

### 2.7.1 Heuristic-based methods

Heuristic methods include simple greedy strategies such as First-In-First-Out (FIFO) and Shortest Job First (SJF) [55, 56]. These algorithms typically rely on handcrafted rules and do not explicitly account for workload-specific features or system-level complexities. Nevertheless, they are widely adopted in practice due to their simplicity, generality, and low scheduling overhead [57, 58, 59].

As one of the most fundamental algorithms, **First-In-First-Out (FIFO)** is among the simplest and most widely adopted scheduling strategies, and became the default algorithm for thread block scheduler of many GPUs [60]. It processes jobs strictly in the order of their arrival, without considering factors such as resource demands, expected execution time, or deadline. In multi-GPU clusters, FIFO places incoming training jobs into a queue and launches them sequentially once sufficient resources become available [17]. Its advantages lie in its simplicity and minimal scheduling overhead [57], which is why it is commonly used as a default policy in production systems. However, in resource-constrained or workload-heterogeneous scenarios, FIFO may lead to poor resource utilization: large jobs that consume significant memory can block the queue, delaying smaller jobs that could otherwise be scheduled immediately. This often results in volatile system

throughput [60]. Moreover, since FIFO is a non-preemptive algorithm, shorter jobs arriving during the execution of longer ones must wait until the current job finishes. This can result in a scheduling behavior similar to the worst-case strategy—Longest Job First. As a consequence, FIFO performs poorly on fairness metrics such as StrictF [60].

**Shortest Job First (SJF)** is another classical scheduling strategy that prioritizes jobs with the shortest estimated execution time, aiming to minimize the system's average waiting time [58, 61]. While highly efficient for workloads dominated by short jobs, a major drawback of SJF is that longer jobs may suffer from starvation under high system load, leading to poor fairness and reduced throughput.

**Earliest Deadline First (EDF)** is a widely used real-time scheduling algorithm that orders jobs according to their deadlines, giving higher priority to jobs whose deadlines occur sooner. In its preemptive form, if a newly arriving job has a tighter deadline than the currently running job, the scheduler interrupts the current execution and reallocates the processor to the new job. EDF has been shown to be optimal for deadline-based scheduling on a single processor under standard real-time assumptions [62]. However, this optimality does not directly extend to distributed cluster environments, where scheduling decisions must consider multiple GPUs, heterogeneous resource capacities, and varying job characteristics such as memory demand and execution length. As a result, EDF may fail to achieve optimal deadline satisfaction or overall system efficiency in such settings.

Once the job scheduling order is determined, a classic strategy for mapping jobs to GPUs is **First-Fit**. The core idea of this method is to sequentially scan the resource pool and assign each job to the first available GPU (or group of GPUs) that meets its resource requirements [56]. Due to its fixed traversal order, First-Fit is simple to implement and offers good resource utilization while effectively reducing fragmentation [56, 17]. However, because of its greedy nature and lack of backtracking, the algorithm may only achieve a locally optimal solution.

An early attempt to combine job scheduling policies with resource allocation strategies is the **Rate-Monotonic First-Fit (RMFF)** algorithm [63]. This approach is conceptually similar to combining SJF with First-Fit, but differs in that it employs a preemptive scheduling policy based on task urgency—specifically, the tightness of task deadlines [64]. Compared to SJF, RMFF can theoretically achieve a higher deadline satisfaction rate, potentially reaching 100% under certain conditions [63].

Based on the above strategies, researchers have proposed a variety of scheduling algorithms tailored to local conditions, striving to achieve higher GPU utilization and fairness while meeting different constraints (such as deadlines, resource balance and preemption overhead)

In 2019, Gu *et al.* proposed **Tiresias**, a modern heuristic scheduling algorithm for deep learning workloads on GPU clusters [16]. It was evaluated using real traces from a Microsoft production cluster in simulation, and on the Michigan ConFlux cluster with

60 NVIDIA P100 GPUs [16]. Tiresias adopts *exclusive GPU allocation*: a DL job either acquires all required GPUs at once or waits. It also supports *preemptive scheduling* with carefully estimated costs—each preemption is measured to incur a saving overhead of 1.3 to 26.3 seconds for the main GPU [16].

The scheduling policy revolves around the concept of **attained service**, defined as the product of a job's runtime and the number of GPUs allocated to it. Once a job's attained service exceeds the total number of GPUs in the cluster, it is demoted to a lower-priority queue to prevent starvation.

Within each priority queue, job ordering is determined based on the availability of historical information:

- **With prior knowledge:** If historical execution time distributions for similar jobs are available, the **Discretized 2D-Gittins Index** is used to compute priority. This index originates from the classic Gittins Index theory [65], and is defined as:

$$\text{Gittins Index(priority)} = \frac{P(\text{completion in } \Delta)}{E[\text{remaining service}]} \text{ [16]}$$

  where $\Delta$ is a tunable parameter called the *Service Quantum*, typically chosen such that 60% of jobs finish within this period. Jobs with a low probability of completion within $\Delta$ and high remaining service are assigned lower priority.

- **Without prior knowledge:** The **Discretized 2D-LAS** (Least Attained Service) metric is employed instead, defined as:

$$\text{Priority} = \frac{1}{W_J \times t_J} \text{ [16]}$$

  where $W_J$ is the number of GPUs allocated to job $J$, and $t_J$ is its elapsed runtime.

For jobs with equal priority, FIFO is applied. To avoid high overhead from frequent preemptions, priority re-evaluation is performed at fixed intervals rather than continuously.

Despite its effectiveness, Tiresias has several limitations. It assumes a homogeneous cluster where all GPUs are identical and does not consider memory usage heterogeneity. Additionally, it does not estimate job complexity using FLOPs or user-defined epochs, relying instead on the empirical observation that most jobs terminate early due to hyperparameter tuning. The algorithm also does not handle deadline constraints.

Although heuristic scheduling methods (e.g., FIFO, SJF) are widely adopted due to their simplicity and low overhead, they lack dynamic adaptability. As a result, they struggle to handle the heterogeneity of jobs in multi-GPU environments—such as varying memory demands and computational intensity—as well as strict deadline constraints.

To enhance flexibility, history-based schedulers like *Tiresias* leverage information from previously executed similar jobs to predict the running time of incoming ones, thereby enabling dynamic prioritization and preemption. However, such approaches still encounter critical limitations in production environments:

- **Limited workload semantic modeling**: Scheduling decisions are based on historical runtime estimates, without explicitly modeling job-level semantics such as computational workload, scaling behavior, or training configuration. This limits their ability to generalize across different models and hardware settings.
- **Insufficient support for deadline-aware scheduling**: Due to the lack of semantic and placement-aware information, existing approaches rely heavily on historical JCT for prioritization. However, JCT alone cannot distinguish whether a long execution time is caused by the intrinsic complexity of the job (e.g., large model or dataset) or by suboptimal resource placement (e.g., GPUs distributed across machines or racks, leading to high communication overhead). As a result, the scheduler may misinterpret the underlying causes of performance degradation and make suboptimal prioritization decisions, particularly when handling dynamically arriving high-priority or latency-sensitive jobs.

Therefore, there is a pressing need for a DRL framework that is better aligned with production settings—one that can explicitly model the coupling of computation, communication, and memory, and dynamically respond to evolving deadline constraints.

### 2.7.2 DRL-based methods

Considering that heuristic-based algorithms often lack flexibility in complex and dynamic environments, recent studies have explored learning-based approaches for resource scheduling. In particular, deep learning and DRL methods enable data-driven decision making without relying on handcrafted models or expert knowledge.

A representative work is DL2 [66], which proposes a deep learning-driven scheduler for distributed deep learning clusters. Instead of relying on analytical performance models or manually designed heuristics, DL2 adopts a hybrid learning paradigm that combines supervised learning and reinforcement learning. Specifically, the scheduler is first initialized using offline supervised learning based on historical scheduling traces, and then further refined online via reinforcement learning to adapt to dynamic cluster conditions. The learning objective is guided by a relatively simple reward function, primarily focusing on job-level performance metrics such as JCT.

However, when extending such a framework to modern distributed training paradigms such as data parallelism (e.g., DDP), several limitations arise. In DL2, GPUs are allocated to jobs incrementally in a fine-grained manner. In contrast, under synchronous

data-parallel training, the set of participating devices typically forms a tightly coupled worker group, where adding or removing GPUs requires reconfiguration of the process group and redistribution of model states. This necessitates pausing the training job and reinitializing communication contexts, effectively introducing implicit preemption and restart overhead.

As a result, frequent incremental allocation decisions may lead to repeated disruptions of running jobs, significantly degrading training efficiency. Furthermore, DL2 does not explicitly model communication costs in its scheduling decisions. In heterogeneous or multi-rack clusters, this may result in highly fragmented GPU allocations, where devices assigned to the same job are distributed across different machines or racks. Such placements can incur substantial synchronization overhead (e.g., during all-reduce operations), ultimately increasing job completion time and limiting scalability.

One step further, Kexuan and Jiayi (2023) proposed a reinforcement learning framework, **DeepGJS**, for job scheduling [27].

The simulated environment consists of a cluster with multiple servers $R_i$, where each $R_i$ is represented by a three-dimensional vector (number of available GPUs, available memory size, and available CPUs) [27].

However, a notable limitation of this work lies in the ambiguous definition of job representation. The authors describe each job $J$ only as a matrix containing at least the job identifier, expected duration, arrival time, completion time, and resource requirements [27].

In particular, the inclusion of completion time in the job state is problematic, as it is not available for jobs that have not yet been scheduled or executed. This raises concerns about the consistency of the state representation.

After all, given the cluster state, and the states of jobs in the waiting queue as inputs, the optimization objective is to minimize both JCT and slowdown [27], where slowdown is defined as the ratio between the actual elapsed time (including queuing delay) and the theoretical shortest execution time. Therefore, the overall reward function is defined as

$$\text{Reward} = \sum_j (f_j - r_j) \; + \; \alpha \sum_j \frac{f_j - r_j}{e_j} \text{ [27]},$$

where $f_j$ denotes the completion time of job $j$, $r_j$ represents its arrival time, $e_j$ is the execution time (theoretical shortest time), and $\alpha$ is a weighting coefficient.

Through experiments on both a randomly generated dataset and the Alibaba cluster trace [14], it was shown that the RL-based approach using Policy Gradient demonstrates the most robust performance, consistently outperforming baselines such as Shortest Job first and the more advanced heuristic algorithms, such as Tiresias [27, 16].

However, the limitations of this early attempt are also evident. In addition to the aforementioned ambiguity in job characterization, the simulated server topology is overly

simplified and fails to capture the heterogeneity of GPU computational capabilities. Furthermore, the system explicitly assumes a non-preemptive setting, which limits its ability to adapt to dynamic workloads and reduces scheduling flexibility.

Previously, Ayadi *et al.* proposed a dual-agent DRL framework called **Priority-DRL**, designed to achieve deadline-aware and low-overhead preemptive scheduling [17], bearing more aspects compare to DeepGJS. Compared to **Chronus** (an earlier deadline aware algorithm), PriorityDRL aims to reduce computational cost while maintaining effective deadline and preemption handling. The architecture consists of two agents: a Job Selector (JS), which determines the priority of jobs in the queue, and a GPU Allocator (GA), which assigns high-priority jobs to specific GPU(s) [17].

The JS agent dynamically adjusts the job priority at each timestep based on the state of each job and the GPU cluster's topology matrix [17]. If a job is preempted during execution, its remaining time is increased by the time needed to save and restore the model, thereby modeling the preemption cost. In general, JS tends to assign higher priorities to near-completion jobs to reduce overall JCT. To account for deadlines, exponential and logarithmic reward shaping functions are introduced to assign higher rewards to urgent or overdue jobs.

The GA agent allocates GPUs to queued jobs in priority order, considering the GPU cluster's topology and communication costs. A three-tier communication cost model is used: $C_{\text{intra-machine}} < C_{\text{intra-rack}} < C_{\text{inter-rack}}$. For a job $j$ requiring $k$ GPUs, the GA computes a normalized communication cost penalty $R_{cc}(j)$ as follows:

$$R_{cc}(j) = \begin{cases} 0, & \text{if } k = 1 \\ 1 - \frac{C_{\text{comm}}(j) - \min_k}{\max_k - \min_k}, & \text{otherwise} \end{cases}$$

Adapted from [17].
Here, $\min_k$ and $\max_k$ refer to the minimum and maximum possible communication costs for a $k$-GPU job, assuming best-case (e.g., same machine) and worst-case (e.g., across racks) placements, respectively.

However, since job length are provided as input values [17], and the paper does not clarify whether the (remaining) execution time $\tau$ is normalized based on the heterogeneous computing capacities of GPUs, we assume that the framework operates under a homogeneous GPU cluster. One fundamental reason for this limitation is that the paper directly treats the estimated runtime of a job as its job length. Consequently, the job length implicitly depends on the computational capability of the GPUs in the cluster where the trace was collected. This design makes the workload representation tightly coupled with the underlying hardware performance, which limits the generalizability of the scheduling framework when deployed on clusters with different GPU configurations. Furthermore, such a representation makes it difficult to extend the approach to hetero-

geneous GPU environments, where the execution time of the same job vary significantly depending on the GPU model.

Additionally, although PriorityDRL explicitly encodes communication cost into the reward function to reduce fragmentation and improve job placement, its simulation environment does not incorporate communication delays into the actual job execution process. In other words, communication overhead is not reflected in the simulated JCT or deadline satisfaction. As a result, the framework was not able to capture the real-system impact of communication overhead, which can significantly influence both job completion time and the likelihood of meeting deadlines in the cluster.

In contrast, representing workloads in terms of computational demand (e.g., FLOPs) allows the scheduler to decouple job characteristics from specific hardware configurations.

### 2.7.3 Optimization-based methods in heterogeneous GPU cluster

In addition to heuristic-based and learning-based approaches, another line of work formulates scheduling as an explicit optimization problem. These methods differ fundamentally in that they aim to derive scheduling decisions by directly minimizing a predefined objective function under system constraints, rather than relying on handcrafted rules or learned policies.

A representative optimization-based approach is FFT [67], which targets efficient and fair scheduling of distributed deep learning workloads in heterogeneous GPU clusters. Unlike heuristic-based schedulers, FFT formulates the scheduling process as a global cost minimization problem, explicitly balancing JCT and long-term fairness.

At the core of FFT is a per-round resource allocation strategy, where the scheduler determines a resource allocation vector for all active jobs at each scheduling round. This vector specifies not only the number of GPUs assigned to each job but also the types of GPUs used, enabling fine-grained control over heterogeneous resources. Such a design allows FFT to capture the dynamic nature of cluster states while avoiding excessive reallocation overhead.

To address the inherent trade-off between efficiency and fairness, FFT introduces a fairness compensation mechanism that penalizes deviations from an ideal fair allocation. This mechanism is seamlessly integrated into the optimization objective, allowing the scheduler to jointly optimize system throughput and fairness in a unified framework. Furthermore, FFT incorporates a hierarchical, topology-aware placement strategy to mitigate communication overhead, ensuring that distributed training jobs are assigned to communication-efficient GPU configurations.

Overall, FFT demonstrates that optimization-based methods can effectively leverage fine-grained resource allocation and explicit objective design to achieve significant improvements in both JCT and fairness in heterogeneous GPU environments. However,

solving the underlying optimization problem (e.g., integer linear programming (ILP) or its relaxed variants) can incur substantial computational overhead, especially since such optimization must be performed at every scheduling round over all active jobs. This may limit the practicality and scalability of the approach in large-scale or highly dynamic cluster environments.

Moreover, FFT implicitly assumes that resource reallocation can be performed with negligible overhead. In practice, under distributed data parallel (DDP) training, adjusting the number of GPUs assigned to a job typically requires interrupting execution and reinitializing communication groups (e.g., all-reduce collectives), especially when checkpointing mechanisms are employed. This introduces non-trivial preemption and reconfiguration costs, which are not explicitly modeled in the framework.

# 3 Methodology

In this chapter, we present **SARL**, a Semantic-Aware Reinforcement Learning Scheduler for deep learning workloads in heterogeneous GPU clusters. We first formulate the scheduling problem and describe the system model. We then introduce the semantic-aware simulation environment. Finally, we present the reinforcement learning framework and training methodology.

## 3.1 Problem Definition

We consider the problem of scheduling deep learning training jobs in a heterogeneous GPU cluster. Jobs arrive dynamically over time and are placed into a waiting queue. Each job is characterized by a set of semantic attributes, including model type, dataset size, batch size, number of epochs, and GPU demand (i.e., the number of GPUs requested), which jointly determine its computational workload, memory requirement, and communication behavior. At each decision point, the scheduler selects a job from the queue and allocates a subset of available GPUs to it, subject to resource constraints. Unlike homogeneous settings, the execution time of a job depends not only on the number of assigned GPUs but also on their types and placement, due to heterogeneity in computational capability and communication bandwidth.

The scheduling problem is inherently challenging due to several factors. First, the system operates in a dynamic environment where job arrivals and resource availability evolve over time. Second, heterogeneous GPUs introduce non-uniform performance characteristics, making resource allocation decisions more complex. Third, preemption and job migration incur non-negligible overhead that depends on workload semantics, further complicating decision making.

From a system perspective, the scheduler aims to maximize resource utilization by efficiently assigning available GPUs. From a workload perspective, the objective is to minimize the deadline miss rate while reducing average JCT and maintaining reasonable waiting times.

We formulate this problem as a sequential decision-making process, where the scheduler must continuously make allocation decisions under evolving system states. The detailed system modeling and formulation are described in the following sections.

## 3.2 System Overview

Figure 2 illustrates the overall architecture of the proposed system. The system consists of two main components: a simulation environment and a reinforcement learning (RL) agent. The environment maintains the state of the heterogeneous GPU cluster and the job queue, including the semantic attributes of waiting jobs and the availability

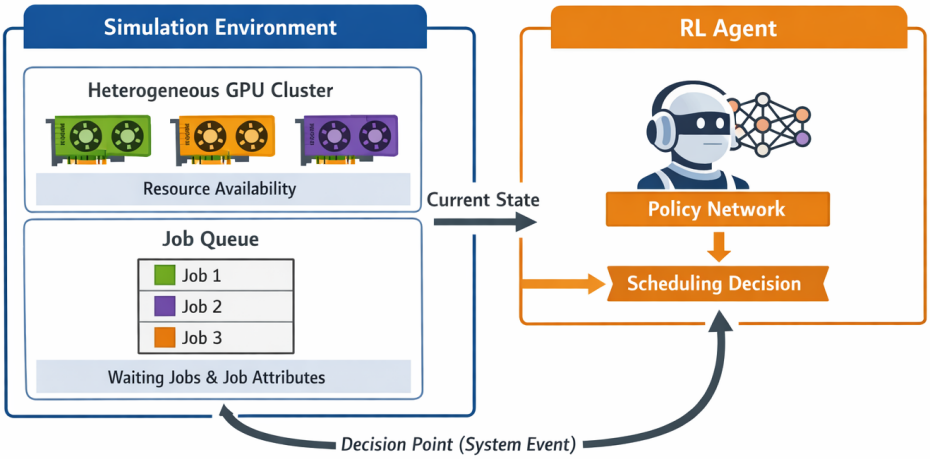



Figure 2: System architecture.

of GPU resources. Based on the observed system state, the agent makes scheduling decisions at each decision point, i.e., when significant system events occur (defined in Subsection *Event-Driven design*).

Specifically, the agent first encodes the system state using a Transformer-based feature extractor, followed by PPO policy and value networks implemented as multilayer perceptrons (MLPs). Based on this representation, the agent first selects a job from the job queue and then chooses a GPU model for allocation. Given the selected job and GPU model, the environment applies a topology-aware best-fit allocation strategy to assign concrete GPUs.

The allocation policy prioritizes GPUs within the same machine to reduce communication overhead and resource fragmentation. If this is infeasible, the allocator expands the search to machines within the same rack, and finally across racks when necessary.

The detailed GPU placement procedure is summarized in Algorithm 1. Given a job selected by the agent and a target GPU model, the allocator first identifies all feasible idle GPUs with sufficient memory under the selected model. It prioritizes allocations within the same machine; if this is infeasible, it expands the allocation scope to machines within the same rack, and finally across racks when necessary, in order to balance feasibility, communication overhead, and resource fragmentation.

We further enforce that each job is allocated GPUs of a single model and must obtain its required number of GPUs in a single allocation step. This design is consistent with common scheduling practices in modern GPU cluster schedulers, where jobs typi-

**Algorithm 1** Topology-Aware Best-Fit Allocation under a Selected GPU Model

**Require:** Job $j$, selected GPU model $m$
**Ensure:** Allocation result
$need \leftarrow$ number of GPUs required by $j$
$mem \leftarrow$ memory requirement of $j$
$candidates \leftarrow$ all idle GPUs of model $m$ with sufficient remaining memory
**if** $|candidates| < need$ **then**
    **return None**
**end if**
Group GPUs in *candidates* by machine
▷ First: exact match
**for** each machine $M$ **do**
    **if** $|M| = need$ **then**
        Select all GPUs from $M$
        **return** allocation result
    **end if**
**end for**
▷ Second: any feasible machine
**for** each machine $M$ **do**
    **if** $|M| > need$ **then**
        Select the first *need* GPUs from $M$
        **return** allocation result
    **end if**
**end for**
Group the candidate GPUs by rack
**for** each rack $R$ **do**
    **if** the total number of GPUs in $R$ is at least *need* **then**
        Sort machine in $R$ by available GPU count in descending order
        Initialize $selected \leftarrow \emptyset$
        **for** each machine $G$ in sorted order **do**
            Add as many GPUs as possible from $G$ to *selected*
            **if** $|selected| = need$ **then**
                **return** allocation result
            **end if**
        **end for**
    **end if**
**end for**
Sort all machines by available GPU count in descending order
Initialize $selected \leftarrow \emptyset$
**for** each machine group $G$ in sorted order **do**
    Add as many GPUs as possible from $G$ to *selected*
    **if** $|selected| = need$ **then**
        **return** allocation result
    **end if**
**end for**

cally require homogeneous resources and are not partially scheduled across heterogeneous devices [67].

This design allows the learning agent to focus on high-level scheduling decisions, while delegating low-level GPU placement to a lightweight heuristic. As a result, the action space is significantly reduced without ignoring cluster topology.

## 3.3 Cluster architecture

We consider a GPU cluster composed of multiple racks, where each rack contains several machines, and each machine is equipped with multiple GPUs. GPUs within the same machine are assumed to be homogeneous, while different machines may host different types of GPUs, resulting in a heterogeneous cluster environment.

For illustration, Fig. 3 shows an example of a heterogeneous GPU cluster. The cluster consists of three racks (Rack 1, Rack 2, and Rack 3), each containing a different number of machines.

Specifically, Rack 1 contains three machines (Machine 1, Machine 2, and Machine 3), Rack 2 contains two machines (Machine 4 and Machine 5), and Rack 3 contains one machine (Machine 6).

Within the same machine, GPUs are assumed to be homogeneous, whereas different machines may be equipped with different GPU models, reflecting hardware heterogeneity across the cluster.

In this example, Machine 1 is equipped with eight A100_80GB GPUs (GPU 1 to GPU 8), Machine 2 and Machine 3 are each equipped with eight H100_80GB GPUs (GPU 9 to GPU 16 and GPU 17 to GPU 24, respectively). Machine 4 and Machine 5 are equipped with four and six V100_16GB GPUs (GPU 25 to GPU 28 and GPU 29 to GPU 34), respectively, while Machine 6 contains three A100_80GB GPUs (GPU 35 to GPU 37).

This example highlights both the hierarchical organization of the cluster (rack $\rightarrow$ machine $\rightarrow$ GPU) and the heterogeneity in GPU types and capacities across different machines.

We assume a unified communication framework across the cluster, allowing distributed training jobs to span multiple GPUs across machines and racks. However, the performance of such distributed execution depends on the underlying topology and resource allocation.

## 3.4 Job Model

As time progresses, deep learning training jobs arrive dynamically and are placed into a job queue. The job queue consists of multiple jobs with diverse resource demands and execution characteristics, reflecting the heterogeneity of real-world workloads.

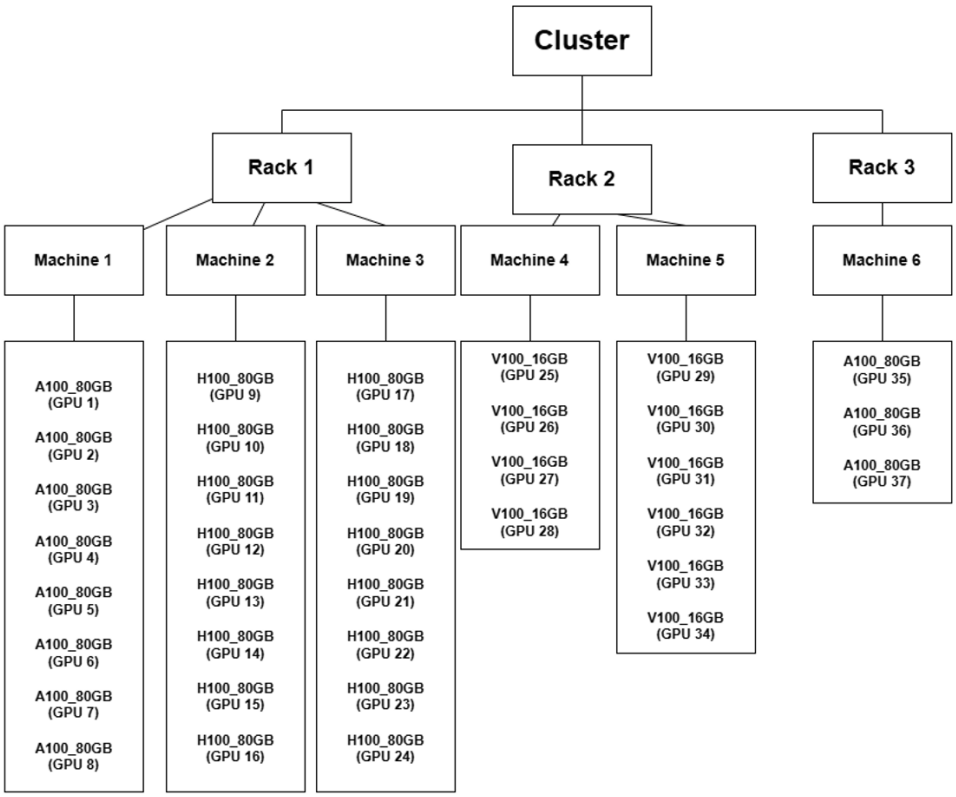



Figure 3: An example of a heterogeneous GPU cluster organized hierarchically into racks, machines, and GPUs. Different machines may host different GPU models, resulting in heterogeneity in both computational capability and memory capacity.

Each job is associated with a set of semantic attributes specified at submission time, including the model type, batch size, number of training samples (or sequences for language models), and number of epochs. These attributes collectively determine the computational cost and memory requirements of the job.

We refer to these attributes as the *semantic information* of the workload. In practice, additional parameters such as random seed, learning rate, and dataset-specific configurations (e.g., number of classes or class labels) may also be specified by users. However, these parameters have negligible impact on training time and resource consumption, and are therefore omitted in our abstraction.

In our simulation environment, we consider four representative model types, each paired with a corresponding dataset, as summarized in Table 1. These models are selected as representative deep learning workloads for several reasons. First, they cover diverse application domains, including image classification (ResNet18, Inception, VGG16), natural language processing (GPT-2), and speech recognition (DeepSpeech-2), thereby reflecting the heterogeneity of real-world training workloads. Second, these models are well-established and widely adopted in prior literature, making them suitable benchmarks for evaluating scheduling strategies. Third, and most importantly, all selected models are publicly available and open-sourced, which enables reproducibility and facilitates esti-

Table 1: Model and Dataset Mapping

| **Model** | **Dataset** |
|---|---|
| ResNet18 [68] | ImageNet [12] |
| Inception [69] | ImageNet [12] |
| VGG16 [70] | ImageNet [12] |
| GPT-2 [71] | IMDB50K [72] |
| DeepSpeech-2 [73] | LibriSpeech [74] |

mation of computational cost (e.g., FLOPs and memory consumption) through profiling tools.

We assume that all input data has been properly preprocessed (e.g., converted into efficient formats such as `.pt` for PyTorch-based models), and therefore the data I/O cost is not considered in our model.

It is important to note that key execution characteristics, such as job length and memory consumption, are not explicitly specified at submission time. In practice, users often have limited knowledge of these properties before execution. This motivates the need for a profiling mechanism to estimate job length behavior based on workload semantics.

### 3.4.1 Profiler

To quantify the computational complexity of different input jobs, we develop a profiling mechanism to estimate their workload. To provide a unified metric that is applicable across heterogeneous GPU clusters, we measure job length in terms of FLOPs, rather than execution time.

**job length estimation** For CNN-based and deepspeech2 models, including ResNet, VGG, and Inception, we estimate the computational cost using the THOP profiling tool [75]. Specifically, given a model and an input tensor, THOP computes the number of multiply-accumulate operations (MACs), from which the forward-pass FLOPs are approximated as twice the MACs, as summarized in Algorithm 2. When estimating training cost, we follow the common assumption that the backward pass incurs approximately twice the computational cost of the forward pass. As a result, the total training FLOPs per iteration are estimated as three times the forward-pass FLOPs. Let $N$ denote the number of training examples and $E$ the number of epochs. The total computational cost of a CNN-based job is then estimated as:

$$\text{FLOPs}_{\text{CNN\_job}} = 3 \times \text{FLOPs}_{\text{forward}} \times N \times E.$$

An example profiling procedure is illustrated in Algorithm 2. For large language

**Algorithm 2** FLOPs Estimation for CNN Models

**Require:** Model architecture $M$, input tensor shape $S$
**Ensure:** Estimated FLOPs
1: Initialize model $M$ on device
2: Generate dummy input tensor $x \sim \mathcal{N}(0, 1)$ with shape $S$
3: Compute MACs using profiling tool:
4: MACs $\leftarrow$ Profile$(M, x)$
5: Convert to FLOPs:
6: FLOPs $\leftarrow 6 \times$ MACs
7: **return** FLOPs

models (LLMs), we estimate the computational cost based on the scaling law proposed by Kaplan et al. [29]. Specifically, the total training FLOPs can be approximated as:

$$\text{FLOPs} \approx 6MD,$$

where $M$ denotes the number of model parameters, and $D$ represents the total number of tokens processed during training.

The number of tokens $D$ is given by:

$$D = L \times B \times T,$$

where $L$ is the sequence length, $B$ is the batch size, and $T$ is the number of training steps.

The number of training steps can be expressed as:

$$T = \frac{N_{\text{seq}} \times E}{B},$$

where $N_{\text{seq}}$ is the number of training sequences and $E$ is the number of epochs.

By substitution, the total computational cost of an LLM training job can be expressed as:

$$\text{FLOPs}_{\text{LLM\_job}} \approx 6M \times L \times N_{\text{seq}} \times E.$$

For example, for GPT-2 XL with approximately 1.5 billion parameters, the total training cost can be estimated as:

$$\text{FLOPs}_{\text{job}} \approx 6 \times 1.5\text{B} \times L \times N_{\text{samples}} \times E.$$

**Memory Estimation** To estimate the memory consumption of different workloads, we perform offline profiling using a reference GPU (NVIDIA A40 with 48 GB memory). For each model, we measure the peak GPU memory usage during training under different batch sizes.

Specifically, for a given model and batch size, we simulate one training iteration,

**Algorithm 3** Memory Estimation via Offline Profiling

**Require:** Model $M$, batch size $B$, input shape $S$, GPU device
**Ensure:** Estimated peak memory usage
Initialize model $M$ on GPU in training mode
Generate input tensor $x$ with shape $(B, S)$
Generate target labels $y$
Reset GPU memory statistics
Perform forward pass: $o \leftarrow M(x)$
Compute loss: $\ell \leftarrow \text{Loss}(o, y)$
Perform backward pass: $\ell$.backward()
Record peak memory usage:
    $m \leftarrow$ MaxMemoryAllocated()
**if** $B$ exceeds GPU memory capacity **then**
    Fit regression model using previous $(B, m)$ pairs
    Estimate memory usage via regression
**end if**
**return** $m$

including forward propagation, loss computation, and backward propagation, and record the peak memory allocation. This process captures the dominant memory usage during training.

When the target batch size exceeds the memory capacity of the profiling GPU, direct measurement becomes infeasible. In such cases, we construct a regression model based on previously observed data points to extrapolate memory consumption for larger batch sizes. This approach enables scalable and model-aware estimation of memory requirements, which is critical for scheduling in heterogeneous GPU environments.

For large language models such as GPT-2, we adopt a similar profiling strategy to estimate memory consumption. Specifically, we simulate a single training iteration with given batch size and sequence length, and record the peak GPU memory usage. For speech recognition models such as DeepSpeech2, we follow the same profiling approach by simulating training iterations with varying batch sizes and input sequence lengths (i.e., audio durations), and measuring the corresponding peak memory usage. Together, the proposed profiling methods provide a unified estimation of both computational and memory requirements for different types of workloads.

**Submission time** Each job is also associated with a submission time, which records the time at which the job is submitted to the cluster.

**Deadline** In addition to workload characterization, we assign a deadline to each job upon submission. While deadlines are typically specified by users in real-world systems, we generate them automatically based on the estimated job length.

Specifically, we first compute the total computational cost of a job in FLOPs using

the proposed profiler. The baseline deadline is then estimated by normalizing the job length with respect to the average computational capacity of the cluster:

$$T_{\text{base}} = \frac{\text{FLOPs}_{\text{job}}}{C_{\text{avg}}},$$

where $C_{\text{avg}}$ denotes the average computational throughput (in FLOPs/s) across all GPUs in the cluster. To ensure fairness, $C_{\text{avg}}$ is computed from the static hardware configuration and does not depend on the current system load. This design prevents later-arriving jobs from being assigned excessively long deadlines due to transient system congestion, which would otherwise bias the evaluation against jobs submitted under high load conditions.[1]

To model varying levels of job urgency and to avoid a strict linear correlation between job length and deadline, we further introduce a random scaling factor:

$$T_{\text{deadline}} = \alpha \cdot T_{\text{base}}, \quad \alpha \sim \mathcal{U}(0.7, 1.5).$$

This design enables the generation of realistic and diverse deadlines while preserving the relative difficulty of scheduling decisions.

## 3.5 Communication Cost

In distributed training, when a job utilizes multiple GPUs, communication overhead arises due to synchronization operations such as all-reduce. The communication cost depends on both the placement of GPUs and the workload characteristics.

It is widely observed that communication latency increases with the physical distance between GPUs [15]. This effect is particularly pronounced in hierarchical cluster architectures, where communication performance differs across intra-machine, intra-rack, and cross-rack placements.

**Placement-Aware Communication Efficiency** When a job is distributed across multiple GPUs, synchronization and data exchange introduce communication overhead, which reduces the amount of useful computation completed within a given simulation interval. Rather than explicitly simulating communication time from message size and link bandwidth, we approximate this effect using a placement-dependent communication efficiency factor $\eta \in (0, 1]$.

Let $G$ denote the set of GPUs allocated to a job. The communication-adjusted computation completed during a simulation interval $\Delta t$ is defined as

[1] In our simulation, each time step corresponds to one second of simulated time, which is aligned with one second of real-world execution time.

$$F_{\text{step}} = \left( \sum_{g \in G} \text{FLOPs}_g \right) \cdot \Delta t \cdot \eta \tag{1}$$

where $\text{FLOPs}_g$ denotes the peak computational throughput of GPU $g$, and $\Delta t$ is the simulated time interval in seconds. Here, $\eta$ captures the reduction in effective throughput caused by inter-GPU communication.

The communication efficiency factor $\eta$ is determined by the placement of the allocated GPUs:

$$\eta = \begin{cases} 1.0, & |G| = 1 \\ \eta^{\text{type}}_{\text{same\_machine}}, & \text{if GPUs are on the same machine} \\ \eta^{\text{type}}_{\text{same\_rack}}, & \text{if GPUs span multiple machines within the same rack} \\ \eta^{\text{type}}_{\text{cross\_rack}}, & \text{if GPUs span multiple racks} \end{cases}$$

In the current simulator, communication overhead is approximated at a coarse placement level. In particular, allocations spanning multiple machines within the same rack share the same communication efficiency factor, regardless of the exact number of machines involved. This simplification is adopted for tractability, although in real distributed training, collective communication cost may further increase with the number of machines participating in the job.

**Workload-Aware Communication Modeling** To further capture workload characteristics, we define model-specific communication efficiency coefficients

$$\eta_{\cdot}^{\text{type}} \in \{\eta_{\text{same\_machine but different machines}}, \eta_{\text{same\_rack}}, \eta_{\text{cross\_rack}}\},$$

which vary across model types. Larger models typically incur higher communication overhead due to increased parameter synchronization and communication volume.

These coefficients are not directly measured from a specific hardware platform. Instead, they are coarse-grained simulator parameters calibrated from communication trends reported in prior work. In particular, [19] shows how the ratio between exposed communication cost and computation time varies across model types under different bandwidth settings. To relate these bandwidth-dependent trends to our simulator, we further draw on the topology-aware discussion in [15], which highlights that communication cost depends strongly on placement locality, such as whether GPUs are placed within the same machine, across machines in the same rack, or across racks. Guided by these two observations, we approximately associate our placement categories with different effective bandwidth regimes and assign representative communication efficiency coefficients

accordingly.

For illustration, we use the following representative settings in the simulator:

- CNN-based models (e.g., ResNet, VGG), which typically exhibit relatively lower communication overhead:

$$\eta_{\text{same\_machine}} \approx 0.95, \quad \eta_{\text{same\_rack but different machines}} \approx 0.80, \quad \eta_{\text{cross\_rack}} \approx 0.70$$

- Large language models (e.g., GPT), which are more communication-intensive:

$$\eta_{\text{same\_machine}} \approx 0.85, \quad \eta_{\text{same\_rack but different machines}} \approx 0.70, \quad \eta_{\text{cross\_rack}} \approx 0.60$$

These values should be interpreted as approximate simulator settings rather than exact hardware-specific measurements. They are chosen to reflect two qualitative effects observed in prior work: (i) communication efficiency decreases as placement locality becomes weaker, and (ii) larger models tend to expose a larger fraction of communication overhead relative to computation.

**Comparison with Prior Modeling Approaches** Compared to prior work that considers communication cost in GPU job scheduling [17, 15], our communication modeling differs in two key aspects. First, beyond capturing placement-dependent communication effects (e.g., intra-machine, intra-rack, and cross-rack), we additionally incorporate workload characteristics by introducing model-specific communication efficiency factors. This allows larger models to incur proportionally higher communication overhead, reflecting their increased synchronization cost. Second, we incorporate communication cost into the job throughput computation. As shown in Eq. (1), communication efficiency $\eta$ determines the amount of computation completed within each simulation interval, thereby directly influencing the training speed of each job. This design enables communication cost to be naturally propagated to higher-level system metrics, such as JCT and deadline satisfaction, resulting in a more faithful and unified modeling of system performance.

## 3.6 Preemption Cost

We define preemption as the action of interrupting a running job and returning it to the job queue, transitioning its state from *running* to *queued*. While preemption improves scheduling flexibility, it introduces additional computational overhead.

We assume that the system employs checkpointing to support preemption. When a job is preempted, it loses all progress made since the last checkpoint and must resume from the most recent saved state. This behavior is consistent with prior observations in [16],

which show that preemption involves both system recovery and checkpoint reloading overhead.

**Preemption Penalty Formulation** We model the preemption penalty as an increase in the remaining computational workload (measured in FLOPs). Specifically, when a job $j$ is preempted at time $t$, its total required computation is increased by an additional amount $\Delta F_j$, defined as:

$$\Delta F_j = \underbrace{\alpha \cdot f_j(t)}_{\text{recovery overhead}} + \underbrace{\beta_j \cdot f_j(t)}_{\text{rollback (checkpoint loss)}}$$

where:

- $f_j(t)$ denotes the amount of FLOPs processed by job $j$ in the last simulation step,
- $\alpha$ is a constant factor representing system-level recovery overhead (e.g., process restart and checkpoint loading),
- $\beta_j$ is a workload-dependent factor representing the expected rollback due to checkpointing.

The first term captures the cost of system interruption and recovery, which is relatively small. The second term models the loss of training progress since the last checkpoint, which dominates the preemption cost.

**Workload-Dependent Modeling** We model $\beta_j$ based on workload characteristics, particularly the checkpointing behavior during training. Checkpointing introduces overhead proportional to model size, as both saving and loading model states involve significant data movement. As a result, larger models typically checkpoint less frequently to amortize this cost, while smaller models can afford more frequent checkpointing.

To capture this effect, we model $\beta_j$ as the expected interval (in simulation steps) between two consecutive checkpoints. Since each simulation step corresponds to one second of training time, $\beta_j$ can be interpreted as a time-based checkpoint interval.

For example:

- For CNN-based models such as VGG:

  $$\beta_j \approx 600$$

  corresponding to checkpointing approximately every 10 minutes.

- For large language models such as GPT:

  $$\beta_j \approx 2100$$

corresponding to checkpointing approximately every 35 minutes.

This setting reflects the trade-off between checkpoint overhead and recovery cost observed in practice. In particular, larger models tend to adopt less frequent checkpointing strategies due to their higher I/O and memory overhead, which is consistent with observations in large-scale training systems [20].

**Implementation** In the simulator, this penalty is implemented by increasing the total required FLOPs of the job:

$$F_j \leftarrow F_j + \Delta F_j$$

which directly delays the job's completion time.

**Integration into Reward** The preemption penalty is implicitly incorporated into the reward function through its impact on job completion time and deadline satisfaction.

Since the reward is defined based on system performance metrics such as JCT and deadline miss rate, any increase in workload due to preemption leads to:

- longer job completion times,
- increased waiting time for subsequent jobs,
- higher likelihood of deadline violations.

Therefore, excessive or unnecessary preemption results in lower cumulative reward, encouraging the agent to balance the benefit of preemption against its cost.

In addition, we include a lightweight explicit penalty for preemption actions in the reward design to further discourage frequent interruptions. The detailed formulation of the reward function is described in Section *Reward Function*.

## 3.7 RL-based Scheduling Model

We model the scheduling problem as a sequential decision-making process and adopt a RL approach to learn an adaptive scheduling policy. Specifically, we employ Masked Proximal Policy Optimization (Maskable PPO) to handle the dynamic and constrained action space.

### 3.7.1 Observation Space

At each decision point, the agent observes a representation of the current system state, which consists of three components: (1) the cluster resource state, (2) Total number of idle GPUs of each model, and (3) the job queue state. The job queue state encodes

the semantic attributes of pending jobs, while the cluster state captures the availability and characteristics of GPUs across the system.

These three components are concatenated and flattened into a fixed-length vector to form the final observation:

$$s_t = \text{Flatten}(\text{ClusterState} \oplus \text{idle\_GPU\_permodel} \oplus \text{JobQueueState}).$$

**Cluster Resource State** We represent the cluster resource state as a matrix

$$C \in \mathbb{R}^{M \times d_c},$$

where $M$ is the number of machines in the cluster and $d_c$ is the feature dimension of each machine. Each row of $C$ corresponds to a machine and encodes its current resource status.

For each machine $m$, we extract a feature vector $\mathbf{f}(m)$ that summarizes its resource characteristics, including:

- **Available GPUs:** the number of idle GPUs on the machine.
- **Busy GPUs:** the number of GPUs currently occupied by running jobs.
- **Computational capability:** the normalized peak FLOPs of the GPU type, using a reference GPU as baseline.
- **Memory availability:** the average and minimum of per-GPU memory availability ratios within the machine, where each ratio is computed as the avaliable memory normalized by the GPU's total capacity, reflecting per-GPU (non-shared) memory constraints.For example, consider a machine with three A100 GPUs (each with 80GB memory), where one GPU is fully occupied and the other two are idle. The corresponding per-GPU memory availability ratios are 0.0, 1.0, and 1.0. The resulting average and minimum values are 0.67 and 0.0, respectively, reflecting both overall availability and the most constrained GPU.
- **GPU model indicator:** a one-hot encoding representing the GPU model type of the machine (e.g., H100, A100, V100).

The cluster state matrix $C$ is constructed by stacking the machine-level feature vectors across all racks:

$$C = \begin{bmatrix} \mathbf{f}(m_1) \\ \mathbf{f}(m_2) \\ \vdots \\ \mathbf{f}(m_M) \end{bmatrix}.$$

To interface with the learning model, $C$ is flattened and concatenated with other components of the observation.

**Job Queue State** The job queue state is represented as a matrix

$$J \in \mathbb{R}^{X \times d_j},$$

where $X$ is the maximum number of jobs considered in the queue, and $d_j$ is the number of features used to describe each job. Since the actual number of jobs in the queue may vary over time, we fix the queue length to $X$ by convention. If the number of jobs is smaller than $X$, the remaining rows are padded with zeros.

Jobs included in the observation are ordered by submission time, so that earlier submitted jobs appear first. Each row of $J$ corresponds to one job and contains the following features:

- **GPU demand:** the total number of GPUs required by the job.
- **Waiting time:** the amount of time the job has spent waiting in the system, normalized by a fixed time scale.
- **Remaining time to deadline:** the remaining time until the job reaches its assigned deadline, normalized by a predefined scale.
- **Job type:** a one-hot encoding indicating the workload type (e.g., ResNet, VGG, Inception, GPT-2 and DeepSpeech-2).
- **Job status:** a one-hot encoding indicating whether the job is queued or running.
- **Estimated computational cost:** the normalized FLOPs of the job, as estimated by the profiler.
- **Estimated memory requirement:** the normalized GPU memory demand of the job.
- **Estimated remaining execution time:** the normalized estimated time required for the job to finish under its current allocation, computed from the remaining training FLOPs and the job's recent effective throughput. For jobs that are not currently running, this value is set to a large predefined constant ($\texttt{MAX\_EST\_TIME} = 10^6$) to indicate that no reliable throughput-based estimate is available.
- **Preemption count:** the number of times the job has been preempted.
- **Original deadline:** the initial deadline assigned to the job before any subsequent updates.

The job queue observation is finally flattened into a one-dimensional vector before being concatenated with the cluster state representation.

### 3.7.2 Hybrid Event-Driven Scheduling

To avoid unnecessary decision making at every simulation step, which may lead to sparse reward signals and increased training overhead, we adopt a hybrid scheduling mechanism that combines event-driven triggers with periodic reconsideration.

Instead of invoking the agent at every simulation step, decisions are made only at specific decision points. Most decision points are triggered by meaningful system events, while an additional periodic trigger is introduced to maintain responsiveness when no such event occurs for an extended period. Between decision points, the system evolves without intervention from the agent. This design reduces inference overhead while still allowing the scheduler to react to important system changes in a timely manner.

We define a decision point as the occurrence of one of the following:

- **Job arrival:** a new job is submitted to the cluster.
- **Job completion:** a running job finishes and releases resources.
- **Periodic trigger:** a scheduling decision is invoked at fixed time intervals when no arrival or completion event occurs for some time. This is included to ensure that the agent can periodically reconsider the current allocation, especially in long-running phases where the system state may remain unchanged in terms of arrivals and completions, but rescheduling or preemption may still be beneficial.

### 3.7.3 Action Space

At each decision point, the scheduler follows a two-stage decision process consisting of optional preemption and a final scheduling action.

**Preemption Stage** Before making a scheduling decision, the agent may optionally perform a sequence of preemption actions. Each preemption action selects a currently running job, terminates it, and returns it to the job queue. Preemption is treated as a "free" operation at the decision level, which will not end this decision point, allowing the agent to preempt multiple jobs within the same decision point if necessary.

**Scheduling Stage** After completing preemption actions, the agent selects one of the following final actions:

- **Schedule:** select a job from the queue and assign it a GPU model for allocation. The environment then performs concrete GPU placement using the heuristic topology-aware allocation policy.

- **No-Op:** No-Op: take no action and allow the system to evolve until the next event. This action is always available to the agent, regardless of whether a preemption has occurred earlier in the decision process. If a job is preempted in the preemption stage, it is immediately returned to the job queue and becomes visible again in the agent's observation. Therefore, in principle, the agent may choose to schedule that job again in the subsequent scheduling stage. However, the agent may also select No-Op, meaning that a preemption can indeed be followed by no new scheduling action in the same decision cycle.

Since the action space is composed of concrete job-specific decisions, not every action entry is valid at every decision point, we employ action masking to restrict the policy to feasible actions only. For example, an action that schedules job $j$ to a specific GPU tier is invalid if job $j$ is not currently in the queue, if the selected tier does not have enough available GPUs, or if the GPUs in that tier cannot satisfy the job's memory requirement. Similarly, a preemption action is invalid if the corresponding job is not currently running. By masking out such invalid action entries, the agent is prevented from selecting infeasible decisions, while valid actions such as feasible scheduling choices and *No-Op* remain available. This significantly improves training stability and sample efficiency.

### 3.7.4 Reward Function

The reward function is designed to guide the agent toward minimizing deadline violations while maintaining efficient scheduling behavior.

**Action-Level Signals** To encourage active scheduling behavior, we introduce a small positive reward for each scheduling action. Since infeasible actions are filtered out by action masking, every scheduling action corresponds to a valid allocation. We assign a constant reward:

$$r_{\text{schedule}} = +0.05$$

This value is intentionally kept small (approximately 1% of the typical job-level reward) to avoid dominating the learning signal. Its purpose is to provide a mild incentive for the agent to utilize available resources without overshadowing the primary optimization objectives.

For preemption, we introduce a lightweight penalty that increases with the number of times a job has been preempted:

$$r_{\text{preempt}} = -0.05 \cdot \min(n_j, N_{\max})$$

where $n_j$ denotes the number of times job $j$ has been preempted, and $N_{\max} = 5$ is a clipping threshold to prevent excessively large penalties.

This penalty is also designed to be small relative to job-level rewards. Its role is to regularize the agent's behavior by discouraging excessive or unnecessary preemptions during exploration.

We note that the primary cost of preemption is already modeled in the environment through increased computational workload (see Section *preemption cost*), which directly affects job completion time and deadline satisfaction. The explicit penalty here serves only as an auxiliary signal to stabilize training and guide exploration.

**Completion-Based Reward** The primary reward signal is defined upon job completion, based on whether the job meets its deadline.

Let $T_d$ denote the job deadline and $T_c$ denote the completion time. We define the overdue ratio as:

$$r_{\text{overdue}} = \frac{T_c - T_d}{T_d}.$$

The completion-based reward is defined as:

$$r_{\text{completion}} = \begin{cases} +5, & \text{if } T_c \leq T_d \\ -\phi(r_{\text{overdue}}), & \text{if } T_c > T_d \end{cases}$$

where $\phi(\cdot)$ is a piecewise penalty function:

$$\phi(r) = \begin{cases} 2r, & 0 < r < 0.2 \\ 4r, & 0.2 \leq r < 0.5 \\ 6r, & 0.5 \leq r < 0.75 \\ 8r, & 0.75 \leq r < 1 \\ \min(9r, 12), & r \geq 1 \end{cases}$$

This design ensures that the penalty increases monotonically with the overdue ratio, and grows more aggressively as the delay becomes larger.

The use of piecewise scaling allows the reward function to distinguish between mild and severe deadline violations, while clipping prevents excessively large penalties that could destabilize training.

**Overall Reward** The final reward at each decision step is defined as the sum of the action-level signals and the completion-based reward:

$$r_t = r_{\text{schedule}} + r_{\text{preempt}} + r_{\text{completion}},$$

where $r_{\text{schedule}}$ and $r_{\text{preempt}}$ are applied when the corresponding actions are taken, and $r_{\text{completion}}$ is issued upon job completion events.

**Design Rationale** The reward function prioritizes deadline satisfaction as the primary objective, while implicitly promoting lower JCT. By penalizing the magnitude of delay rather than using a binary penalty, the agent is encouraged to complete overdue jobs as early as possible. This design provides richer learning signals and empirically leads to improved JCT and waiting time performance.

### 3.7.5 Policy Network

To effectively represent the structured state in the scheduling environment, we employ a Transformer-based feature extractor to encode the raw observation into a compact latent representation for reinforcement learning. As illustrated in Figure 4, the environ-

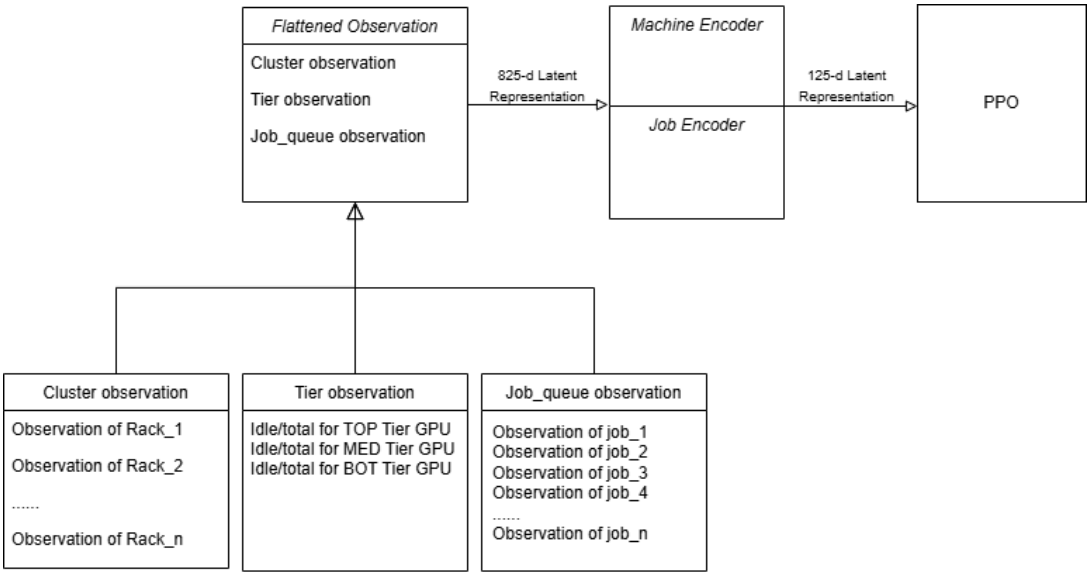



Figure 4: Overview of the observation encoding pipeline. Raw cluster-, model-, and job-level features are concatenated into a flattened vector and encoded by a transformer-based dual-tower into a compact latent representation for PPO policy and value estimation.

ment state consists of three semantically distinct components: cluster observation, model observation, and job-queue observation. The cluster observation captures the current resource status of the system, the model observation provides global information about resource availability across different GPU models, and the job-queue observation describes the candidate jobs waiting to be scheduled. These components are concatenated into a flattened observation vector for input to the policy.

Although this flattened representation is convenient for standard RL pipelines, it obscures the internal structure of the state. In particular, machine states form a set of resource-bearing entities, while queued jobs form a set of candidate entities competing

for resources. A naive flatten-and-MLP design treats all input dimensions uniformly and does not explicitly exploit such structure, requiring the model to implicitly learn relationships such as inter-machine dependencies, inter-job prioritization, and job–resource compatibility from unstructured inputs.

To address this limitation, we design a Transformer-based feature extractor with separate machine and job towers. Machine-level observations are reshaped into machine tokens and encoded by one Transformer tower, while job-level observations are reshaped into job tokens and encoded by another. Since the number of observable jobs may vary across states, padded job tokens are masked during encoding.

To further incorporate resource awareness into job representations, we apply a cross-attention block in which job tokens attend to the encoded machine tokens. This allows each job representation to integrate information about the current machine states and thus capture job–resource compatibility more explicitly.

The encoded machine tokens are aggregated through attention pooling to obtain a machine-side context vector, while the job-side representations are aggregated using masked mean pooling to produce a job-side context vector. In parallel, the global GPU-model observation is encoded by a lightweight MLP. These three context vectors are then augmented with learnable type embeddings, concatenated, and projected into a fixed-dimensional latent representation.

This latent representation is subsequently passed to the PPO policy and value networks, both implemented as MLPs. The actor and critic share the same encoded state representation but use separate fully connected heads. The policy head outputs action probabilities, while the value head estimates the state value. In this way, the Transformer module serves as a structured feature extractor, while the downstream PPO networks remain lightweight and efficient.

### 3.7.6 Supervised Pretraining

To improve training stability and reduce variance across random seeds, we initialize the PPO agent with a supervised pretraining stage before standard reinforcement learning. Rather than starting from a fully random policy, this stage provides the agent with a task-relevant initialization based on heuristic demonstrations.

In our scheduling environment, purely random early-stage exploration can be highly unstable, since scheduling decisions are sequential and their effects accumulate over time. Moreover, the valid action set is dynamically constrained by action masks, which further increases the difficulty of effective exploration at the beginning of training. To alleviate this issue, we adopt a teacher-guided warm-start strategy.

Specifically, we use EDF as the teacher policy. EDF is chosen because it serves as a strong heuristic baseline in our environment and provides reasonable scheduling decisions

without additional learning. We run multiple rollout episodes using EDF and collect state–action pairs together with their corresponding action masks. Each training sample therefore contains the observation, the teacher-selected action, and the set of valid actions at that decision step.

Using this dataset, we pretrain the policy network in a supervised manner. Given an observation, the agent first produces latent features through the proposed Transformer-based feature extractor, followed by action logits from the PPO policy head. Invalid actions are then masked out, and the model is trained to predict the teacher action using a cross-entropy loss over the valid action space. In this way, the policy learns to imitate EDF under the same masked action constraints used during RL training.

The purpose of this stage is not to force the final policy to mimic EDF exactly, but to provide a stronger initialization for subsequent PPO optimization. After pretraining, the agent continues interacting with the environment and is further optimized using standard PPO updates, allowing it to refine or deviate from the teacher policy when beneficial.

### 3.7.7 Training Details

After supervised pretraining, the agent is further optimized using Maskable PPO with generalized advantage estimation (GAE). The PPO agent takes as input the latent state representation produced by the proposed Transformer-based feature extractor, and optimizes separate MLP-based actor and critic heads. Training follows the standard PPO procedure, including rollout collection, mini-batch updates, and multiple optimization epochs for each policy update.

To stabilize training, we employ standard techniques including clipped policy updates, entropy regularization for exploration, and gradient norm clipping. The discount factor and GAE parameters are chosen to balance long-term credit assignment and variance in return estimation. Invalid actions are filtered through action masking, which ensures that the policy only assigns probability mass to feasible scheduling decisions during both training and inference.

## 3.8 Summary of the Proposed Framework

In this chapter, we presented a semantic-aware reinforcement learning framework for multi-GPU job scheduling. We first formulated the scheduling problem in a heterogeneous GPU cluster and described the design of the simulation environment. We then introduced a transformer-based dual-tower feature extractor to encode structured observations, along with a supervised pretraining strategy to improve training stability. Finally, we integrated these components into a PPO-based training pipeline.

In the next chapter, we evaluate the proposed framework against several baseline scheduling methods under diverse workload settings.

# 4 Experiments

We compare SARL with seven representative baseline methods from different categories. Specifically, four classical heuristic algorithms are included, namely FIFO, SJF, Backfill, EDF with preemption. In addition, we consider a modern GPU-cluster-oriented heuristic proposed by Gu et al. [16], a reinforcement learning (DRL)–based scheduler, DL2 [66], and an optimization-based method, Fast and Fair Training (FFT) [67].

We provide detailed descriptions of all baseline methods in the following section.

## 4.1 RL Hyperparameter Settings

The proposed scheduler is implemented in PyTorch and trained using MaskablePPO, which enables dynamic action masking to exclude invalid scheduling decisions. Given the large discrete action space induced by joint job selection and GPU allocation, action masking plays a critical role in improving learning efficiency.

The policy adopts an actor–critic architecture, where the policy (actor) and value (critic) networks are parameterized by separate MLP. The actor network consists of three fully connected layers with sizes 1024, 1024, and 512, while the critic network uses three layers of size 512 to provide stable value estimation.

During training, we use a learning rate of $1 \times 10^{-4}$ and a rollout horizon of 3072 steps, with minibatch updates of size 256 over 5 epochs per iteration. The discount factor is set to 0.99, and GAE with $\lambda = 0.98$ is applied to reduce variance. An entropy regularization coefficient of 0.05 is used to encourage exploration, and gradient clipping with a maximum norm of 0.5 is applied to improve training stability.

We use a fixed time scale of 1800 seconds for normalization, which is also applied to features such as waiting time and periodic trigger.

All models are trained on GPU, and training progress is monitored using TensorBoard.

## 4.2 Environment

Our environment consists of a cluster and a job queue.

### 4.2.1 Cluster Configuration

We simulate a heterogeneous GPU cluster organized in a hierarchical structure consisting of racks, machines, and GPUs. Each cluster contains multiple racks, where each rack is composed of several machines, and each machine hosts a set of GPUs. This hierarchical design reflects the topology of modern data center clusters and enables modeling of placement-dependent communication overhead.

The cluster includes multiple GPU types with different computational and memory capabilities, including A100 (80GB), H100 (80GB), and V100 (16GB). GPUs are unevenly distributed across machines to emulate realistic hardware heterogeneity.

In our default configuration, the cluster consists of two racks with a total of eight machines and 46 GPUs. This configuration is informed by prior measurements of real-world GPU clusters [16], and represents a realistic small-to-medium scale deployment. Despite its moderate size, it captures essential system dynamics such as resource contention, heterogeneous device characteristics, and scheduling complexity, making it suitable for evaluating scheduling policies.

**Rack 1 (high-capacity homogeneous machines).** The first rack contains four high-capacity machines, each equipped with 8 GPUs of the same model:

- Machine 1: 8 × A100 (80GB)
- Machine 2: 8 × H100 (80GB)
- Machine 3: 8 × A100 (80GB)
- Machine 4: 8 × H100 (80GB)

This rack represents the high-performance portion of the cluster, where each machine provides a large number of homogeneous GPUs.

**Rack 2 (heterogeneous low-capacity machines).** The second rack contains four smaller and heterogeneous machines with varying GPU counts and types:

- Machine 5: 3 × A100 (80GB)
- Machine 6: 3 × H100 (80GB)
- Machine 7: 4 × V100 (16GB)
- Machine 8: 4 × V100 (16GB)

Compared to Rack 1, machines in Rack 2 have fewer GPUs (3–4 per machine) and include both modern (A100/H100) and legacy (V100) devices, resulting in a highly heterogeneous and non-uniform resource pool.

Overall, this configuration captures both: (i) high-capacity homogeneous nodes for efficient large-scale training, and (ii) fragmented heterogeneous resources that introduce realistic scheduling challenges such as resource fragmentation and device compatibility constraints.

To capture system-level effects, we distinguish between intra-machine, intra-rack, and inter-rack placements when allocating multiple GPUs to a job. Different placement

patterns incur different levels of communication overhead, reflecting the varying bandwidth and latency characteristics of NVLink, intra-rack interconnects, and cross-rack communication.

This configuration results in a moderately sized yet heterogeneous cluster, providing a realistic testbed for evaluating scheduling policies under both resource diversity and topology-aware constraints.

**GPU Modeling.** Each GPU in the cluster is characterized by a set of hardware attributes, including memory capacity, peak computational throughput, memory bandwidth, and interconnect bandwidth. Among these, we primarily focus on memory capacity and peak FLOPs, as they are the dominant factors affecting the performance of deep learning workloads.

Specifically, memory capacity determines whether a model can be accommodated on a given device and directly impacts batch size and training feasibility, while peak FLOPs provide an upper bound on the achievable computational throughput. These two factors are therefore explicitly incorporated into both job profiling and scheduling decisions.

The GPU configurations used in our simulator are based on publicly available specifications from NVIDIA, including A100 (Ampere), H100 (Hopper), and V100 (Volta) architectures [76, 77, 78].

Other attributes, such as memory bandwidth and NVLink interconnect bandwidth, are also included in the system model, but are primarily used to support future extensions (e.g., model affinity) and are not the main focus of this work.

### 4.2.2 Job Queue

Jobs are maintained in a priority queue with a fixed capacity $N$, where jobs are ordered by their submission time. This queue represents the set of jobs within a single episode, and $N$ can be interpreted as the total number of jobs generated during that episode.

At each decision step, the agent does not observe the entire job queue. Instead, we adopt a truncated observation mechanism, where only the first $K$ jobs (sorted by submission time) are exposed to the agent. In our implementation, $K$ is fixed to 50. This design reflects practical system constraints, where schedulers typically operate on a limited visible window of pending jobs.

When a job completes, its allocated resources are released and returned to the cluster. The completed job is then moved to a separate completed-job queue for evaluation purposes (e.g., computing performance metrics such as job completion time and deadline satisfaction). Meanwhile, new jobs from the tail of the queue are automatically promoted into the observable window, ensuring a continuous flow of scheduling decisions.

To evaluate the proposed scheduling framework, we next describe the synthetic workload used in our experiments.

## 4.3 Datasets

We construct a synthetic workload to emulate the characteristics of deep learning training jobs in GPU clusters. Each episode consists of 50 jobs, and the GPU demand of each job is sampled uniformly from 1 to 8 GPUs, covering the typical resource demand range observed in real-world cluster traces [14]. Job arrivals follow a stochastic process within a fixed time horizon of 36,000 seconds, resulting in diverse system load conditions.

The configuration of representative workloads is summarized in Table 2.

**Remark on model coverage.** While the main workload focuses on representative models such as CNNs and GPT-2 for consistency in workload calibration, recent large language models (e.g., Llama 2) are not included in the primary dataset. This is because, under our cluster configuration, the available GPU memory capacity limits the feasible batch size and sequence length for Llama 2. As a result, only small-scale configurations can be supported, which are not representative of typical real-world LLM workloads.

In practice, training or serving models at the scale of Llama 2 commonly relies on model parallelism to distribute parameters across multiple GPUs. Such setups introduce additional system-level factors, including inter-GPU communication overhead and pipeline or tensor parallel scheduling, which are not explicitly modeled in our current workload abstraction.

Including Llama 2 under these constraints would therefore introduce inconsistencies in workload characterization and affect the validity of performance comparisons. To maintain methodological consistency, we exclude Llama 2 from the main experiments. Nevertheless, for completeness, we provide an extended workload configuration including Llama 2 in Appendix C with a smaller setting of batch size and sequence length.

Table 2: Configuration of representative training workloads in the synthetic dataset.

| Job Type | Model | Dataset | Dataset Size | Batch Size | Epochs | Sequence Length / Input Length | GPUs |
|---|---|---|---|---|---|---|---|
| Image Classification | ResNet18 [68] | ImageNet [12] | $\leq$ 11,060,223 | 32–512 | 1–20 | N/A | 1–4 |
| Image Classification | Inception [69] | ImageNet [12] | $\leq$ 11,060,223 | 32–512 | 1–20 | N/A | 1–5 |
| Image Classification | VGG16 [70] | ImageNet [12] | $\leq$ 11,060,223 | 32–512 | 1–20 | N/A | 4–6 |
| Natural Language Processing | GPT-2 [71] | IMDB50K [72] | $\leq$ 50,000 | 2–128 | 1–20 | 64–128 tokens | 6–8 |
| Automatic Speech Recognition | DeepSpeech-2 [79] | LibriSpeech [74] | $\leq$ 104,014 | 8–128 | 5–20 | 10–20 s (audio) | 3–4 |

During training, the reinforcement learning agent interacts with the environment over a sequence of episodes generated on-the-fly. Rather than relying on a fixed dataset, each episode is newly sampled according to a predefined job distribution, which improves the generalization ability of the learned policy.

In practice, we train the agent for a maximum of 1500 episodes. To monitor training progress and perform model selection, we employ a checkpointing mechanism: every 15

episodes, the current model is saved and evaluated on a separate validation set.

The validation set consists of 4 fixed episodes, each containing 50 jobs sampled from the same distribution as the training workload. For each checkpoint, we compute the average deadline miss rate across the validation episodes. If the new model achieves a lower validation miss rate than the previously best-performing model, it replaces the previous checkpoint as the best model.

Training does not rely on an explicit convergence criterion; instead, the final model is selected as the checkpoint that achieves the best validation performance over the entire training process.

For evaluation, we prepare 10 independent test episodes, each with 50 jobs, to assess the performance of different scheduling methods under diverse workload realizations.

Each job is initialized with a minimal set of semantic attributes, including the model type, batch size, number of training epochs, submission time, and deadline-related configuration. These attributes are sufficient to characterize the computational complexity and resource requirements of the job. Other hyperparameters that have limited impact on overall training complexity, such as learning rate, are omitted for simplicity.

To generate such jobs in a structured manner, we define a set of job templates, each corresponding to a representative model family (e.g., ResNet, VGG, Inception, and GPT).

### 4.3.1 Job Templates

Jobs are generated based on a set of predefined templates corresponding to different model families, including ResNet, VGG, Inception, and GPT. Each template specifies a range of feasible batch sizes and candidate GPU counts, reflecting the typical training configurations of different workloads. For example, convolutional models generally support larger batch sizes and moderate GPU parallelism, while large language models tend to require smaller batch sizes but a higher number of GPUs.

### 4.3.2 Batch Size and Epochs

Batch sizes are sampled from template-specific candidate sets with a bias toward smaller values, using a skewed sampling distribution. Similarly, the number of training epochs is sampled such that most jobs are relatively short, while a small fraction of jobs are significantly longer. This design captures the heavy-tailed nature of training workloads observed in practice.

### 4.3.3 Job Arrival Process

Job submission times are generated within a fixed time horizon and follow a mixture distribution combining a uniform background arrival process and multiple Gaussian

peaks. This allows us to model both steady job arrivals and bursty workload patterns, which are commonly observed in shared cluster environments.

Specifically, all jobs arrive within a horizon of $10 \times 3600$ simulated time steps, corresponding to a 10-hour period in real time (where one simulation time step represents one second). Within this horizon, submission times are sampled using a mixture of a uniform background component and several Gaussian peaks, enabling the generation of both baseline traffic and bursty arrival patterns.

### 4.3.4 Training Dataset Size for Job Generation

In this subsection, we describe the dataset sizes associated with individual training jobs in the simulated workload, rather than the datasets used for training or evaluating the RL scheduler.

For convolutional neural network (CNN) jobs, the dataset size (i.e., the number of training samples) is randomly sampled within a predefined range. Specifically, we define a minimum and maximum dataset size, and sample a normalized variable from a Beta distribution $\text{Beta}(2, 5)$ to bias the sampling toward smaller datasets while still allowing occasional large-scale jobs. The final dataset size is then obtained by linearly scaling this sample to the predefined range (with the maximum set to 11,060,223 samples and the minimum set to 40% of this value), as summarized in Table 2. This design reflects the observation that most training jobs are relatively small, while large-scale jobs appear less frequently.

For other job types, dataset size is modeled in a similar range-based manner but adapted to their training characteristics, with configuration summarized in Table 2. For transformer-based language models (e.g., GPT-style models), the effective dataset size is represented by the total number of training tokens, with an upper bound of 50,000. For speech recognition models (e.g., DeepSpeech2), dataset size corresponds to the total number of audio samples or utterances, with an upper bound of 104,014. In both cases, sampling follows the same skewed distribution strategy to preserve workload diversity.

These dataset-related attributes are not directly used by the scheduler. Instead, they serve as intermediate variables for deriving system-level characteristics, including computational cost (in FLOPs), memory requirements, and job deadlines, which are ultimately exposed to the scheduling agent.

### 4.3.5 Job Profiling

Upon arrival at the job queue, each job is automatically processed by a profiling module that derives its resource and computational characteristics based on its semantic attributes. Specifically, given the job type, batch size, dataset size, and training configuration, the profiler estimates the total computational cost of the job in FLOPs, as well

as its memory requirement.

These estimates are subsequently used to determine job feasibility, such as whether a job can fit within GPU memory constraints, and to approximate execution time under different resource allocations. In other words, once a job is received by the cluster-side job queue, the profiling procedure is automatically invoked to transform high-level job semantics into quantitative system-level descriptors that can be directly used by the scheduler.

### 4.3.6 Deadline Generation

Based on the profiling results, we further assign a deadline to each job. Specifically, the deadline is derived from the estimated computational cost of the job, measured in terms of total FLOPs, together with the number of GPUs requested by the job.

We first estimate a baseline execution time by dividing the total computational demand by the effective cluster throughput, which is determined by the average peak FLOPs per GPU, the number of GPUs assigned to the job, and a scaling efficiency factor. The average peak FLOPs per GPU is treated as a tunable hyperparameter reflecting the overall computational capability of the cluster.

To model diverse user expectations, we introduce a slack factor sampled from a mixture distribution. This results in three categories of deadlines: tight, medium, and loose. Tight deadlines correspond to latency-sensitive jobs that require prompt scheduling, while loose deadlines allow greater flexibility in resource allocation. The final deadline is obtained by scaling the baseline execution time with the sampled slack factor.

This design ensures that deadlines remain correlated with job complexity while preserving diversity in urgency levels, thereby enabling a more realistic evaluation of scheduling policies.

## 4.4 Baselines

We compare the proposed SARL scheduler against a diverse set of baselines, including five heuristic scheduling algorithms, one RL-based method, and one optimization-based approach.

For a fair comparison, all learning-based baselines are implemented using the same job and cluster representations, and are adapted with minimal modifications to be compatible with our environment.

### 4.4.1 Heuristic Scheduling Algorithms

We consider five widely used heuristic scheduling algorithms as baselines:

**FIFO.** FIFO schedules jobs strictly in the order of their arrival. Resources are allocated to jobs based on submission time, without considering job length or deadline. This strategy provides a reasonable balance in waiting time across jobs and is commonly adopted as a default scheduling policy in production GPU clusters.

**SJF.** SJF prioritizes jobs with the smallest estimated size. In our setting, job length is defined based on the total computational workload. By favoring shorter jobs, SJF improves system throughput and is often effective in reducing average JCT.

**EDF.** EDF prioritizes jobs with the most urgent deadlines. We adopt a preemptive version of EDF, where newly arrived jobs with tighter deadlines can preempt currently running jobs. This enables the scheduler to better satisfy deadline constraints under dynamic workloads.

**Backfilling.** Backfilling [80] improves resource utilization by allowing smaller jobs to be scheduled ahead of waiting large jobs, as long as the execution of the large jobs is not delayed. Specifically, when a large job is waiting for sufficient resources, the scheduler reserves the required resources while temporarily filling idle slots with smaller jobs that can complete before the reserved resources are needed. This approach increases cluster utilization without violating the scheduling guarantees of large jobs.

**Tiresias (Discretized 2D-LAS).** We implement the discretized 2D-LAS scheduler proposed in Tiresias [16]. This method is an information-agnostic scheduling policy that does not rely on prior knowledge of job duration. Instead, it assigns priority based on attained service, i.e., the amount of GPU time a job has already received, following the Least Attained Service (LAS) principle.

The 2D-LAS formulation extends LAS to a two-dimensional setting by jointly considering temporal progress (attained service) and resource demand (number of GPUs). Jobs with less attained service are prioritized, which helps reduce queueing delay and mitigates head-of-line blocking caused by long-running jobs.

Compared to approaches such as the 2D-Gittins index scheduler, which rely on partial knowledge of job duration distributions, 2D-LAS is fully information-agnostic and therefore more suitable for our setting. In our environment, job execution time depends on heterogeneous factors such as GPU type (e.g., A100 vs. V100) and placement topology, making runtime estimation unreliable. As a result, methods that depend on prior distributional assumptions or analytical predictions are not directly applicable.

### 4.4.2 RL Algorithm

$DL^2$ [66] is a learning-based scheduler designed for deep learning clusters. It leverages a neural network to dynamically adjust resource allocation for running jobs, aiming to minimize overall training completion time.

$DL^2$ adopts a hybrid training strategy that combines supervised learning and reinforcement learning. The model is first initialized using offline supervised learning based on historical scheduling traces, and is then refined online through reinforcement learning during job execution. This design enables a smooth transition from existing heuristic schedulers while improving scheduling performance.

We also considered implementing TopDRL [26], a representative single-agent RL scheduler that explicitly considered topology of the cluster. However, due to a mismatch between its underlying assumptions and our simulation environment, a faithful implementation was not feasible. A detailed discussion of these discrepancies is provided in Appendix A.

**$DL^2$ Implementation Details.** We implement a $DL^2$-style scheduling environment following an incremental resource allocation paradigm. At each scheduling round, the agent iteratively assigns GPUs to jobs by selecting a job and allocating one additional GPU, until a stop action is issued. This formulation converts the combinatorial resource allocation problem into a sequence of discrete decisions.

To ensure consistency with our simulation framework, we adopt a two-phase execution mechanism. During the allocation phase, decisions are applied to a temporary copy of the cluster state, allowing the agent to construct a full allocation plan without affecting the actual system. Once the stop action is triggered, all allocation decisions are committed to the real cluster automatically, and the environment advances for a fixed simulation horizon.

The observation space consists of per-job features derived from the cluster state, including resource demand, allocation status, and system-level context. The action space is defined as selecting one of the candidate jobs to allocate an additional GPU, along with a special stop action. An action masking mechanism is employed to ensure feasibility, preventing invalid allocations such as exceeding job demand or violating resource constraints.

We adopt a best-fit incremental allocation heuristic to determine the placement of GPUs for each allocation decision. Specifically, the allocator prioritizes assigning GPUs from the same machine as those already allocated to the target job. If this is not feasible, it expands the allocation scope to other machines within the same rack, and finally to machines across different racks.

The reward is computed based on normalized training progress accumulated during

simulation, encouraging the agent to improve overall system throughput while maintaining fairness across jobs.

To extend $DL^2$ to distributed training scenarios, we adapt its original single-resource allocation formulation to support multi-GPU jobs. Since $DL^2$ allocates resources incrementally, we allow jobs to start execution even when the number of allocated GPUs is smaller than their requested demand. In such cases, the job progresses at a reduced speed determined by the allocated GPU count and placement topology, reflecting the impact of parallelism and communication overhead.

In addition, we follow the original $DL^2$ design and assume that dynamically increasing the number of GPUs for a running job does not incur additional reconfiguration cost. Specifically, expanding a job's allocation does not trigger a full preempt-and-reschedule procedure, enabling smooth scaling of resources during execution.

### 4.4.3 Optimization Based

We include an optimization-based scheduler(namely, FFT) proposed in [67] as a baseline. This method explicitly targets deep learning workloads in heterogeneous GPU clusters and formulates scheduling as a global optimization problem.

Unlike heuristic policies, this approach models job execution performance under different GPU types and configurations, and jointly optimizes resource allocation to improve both system throughput and fairness. In particular, it explicitly accounts for GPU heterogeneity by considering the performance differences across devices when assigning resources, allowing jobs to be mapped to the most suitable GPUs.

While meta-heuristic methods and FFT both aim to explore the space of possible job-to-resource assignments, they differ fundamentally in how this exploration is performed. Meta-heuristic methods rely on iterative and stochastic search over a large candidate space, whereas FFT explicitly constructs a set of candidate assignments and selects the optimal combination in a single pass via integer linear programming (ILP). This structured optimization process replaces the need for iterative meta-heuristic search, enabling more efficient decision making in online scheduling scenarios.

**FFT Implementation Details.** We implement an optimization-based scheduler following the design of [67]. At each scheduling round, the scheduler evaluates candidate assignments for each job across different GPU models. For each candidate, the expected training throughput is estimated under the corresponding resource configuration, enabling heterogeneity-aware decision making. This allows the scheduler to account for performance differences across GPU types when assigning resources.

The scheduling objective considers job progress, resource efficiency, and fairness, and aims to determine a set of job-to-resource assignments that minimizes the overall

cost while respecting resource capacity constraints. In addition, switching overhead is incorporated to capture the cost of reallocating running jobs.

Since the original paper does not explicitly specify the interval between scheduling rounds, we adopt a fixed scheduling interval (1800 simulated time steps) in our implementation, where the scheduler is invoked periodically every several simulation time steps. This design aligns with our event-driven environment while ensuring consistent comparison across different methods.

## 4.5 Evaluation metrics

We evaluate scheduling performance using three widely adopted metrics: mean JCT, deadline miss rate, and maximum waiting time. While mean JCT and deadline miss rate are the primary optimization targets, maximum waiting time is included to account for fairness across jobs.

**Mean JCT.** For each job $i$, the job completion time is defined as:

$$\mathrm{JCT}_i = t_i^{\text{completion}} - t_i^{\text{submission}}.$$

The mean JCT over all jobs is computed as:

$$\mathrm{JCT}_{\text{mean}} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{JCT}_i,$$

where $N$ is the total number of jobs. Lower values indicate better overall system efficiency.

**Deadline Miss Rate.** Each job is associated with a deadline. A job is considered to meet its deadline if it completes before its deadline. The deadline miss rate is defined as:

$$\mathrm{DMR} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\left(t_i^{\text{completion}} > t_i^{\text{deadline}}\right),$$

where $\mathbb{I}(\cdot)$ is the indicator function. Lower values indicate better deadline satisfaction.

**Waiting Time.** The waiting time of a job is defined as the cumulative duration it spends in the queue without being allocated any GPU resources. In particular, if a job is preempted and returned to the queue, its waiting time continues to accumulate during subsequent queueing periods.

Formally, the waiting time of job $i$ is defined as the total time spent in the queued

state across its entire lifetime:

$$\mathrm{WT}_i = \sum_k \left( t_{i,k}^{\mathrm{start}} - t_{i,k}^{\mathrm{queued}} \right),$$

where each term corresponds to a queueing interval between a job entering the queue and being scheduled again.

We report the maximum waiting time across all jobs.

The maximum waiting time is defined as:

$$\mathrm{WT}_{\max} = \max_{i=1}^{N} \mathrm{WT}_i,$$

which captures the worst-case waiting time and indicates the degree of starvation.

Lower values indicate improved fairness.

## 4.6 Training Performance

We train the vanilla SARL agent from scratch in an online manner. During training, the model is evaluated every 10,000 time steps on workloads generated from five different random seeds. Performance is measured separately on each seed and then aggregated to assess overall generalization performance. The total training budget is set to 400,000 time steps, corresponding to approximately 1,200 episodes.

To ensure stable model selection, we track the best-performing model based on the training reward. Specifically, the final model used for evaluation is the checkpoint that achieves the highest reward during training, rather than the one obtained at the end of training.

After training, the selected SARL model is evaluated on a separate test set and compared against the seven baselines described above. For baseline methods that do not explicitly perform GPU allocation, we adopt a variant of the Best-Fit allocation strategy (Algorithm 3), where GPU model distinctions are ignored to ensure a fair comparison.

At the early stage of training, the vanilla SARL agent performs significantly worse than strong heuristic baselines due to random initialization and exploration. However, as training progresses, the performance steadily improves and eventually surpasses all baselines.

A detailed comparison with baseline methods is presented in the following subsection.

## 4.7 Comparison with Baselines

Due to the interaction between the allocation strategy in DL2 and preemption overhead, its performance is sensitive to the assumed preemption cost. It is worth noting that

the original DL2 framework does not explicitly model fine-grained preemption overhead, whereas our environment incorporates workload-dependent preemption costs.

To account for this difference and ensure a fair evaluation, we assess all methods under two preemption settings: (i) a low preemption cost, set to 0.001 times the normal cost, and (ii) the default (normal) preemption cost. This setup enables us to examine the robustness of different scheduling policies under varying levels of preemption overhead.

Although SARL does not achieve the best performance in terms of maximum waiting time, this metrics is not the primary optimization targets in our setting. Instead, our objective focuses on reducing mean JCT and deadline miss rate, where SARL consistently outperforms all baselines. The differences in waiting time remain minor and do not affect the overall conclusions.

The quantitative results are summarized in Table 3, where each reported value is averaged over 10 independent test seeds. Under all settings, SARL uses the Transformer-based feature extractor and is initialized with the supervised pretraining strategy described earlier, which together improve representation learning and training stability under more challenging conditions.

| Preemption Cost | Metric | Heuristic Models | | | | | RL | Opt. | Ours |
|---|---|---|---|---|---|---|---|---|---|
| | | SJF | FIFO | Backfill | Tiresias | EDF | $DL^2$ | FFT | SARL |
| Low | Mean JCT (TU) | 8319.6 | 8192.2 | 8561.0 | 8808.4 | 8147.0 | 9054.5 | <u>7025.4</u> | **6268.7** |
| | Max Wait (TU) | 4646.8 | 3789.8 | **3227.2** | 3869.6 | 7123.0 | <u>3560.2</u> | 8287.0 | 4245.8 |
| | Miss Rate (%) | 46.8 | 38.0 | 37.6 | 33.6 | <u>31.6</u> | 46.0 | 32.0 | **23.5** |
| Normal | Mean JCT (TU) | 8319.6 | 8192.2 | 8561.0 | 8808.4 | <u>8005.6</u> | 10052.3 | 9324.0 | **6852.8** |
| | Max Wait (TU) | 4646.8 | <u>3789.8</u> | **3227.2** | 3869.6 | 7250.1 | 4920.8 | 14980.3 | 3883.9 |
| | Miss Rate (%) | 46.8 | 38.0 | 37.6 | 33.6 | <u>32.4</u> | 54.0 | 56.0 | **22.8** |

Table 3: Comparison under different preemption cost settings. Results are averaged over 10 test seeds. **Bold** indicates the best result, and <u>underline</u> indicates the second best. Lower values indicate better performance for all metrics.

### 4.7.1 Result Analysis

From Table 3, several important observations can be made.

We first note that heuristic methods such as SJF, FIFO, Backfill, and Tiresias exhibit identical performance under both low and normal preemption cost settings. This is because these methods are non-preemptive, and therefore their scheduling decisions are not affected by changes in preemption overhead.

First, SARL consistently outperforms all baselines across both preemption settings, achieving the lowest mean JCT and deadline miss rate. This demonstrates the effectiveness of learning-based scheduling in capturing complex interactions between job characteristics, resource allocation, and preemption decisions. In particular, SARL is able to implicitly infer the impact of preemption on job progress under different system conditions, and accordingly adjust its scheduling strategy, enabling more informed decisions on when preemption is beneficial or should be avoided.

Second, the performance gap becomes more pronounced under the normal preemption cost setting. In particular, $DL^2$ and FFT significantly underperform in both JCT and miss rate when preemption becomes expensive. This is because these methods either rely on frequent reallocation or assume relatively low switching overhead, making them less robust under realistic preemption costs.

To further validate the improvement in deadline-related performance, we compare SARL with EDF, which is the strongest baseline in terms of deadline satisfaction. Under the normal preemption cost setting, we perform a Welch's t-test on the deadline miss rate across multiple test seeds. The test yields a p-value of 0.046, indicating that the observed performance difference is statistically significant at the 5% significance level. Therefore, we reject the null hypothesis that the two methods have equal performance, and conclude that SARL achieves a statistically significant improvement over EDF.

In contrast, SARL adapts its behavior by implicitly learning to avoid unnecessary preemption under high-cost conditions, leading to more stable execution and fewer deadline violations.

Among heuristic baselines, EDF achieves the best overall performance, particularly in terms of deadline miss rate, followed by FIFO. This indicates that simple priority-based strategies remain effective when the required scheduling signals (e.g., deadlines or arrival order) are readily available in the environment. Backfill, while not achieving the best overall performance, exhibits the lowest maximum waiting time. This suggests that it effectively utilizes idle resources by scheduling smaller jobs when large jobs block the queue head, thereby mitigating starvation. Since we report maximum waiting time rather than average waiting time, this result highlights Backfill's ability to control worst-case delays.

Interestingly, we observe that under the normal preemption cost setting, EDF shows a slight (though not statistically significant) improvement in mean JCT compared to the low-cost setting. We attribute this phenomenon to the dynamic nature of EDF's decision process. Since EDF decisions are triggered by job arrivals and completions,

higher preemption cost leads to larger progress rollback when jobs are interrupted, which in turn delays job completion events. This effectively alters the timing of EDF's decision points, resulting in a different sequence of scheduling decisions that can occasionally lead to marginally better JCT.

However, from a system-level perspective, the increase in preemption cost still negatively impacts overall responsiveness. In particular, the maximum waiting time of EDF increases (by 1.7%), indicating that some jobs experience longer delays. This suggests that while local scheduling decisions may occasionally improve, the global effect of higher preemption cost is still detrimental to latency-related metrics.

In contrast, SARL demonstrates improved robustness under higher preemption cost, maintaining a comparable deadline miss rate to the low-cost setting and reduced maximum waiting time (-9.3%).

$DL^2$ demonstrates relatively stable performance across both settings, which may be attributed to its use of heuristic-based pretraining. However, its performance remains limited, as it does not explicitly consider cluster topology during GPU allocation. As a result, it cannot effectively capture communication overhead induced by distributed execution, leading to suboptimal scheduling decisions.

In particular, when the preemption cost increases, $DL^2$ exhibits a noticeable degradation in both average and tail performance. Its mean JCT increases by 11.0%, while its maximum waiting time rises significantly (by 38.2%). Consequently, the advantage it previously held under the low preemption cost setting—where it achieved the second-lowest waiting time after Backfill—no longer exists.

Finally, the optimization-based scheduler (FFT) exhibits mixed performance in our setting. We observe that its performance improves noticeably under the low-preemption setting, while achieving lower mean JCT than other baselines.

This method relies on several key quantities, such as switching cost and job throughput, which are assumed to be accurately known. However, in our simulation environment, these quantities are not directly observable and must be estimated. For example, switching cost depends on the amount of lost training progress, which is influenced by model characteristics, checkpointing behavior, and previously accumulated progress. Similarly, job throughput depends on GPU placement and cluster topology, including intra-machine and cross-rack communication overhead, as well as resource contention among concurrently running jobs.

As a result, the optimization procedure relies on approximate estimations, and errors in these inputs can propagate to the final scheduling decisions, leading to suboptimal performance. Unlike learning-based approaches, FFT does not adapt its policy based on past experience and therefore cannot correct such modeling inaccuracies over time.

In addition, we observe that FFT behaves differently under varying preemption costs. Although the original design acknowledges the need to avoid excessive job migration,

preemption is not explicitly modeled as a first-class decision variable. Instead, it is implicitly controlled through per-round allocation and the choice of scheduling interval.

This design introduces a trade-off between responsiveness and stability. With a small scheduling interval (i.e., frequent re-optimization), the scheduler can react quickly but may trigger frequent job migrations. Under low preemption cost, this does not significantly impact performance and can even improve resource utilization. However, under realistic (normal) preemption cost, frequent migrations become expensive. Increasing the scheduling interval can reduce switching overhead, but overly large intervals lead to delayed decisions, increased waiting time, and potential starvation.

As a result, FFT faces an inherent trade-off: balancing preemption overhead against scheduling responsiveness. In our experiments, this leads to increased mean JCT and maximum waiting time under normal preemption cost, despite improved performance in the low-preemption setting.

## 4.8 Inference Time

We evaluate the total inference time of different scheduling methods over 10 test episodes, and averaged. For learning-based methods (SARL and $DL^2$), inference corresponds to policy evaluation during each scheduling decision.

Figure 5 presents the comparison. SARL achieves the lowest total inference time, with an average of 1.05 s per episode, and 2.85 s when using the Transformer-based feature extractor (with supervised pretraining). Note that pretraining only affects parameter initialization and does not introduce additional inference overhead. This is compared to 3.78 s for $DL^2$ and 5.59 s for FFT.

The lower inference cost of SARL is primarily due to its use of an MLP-based PPO policy, which enables efficient inference through a single forward pass after training. Given the current system state, SARL produces a complete scheduling decision in one step, resulting in low and consistent latency.

When incorporating the Transformer-based feature extractor, the inference time increases significantly. This is mainly due to the higher computational complexity of self-attention mechanisms, which introduce additional matrix multiplications and require processing interactions among all jobs and machines. Moreover, the sequence-based encoding and attention pooling operations add overhead compared to simple MLP feature extraction, leading to increased latency during each decision step.

In contrast, $DL^2$ incurs higher inference overhead due to its sequential decision-making process. Specifically, selecting a job only assigns a single GPU, and the scheduler must repeatedly select actions until a STOP action is issued to complete the allocation. This multi-step decision process increases the overall inference time per scheduling decision.

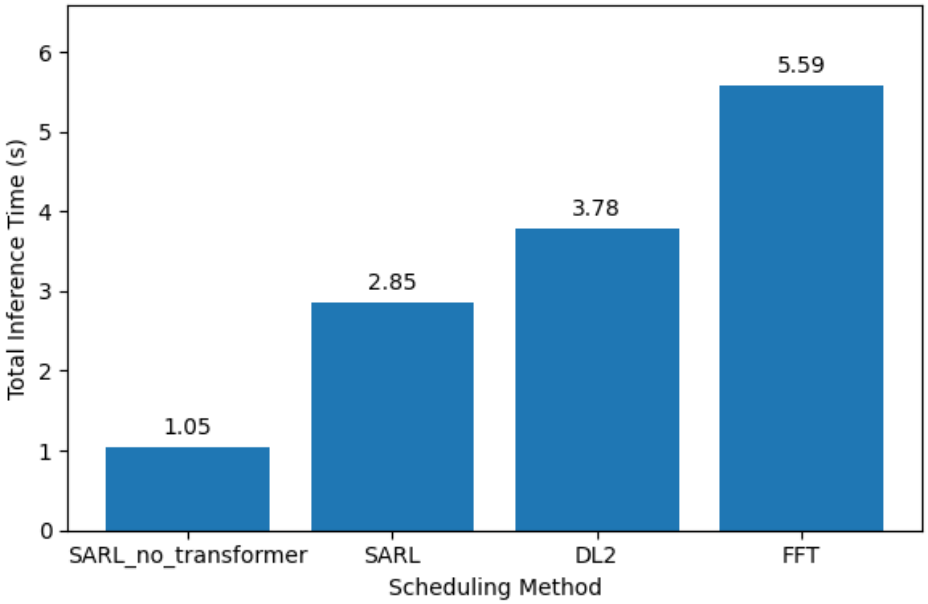



Figure 5: Total inference time comparison across different scheduling methods. SARL achieves the lowest inference overhead, significantly outperforming $DL^2$ and FFT.

We also observe that FFT incurs the highest inference time among all methods. Although FFT operates in a round-based manner, each scheduling decision requires solving an optimization problem. Specifically, FFT formulates the allocation as an ILP problem to efficiently evaluate candidate assignments, which avoids the iterative population-based search required by meta-heuristic methods (see Appendix B), thereby reducing the number of candidate evaluations, while each evaluation is carried out more efficiently through structured optimization rather than repeated simulation-based scoring. However, solving an ILP remains computationally expensive, especially as the number of jobs and resources increases. Moreover, FFT still needs to evaluate and compare multiple candidate allocations within each round, leading to substantial computational overhead per decision.

Noticably that the total inference time of FFT is sensitive to the scheduling interval. Specifically, the choice of *round_time*, which determines how frequently scheduling decisions are made, has a significant impact on the overall inference overhead.

A smaller *round_time* increases scheduling responsiveness, which helps reduce job waiting time and can indirectly improve average JCT. However, more frequent scheduling decisions significantly increase the inference overhead and may lead to excessive preemptions, introducing additional overhead due to checkpointing.

In contrast, a larger *round_time* reduces inference cost and avoids frequent preemptions, but results in coarser scheduling decisions. This may increase job waiting time and consequently lead to higher JCT.

Empirically, we find that reducing the scheduling interval (e.g., decreasing *round_time*

from 1800 to 300) can effectively improve mean JCT under low preemption cost, with a corresponding reduction in deadline miss rate (e.g., mean JCT improves from 7025.4 to 6390, and deadline miss rate decreases from 35.0% to 28.0%). However, this comes at the expense of a several-fold increase in inference overhead. Under normal preemption cost, the benefit diminishes, and the increased preemption overhead outweighs the gains, leading to overall worse performance.

Overall, all methods operate within a second-level inference time, which is acceptable for GPU cluster scheduling, where decisions are made at relatively coarse time granularity (e.g., seconds) and do not require millisecond-level responsiveness.

## 4.9 Enhancement Techniques

In this section, we evaluate the impact of two design components within the proposed SARL framework, namely a Transformer-based feature extractor and supervised pretraining.

Rather than introducing separate variants of the scheduler, these techniques are treated as optional components of the same model, and are analyzed to understand their contributions to performance and training efficiency.

### 4.9.1 Transformer-based Feature Extractor

We replace the original feature extractor with a Transformer architecture while keeping the same policy ($\pi$) and value ($V$) network configurations.

**Faster convergence.** The primary advantage of the Transformer-based extractor lies in significantly improved training efficiency. On the default cluster, the Transformer-based model begins to converge after approximately 75 training episodes, whereas the vanilla SARL requires substantially more episodes to reach a similar performance level.

**Improved training stability.** In addition to faster convergence, the Transformer-based model exhibits lower performance variance during training. In contrast, the vanilla SARL shows noticeably larger fluctuations even after convergence, indicating less stable policy learning.

**Comparable final performance.** After sufficient training, the Transformer-based model achieves performance comparable to vanilla SARL in terms of both mean JCT and deadline miss rate, with no statistically significant differences.

**Better performance under reduced capacity.** More importantly, the Transformer extractor enables effective learning under significantly reduced model capacity. When reducing the network size from $(1024, 1024, 512)$ to $(256, 128, 64)$, the agent is still able to learn a stable scheduling policy, with the mean JCT remaining within 8% of the full SARL model and a deadline miss rate gap of only 4.5%. In contrast, the vanilla SARL fails to converge under the same reduced setting.

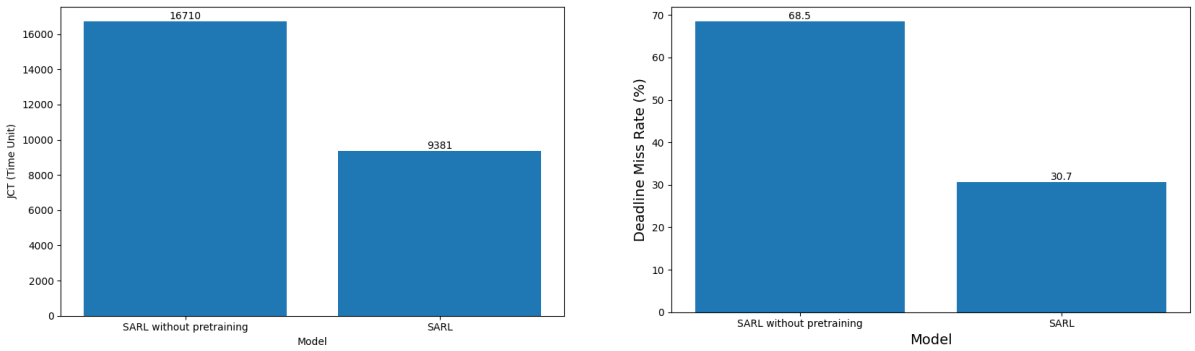
This further demonstrates that the Transformer-based feature extractor can more effectively capture and compress relevant system information, leading to both faster and more stable learning.

### 4.9.2 Supervised Pretraining

We evaluate supervised pretraining using a heuristic policy as a teacher to initialize the model parameters. Under low preemption cost, the pretrained SARL achieves, within only a few episodes, a performance level comparable to the mid-stage of SARL without pretraining training (e.g., after 600+ episodes). However, its final performance is slightly lower(but not significant) and exhibits higher variance during training. We attribute this to reduced exploration diversity caused by the initialization, which limits the agent's ability to fully explore the policy space when preemption is inexpensive.

In contrast, under high preemption cost, supervised pretraining provides substantial benefits. The pretrained SARL converges within 50 episodes to a performance level comparable to EDF, and further improves to match the final performance reported earlier. Without pretraining, SARL either fails to converge or requires significantly more training episodes.

We attribute this improvement to the teacher policy providing a strong and concrete starting point, which effectively guides exploration in a large and complex action space.




(a) Mean JCT

(b) Deadline miss rate

Figure 6: Impact of supervised pretraining under different preemption cost settings(after 500 episodes).

Overall, these results suggest that supervised pretraining is particularly beneficial in scenarios with complex action spaces, such as smaller clusters or high contention settings, where frequent preemption decisions and large candidate sets increase the difficulty of exploration.

### 4.9.3 Combination of Transformer and Supervised Pretraining

We further evaluate the combined effect of the Transformer-based feature extractor and supervised pretraining.

This hybrid approach inherits the advantages of both techniques. Similar to supervised pretraining, the model achieves strong performance within only a few episodes, demonstrating a significantly faster warm-up compared to vanilla SARL. At the same time, the Transformer extractor contributes to more stable training dynamics, as reflected by smoother loss curves.

As training progresses, the deadline miss rate quickly converges to a level comparable to EDF, while maintaining competitive mean JCT performance.

| **Scheduler** | **Mean JCT (TU)** | **Miss Rate (%)** |
|---|---|---|
| SARL (Pretrained + Transformer) | 7261 | 26 |
| EDF | 8005 | 32.4 |

Table 4: Performance comparison between SARL(after 205 episodes of training) with both enhancements and EDF.

Overall, based on the above observations, we find that supervised pretraining is generally beneficial, as it provides the agent with a strong initialization and significantly reduces the exploration time required to learn effective scheduling policies. Meanwhile, the incorporation of the Transformer-based feature extractor improves training stability and helps the agent achieve competitive performance comparable to vanilla SARL.

We attribute this to the fact that a sufficiently trained MLP-based PPO model is capable of learning meaningful representations from the observation space and capturing the underlying scheduling patterns. In this context, the proposed enhancement techniques can be viewed as complementary mechanisms that improve robustness, particularly under more challenging conditions such as smaller clusters with higher contention.

In contrast, under medium or large cluster settings, where more GPUs are available, the scheduling problem becomes less sensitive to individual decisions. Specifically, the larger resource pool provides the agent with more feasible allocation options, reducing the likelihood that a locally suboptimal decision leads to severe resource fragmentation or long-term inefficiency. As a result, SARL can more effectively refine its policy through exploration, since the impact of each individual action on future system states is less drastic, allowing for more stable and gradual performance improvement.

## 4.10 Ablation Study

To better understand the contribution of each component in SARL, we conduct an ablation study by removing or modifying key design elements.

| Variant | Mean JCT (TU) | Miss Rate (%) |
|---|---|---|
| SARL | 6852.8 | 22.8 |
| w/o Job type in observation | 7485.6 (+9.2%) | 28.5 (+25.0%) |
| w/o Preemption | 7734.4 (+12.9%) | 38.0 (+66.7%) |
| w/o model allocation | 7254.0 (+5.9%) | 27.0 (+18.4%) |
| w/o Reward shaping | 7178.0 (+4.7%) | 27.8 (+21.9%) |

Table 5: Ablation study of SARL. Relative changes are computed with respect to the vanilla SARL model.

#### 4.10.1 w/o Job Type in Observation

We remove the job type information from the observation space by assigning the same encoding to all jobs. This eliminates the agent's ability to differentiate between workload characteristics.

#### 4.10.2 w/o Preemption

We disable preemption by modifying the action mask, such that all preemption-related actions are masked out. As a result, the agent can only perform scheduling or no-op decisions.

#### 4.10.3 w/o model Allocation

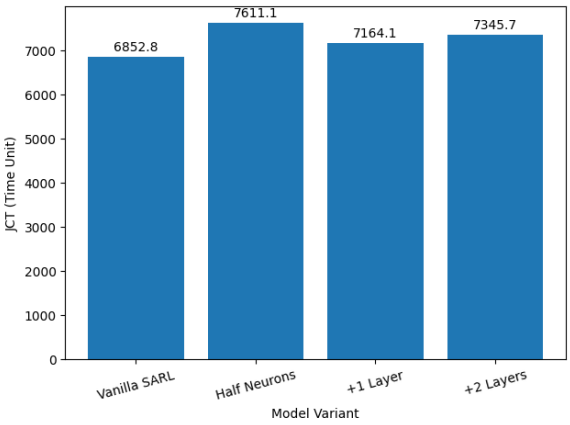
We remove the agent's ability to select GPU models by modifying the allocation strategy. Instead of allowing the agent to specify the target GPU type, we adopt the same best-fit allocation used in heuristic baselines, where the system automatically assigns available GPUs based on current resource availability.

#### 4.10.4 w/o Reward Shaping

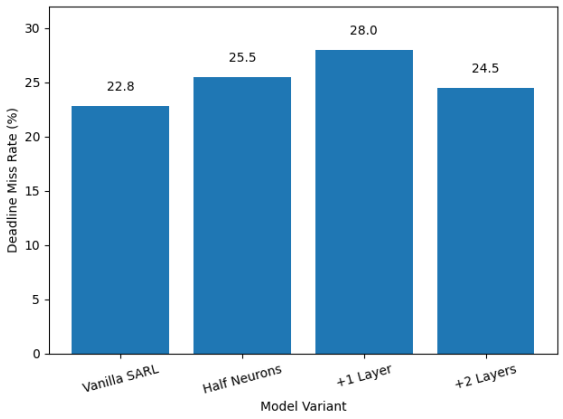
We replace the original reward function with a DeepRM-style formulation [81]. At each decision step, the reward is defined as the negative inverse of the estimated job completion time. The estimated completion time is computed based on the average cluster compute capacity, and thus remains a static approximation independent of system dynamics.

#### 4.10.5 Number of Neurons and Network Depth

We evaluate the impact of network architecture by varying the number of neurons and layers in the policy ($\pi$) and value ($V$) networks. The results are illustrated in Fig. 7a and Fig. 7b.




(a) Mean JCT under different architectures




(b) Deadline miss rate under different architectures

Figure 7: Impact of network architecture (number of neurons and layers) on scheduling performance.

Interestingly, we observe a non-monotonic effect when increasing network depth. Adding a single additional layer leads to noticeable performance degradation, while further increasing the depth results in only marginal additional impact.

We attribute this behavior to the interplay between model capacity and training stability. Introducing one extra layer increases optimization difficulty without providing sufficient representational benefits, leading to higher gradient variance and less stable policy learning. When more layers are added, the increased capacity begins to compensate for this difficulty, allowing the network to better capture complex scheduling patterns and partially mitigating the performance drop.

Overall, increasing network depth does not yield meaningful improvements in either mean JCT or deadline miss rate, while significantly increasing training time. In contrast, reducing the number of neurons leads to clear performance degradation, indicating that sufficient model capacity is essential for capturing the complexity of the scheduling problem.

We also observe that reducing network depth results in unstable training and failure to converge, and thus such configurations are excluded from the results. These findings suggest that, once sufficient model capacity is achieved, increasing network complexity introduces additional optimization difficulty and training instability without bringing meaningful performance gains. Therefore, it is preferable to adopt the simplest architecture—with minimal depth and number of neurons—that satisfies the capacity requirements for both the policy ($\pi$) and value ($V$) networks.

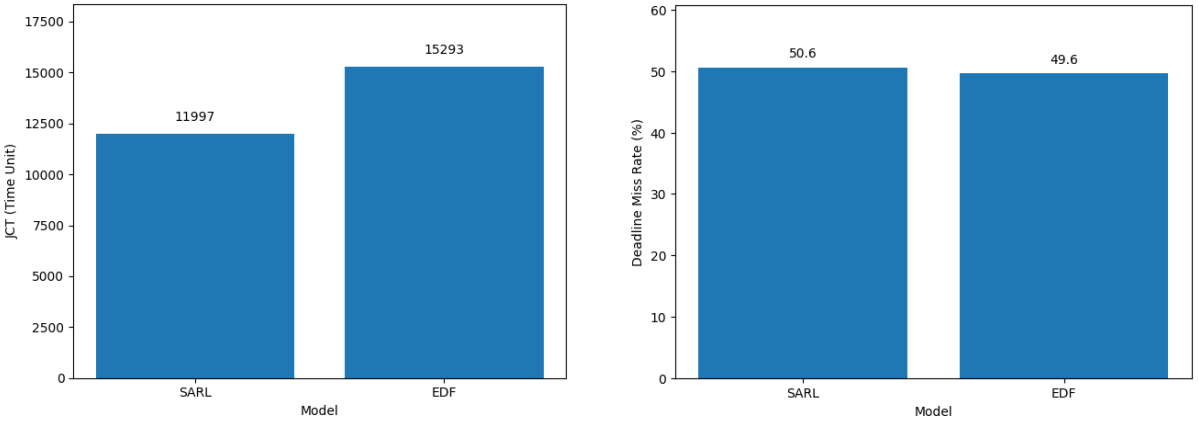
## 4.11 Scalability

To evaluate the scalability of our approach under varying resource conditions, we conduct experiments under both reduced (scale-down) and expanded (scale-up) cluster configurations.

### 4.11.1 Scale-down setting

Compared to the default setting, the cluster contains only half of the GPUs, while the number of jobs per episode remains unchanged, effectively doubling the system load.

The results are shown in Fig. 8. Under this high-load scenario, SARL achieves a mean JCT of 11997 with a deadline miss rate of 50.6%, while the strong baseline EDF obtains a higher mean JCT of 15293 with a comparable miss rate of 49.6%.




(a) Mean JCT

(b) Deadline miss rate

Figure 8: Performance comparison under a small cluster configuration

These results indicate that when server resources are highly limited, meeting job deadlines becomes inherently difficult for all methods, leading to similar deadline miss rates. However, SARL still maintains a clear advantage in terms of mean JCT, demonstrating its ability to better utilize limited resources even under heavy system load.

### 4.11.2 Scale-up Setting

We further evaluate performance under a larger cluster configuration. Compared to the default setting, the total number of GPUs is increased by 1.5× through the addition of a third rack, while the number of jobs per episode remains unchanged.

**Additional rack configuration.** The newly added rack (Rack 3) consists of four machines with heterogeneous GPU configurations:

- Machine 9: 8 × A100 (80GB)
- Machine 10: 8 × H100 (80GB)

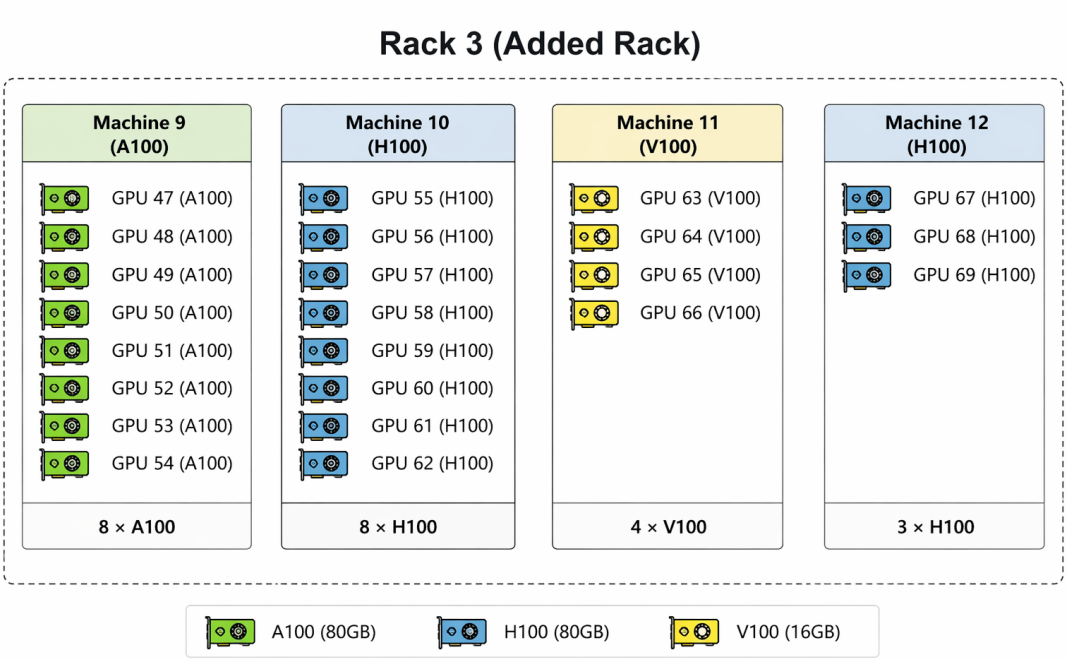
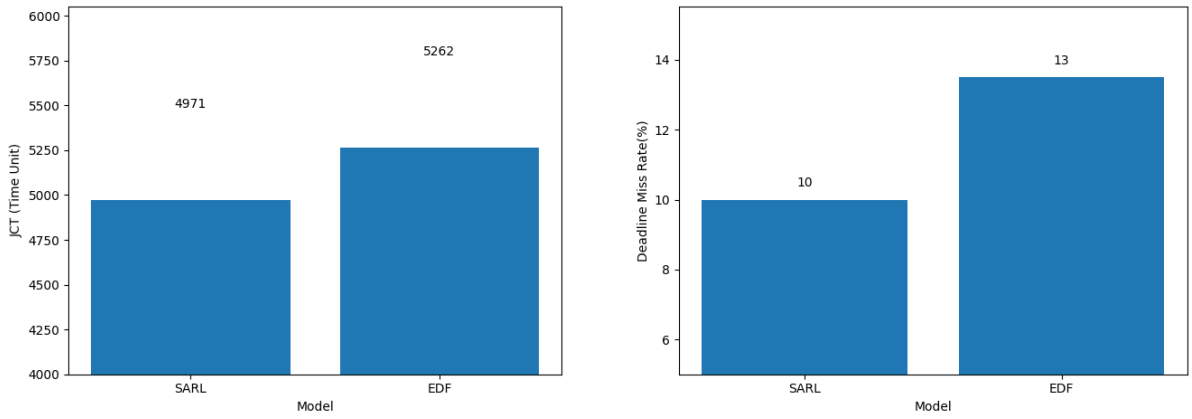



Figure 9: Configuration of the additional rack (Rack 3) used in the scale-up setting.

- Machine 11: 4 × V100 (16GB)
- Machine 12: 3 × H100 (80GB)

This added rack further increases both the scale and heterogeneity of the cluster, introducing additional resource fragmentation and more complex placement decisions. An illustration of the added rack is shown in Figure 9.

The results are shown in Fig. 10, where Fig. 10a and Fig. 10b report the mean JCT and deadline miss rate, respectively. Under this lower-load scenario, SARL consistently outperforms EDF in both metrics, demonstrating improved efficiency and deadline satisfaction when resources are more abundant. Overall, these results show that SARL generalizes well across different resource regimes. As resource availability increases, SARL is able to fully leverage additional capacity and consistently outperform EDF. Under resource-constrained settings, although overall performance degrades for all methods, SARL remains competitive and continues to achieve better or comparable performance compared to EDF, highlighting its robustness across varying system loads.




(a) Mean JCT

(b) Deadline miss rate

Figure 10: Performance comparison under a large cluster configuration

# 5 Conclusion

In this thesis, we present a unified framework for semantic-aware workload modeling and reinforcement learning-based scheduling in multi-GPU clusters. We first develop a semantic-aware deep learning workload simulator that captures both workload semantics and system-level behaviors. Unlike traditional simulators that rely on fixed execution time, our approach models job execution using hardware-agnostic computational workloads, enabling execution time to adapt to different GPU capabilities. Building upon this simulator, we propose SARL, a reinforcement learning-based scheduler for multi-GPU clusters. The proposed approach jointly models job selection, GPU allocation, and preemption decisions, allowing the agent to capture complex interactions in heterogeneous and dynamic environments. Through extensive experiments, we demonstrate that SARL consistently outperforms heuristic, RL and optimization-based baselines in terms of mean job completion time and deadline miss rate.

**Insights** Our study also provides several important insights beyond the design of the scheduler itself.

First, we find that supervised pretraining is an effective way to improve training efficiency. Rather than reducing the action space, pretraining provides a structured initialization that guides the agent toward promising regions of the policy space, thereby reducing ineffective exploration and accelerating convergence.

Second, the Transformer-based feature extractor contributes to more stable training dynamics. By better capturing relationships among jobs and system states, it improves representation quality and reduces variance during training, leading to more consistent performance.

Finally, we observe that many existing scheduling studies rely on trace-based runtime

as a proxy for job length. While this approach simplifies modeling, it limits generalization, particularly in heterogeneous environments where performance depends on hardware characteristics and communication patterns. In contrast, our semantic-aware modeling approach estimates workload based on intrinsic job attributes, enabling more flexible adaptation across different system configurations.

We believe that this bottom-up, semantic-driven perspective can inspire future research to better leverage workload characteristics when designing both simulators and scheduling policies.

**Limitations.** Despite the effectiveness of the proposed approach, several limitations remain.

First, our evaluation is conducted entirely in a simulated environment, and the proposed scheduler has not yet been validated on a real-world GPU cluster. Although the simulator is designed to capture workload semantics and system-level behaviors, discrepancies between simulation and real systems may still exist.

Second, the simulator relies on approximate modeling of key system factors, such as communication overhead and preemption cost. While these approximations improve realism compared to trace-based approaches, they may not fully capture all aspects of real-world system dynamics.

Finally, training the reinforcement learning model introduces additional computational overhead compared to heuristic or optimization-based methods, which may limit its applicability in scenarios with strict deployment constraints. This overhead becomes more pronounced when combining supervised pretraining with a Transformer-based feature extractor. In particular, the training process is dominated by environment simulation and sequential decision making, which can be computationally intensive and CPU-bound, leading to longer training times.

**Future Work** Several directions can be explored to further extend this work.

First, we assume that each GPU executes only one job at a time. A natural extension is to support multi-tenant execution, where multiple jobs share the same GPU by utilizing different CUDA cores or streaming multiprocessors. Such designs could further improve GPU utilization, particularly for computational-bound workloads.

Second, our current framework assumes static resource allocation, where each job requests a fixed number of GPUs. In practice, allowing jobs to dynamically adjust their resource allocation based on system conditions (e.g., elastic training or adaptive scaling) could further improve overall efficiency and system utilization.

Finally, recent advances in memory disaggregation and unified virtual memory (UVM) provide new opportunities for improving resource utilization in GPU clusters. These techniques enable more flexible memory sharing across devices and nodes, potentially

alleviating memory bottlenecks in distributed training. A more detailed discussion of these directions is provided in Appendix D.

# References


[1] X. Liu, D. Gu, Z. Chen, J. Wen, Z. Zhang, Y. Ma, H. Wang, and X. Jin, "Rise of distributed deep learning training in the big model era: From a software engineering perspective," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, Sept. 2023.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. arXiv:1706.03762.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. arXiv:2005.14165.

[4] NVIDIA, "Nvidia a100 tensor core gpu architecture," tech. rep., NVIDIA, 2020. White paper.

[5] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[6] J. Wang, C. Li, T. Wang, J. Guo, H. Yang, Y. Zhuansun, and M. Guo, "Survey of disaggregated memory: Cross-layer technique insights for next-generation datacenters," 2025. arXiv:2503.20275.

[7] F. Yang, S. Peng, N. Sun, F. Wang, Y. Wang, F. Wu, J. Qiu, and A. Pan, "Holmes: Towards distributed training across clusters with heterogeneous nic environment," in *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24, (New York, NY, USA), p. 514–523, Association for Computing Machinery, 2024.

[8] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li,

M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. arXiv:2501.12948.

[9] M. Lin, K. Zhou, and P. Su, "Drgpum: Guiding memory optimization for gpu-accelerated applications," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, (New York, NY, USA), p. 164–178, Association for Computing Machinery, 2023.

[10] B. Li, R. Arora, S. Samsi, T. Patel, W. Arcand, D. Bestor, C. Byun, R. B. Roy, B. Bergeron, J. Holodnak, M. Houle, M. Hubbell, M. Jones, J. Kepner, A. Klein, P. Michaleas, J. McDonald, L. Milechin, J. Mullen, A. Prout, B. Price, A. Reuther, A. Rosa, M. Weiss, C. Yee, D. Edelman, A. Vanterpool, A. Cheng, V. Gadepally, and D. Tiwari, "Ai-enabling workloads on large-scale gpu-accelerated system: Characterization, opportunities, and implications," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1224–1237, 2022.

[11] K. Yang, K. Qinami, L. Fei-Fei, J. Deng, and O. Russakovsky, "Towards fairer datasets: filtering and balancing the distribution of the people subtree in the imagenet hierarchy," in *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, FAT* '20, (New York, NY, USA), p. 547–558, Association for Computing Machinery, 2020.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, 2009.

[13] N. Lakshmipathi, "Imdb dataset of 50k movie reviews." https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews, 2019. Kaggle Dataset.

[14] Alibaba Group, "Alibaba cluster trace program." https://github.com/alibaba/clusterdata, 2025. Accessed: 2025-08-20.

[15] H. Ayadi, A. An, Y. Shao, H. Pourmedheji, J. Deng, J. X. Huang, M. Feiman, and H. Zhou, "Topology-aware gpu job scheduling with deep reinforcement learning and heuristics," *J. Parallel Distrib. Comput.*, vol. 204, Oct. 2025.

[16] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: a gpu cluster manager for distributed deep learning," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (USA), p. 485–500, USENIX Association, 2019.

[17] J. Deng, A. An, H. Ayadi, Y. Shao, H. Pourmodheji, H. Zhou, and M. Feiman, "Gpu Job Scheduler with Deep Reinforcement Learning," *Proceedings of the Canadian Conference on Artificial Intelligence*, may 27 2024. https://caiac.pubpub.org/pub/u6j20p40.

[18] M. Jeon, S. Venkataraman, A. Phanishayee, u. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, (USA), p. 947–960, USENIX Association, 2019.

[19] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, p. 283–294, IEEE, Apr. 2023.

[20] V. Ghadban, "Solving the checkpoint bottleneck: Accelerating ai training with high-performance data intelligence." https://www.ddn.com/blog/accelerating-ai-training-with-high-performance-data-intelligence/, 2025. DDN Blog, accessed 2026-03-13.

[21] Z. Wang, Q. Hu, A. Klimovic, T. Zhang, Y. Wen, P. Sun, and D. Lin, "Semantic-aware scheduling for gpu clusters with large language models," 2025. arXiv:2510.03334.

[22] S. Lee, A. Phanishayee, and D. Mahajan, "Forecasting gpu performance for deep learning training and inference," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems,*

*Volume 1*, ASPLOS '25, (New York, NY, USA), p. 493–508, Association for Computing Machinery, 2025.

[23] Z. Ye, W. Gao, Q. Hu, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen, "Deep learning workload scheduling in gpu datacenters: A survey," *ACM Comput. Surv.*, vol. 56, Jan. 2024.

[24] D. Alistarh, D. Grubic, J. Z. Li, R. Tomioka, and M. Vojnovic, "Qsgd: communication-efficient sgd via gradient quantization and encoding," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 1707–1718, Curran Associates Inc., 2017.

[25] Z. Tang, S. Shi, W. Wang, B. Li, and X. Chu, "Communication-efficient distributed deep learning: A comprehensive survey." arXiv preprint arXiv:2003.06307, 2023.

[26] B. Ryu, A. An, Z. Rashidi, J. Liu, and Y. Hu, "Towards topology aware pre-emptive job scheduling with deep reinforcement learning," in *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, CASCON '20, (USA), p. 83–92, IBM Corp., 2020.

[27] K. Chen and J. Chen, "Gpu job scheduling based on deep reinforcement learning," in *Proceedings of the 2023 7th International Conference on High Performance Compilation, Computing and Communications*, HP3C '23, (New York, NY, USA), p. 34–45, Association for Computing Machinery, 2023.

[28] M. Xing, H. Mao, S. Yin, L. Pan, Z. Zhang, Z. Xiao, and J. Long, "A dual-agent scheduler for distributed deep learning jobs on public cloud via reinforcement learning," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, (New York, NY, USA), p. 2776–2788, Association for Computing Machinery, 2023.

[29] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020. arXiv:2001.08361.

[30] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. arXiv:1412.6980.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[33] D. Narayanan, S. Santhanam, A. Phanishayee, V. Seshadri, N. Devanur, A. Harlap, *et al.*, "Analysis of a large-scale multi-tenant gpu cluster for deep learning workloads," tech. rep., Microsoft Research, 2018.

[34] OpenAI, "Fine-tuning api reference." `https://developers.openai.com/api/reference/resources/fine_tuning`, 2026. Accessed: 2026-03-18.

[35] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, pp. 8024–8035, 2019.

[37] V. V. Baligodugula and F. Amsaad, "Optimizing distributed training approaches for scaling neural networks," 2025. arXiv:2503.23186.

[38] S. Pal, E. Ebrahimi, A. Zulfiqar, Y. Fu, V. Zhang, S. Migacz, D. Nellans, and P. Gupta, " Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training ," *IEEE Micro*, vol. 39, pp. 91–101, Sept. 2019.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[40] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," 2018. arXiv:1706.02677.

[41] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (Red Hook, NY, USA), p. 1223–1231, Curran Associates Inc., 2012.

[42] F. Lai, A. Kadav, and E. Kruus, "Splitbrain: Hybrid data and model parallel deep learning," 2021. arXiv:2112.15317.

[43] H. Yang, Y. Tian, Z. Yang, Z. Wang, C. Zhou, and D. Li, "Research on model parallelism and data parallelism optimization methods in large language model-based recommendation systems," 2025. arXiv:2506.17551.

[44] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, p. 94–110, Jan. 2020.

[45] S. Shi, Z. Tang, X. Chu, C. Liu, W. Wang, and B. Li, "A quantitative survey of communication optimizations in distributed deep learning," 2020. arXiv:2005.13247.

[46] NVIDIA Corporation, "Nvidia nvlink high-speed interconnect application performance brief," tech. rep., NVIDIA, 2014.

[47] T. Gershon, S. Seelam, B. Belgodere, M. Bonilla, L. Hoang, D. Barnett, I.-H. Chung, A. Mohan, M.-H. Chen, L. Luo, R. Walkup, C. Evangelinos, S. Salaria, M. Dombrowa, Y. Park, A. Kayi, L. Schour, A. Alim, A. Sydney, P. Maniotis, L. Schares, B. Metzler, B. Karacali-Akyamac, S. Wen, T. Chiba, S. Choochotkaew, T. Yoshimura, C. Misale, T. Elengikal, K. O. Connor, Z. Liu, R. Molina, L. Schneidenbach, J. Caden, C. Laibinis, C. Fonseca, V. Tarasov, S. Sundararaman, F. Schmuck, S. Guthridge, J. Cohn, M. Eshel, P. Muench, R. Liu, W. Pointer, D. Wyskida, B. Krull, R. Rose, B. Wolfe, W. Cornejo, J. Walter, C. Malone, C. Perucci, F. Franco, N. Hinds, B. Calio, P. Druyan, R. Kilduff, J. Kienle, C. McStay, A. Figueroa, M. Connolly, E. Fost, G. Roma, J. Fonseca, I. Levy, M. Payne, R. Schenkel, A. Malki, L. Schneider, A. Narkhede, S. Moshref, A. Kisin, O. Dodin, B. Rippon, H. Wrieth, J. Ganci, J. Colino, D. Habeger-Rose, R. Pandey, A. Gidh, A. Gaur, D. Patterson, S. Salmani, R. Varma, R. Rumana, S. Sharma, A. Gaur, M. Mishra, R. Panda, A. Prasad, M. Stallone, G. Zhang, Y. Shen, D. Cox, R. Puri, D. Agrawal, D. Thorstensen, J. Belog, B. Tang, S. K. Gupta, A. Biswas, A. Maheshwari, E. Gampel, J. V. Patten, M. Runion, S. Kaki, Y. Bogin, B. Reitz, S. Pritko, S. Najam, S. Nambala, R. Chirra, R. Welp, F. DiMitri, F. Telles, A. Arvelo, K. Chu, E. Seminaro, A. Schram, F. Eickhoff, W. Hanson, E. Mckeever, M. Light, D. Joseph, P. Chaudhary, P. Shivam, P. Chaudhary, W. Jones, R. Guthrie, C. Bostic, R. Islam, S. Duersch, W. Sawdon, J. Lewars, M. Klos, M. Spriggs, B. McMillan, G. Gao, A. Kamra, G. Singh, M. Curry, T. Katarki, J. Talerico, Z. Shi, S. S. Malleni, and E. Gallen, "The infrastructure powering ibm's gen ai model development," 2025. arXiv:2407.05467.

[48] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, p. 3005–3018, Aug. 2020.

[49] Maastricht University Data Science Research Infrastructure, "Checkpointing machine learning training." https://dsri.maastrichtuniversity.nl/docs/checkpointing-ml-training/, 2026. Accessed: 2026-03-13.

[50] R. Fan, T. Ren, M. Xie, S. Gao, J. Shu, and Y. Lu, "Gpreempt: Gpu preemptive scheduling made general and efficient," in *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, (USA), USENIX Association, 2025.

[51] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: fast pipelined context switching for deep learning applications," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, (USA), USENIX Association, 2020.

[52] J. Duan, S. Xu, S. Qian, D. Yang, K. Wang, C. Liao, Y. Yu, Q. Hua, H. Hu, Q. Wang, W. Wu, D. Bao, T. Lu, J. Cao, G. Xue, G. Yang, L. Zhang, and G. Chen, "Gfs: A preemption-aware scheduling framework for gpu clusters with predictive spot instance management," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '26, (New York, NY, USA), p. 117–131, Association for Computing Machinery, 2025.

[53] PyTorch Contributors, "Saving and loading models." https://docs.pytorch.org/tutorials/beginner/saving_loading_models.html, 2018. Accessed: 2026-03-13.

[54] TensorFlow Developers, "Save and load models." https://www.tensorflow.org/tutorials/keras/save_and_load, 2024. Accessed: 2026-03-13.

[55] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), p. 407–420, Association for Computing Machinery, 2015.

[56] Y. Luo, Q. Wang, S. Shi, J. Lai, S. Qi, J. Zhang, and X. Wang, "Scheduling deep learning jobs in multi-tenant gpu clusters via wise resource sharing," 2024. arXiv:2407.13088.

[57] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's time to revisit LRU vs. FIFO," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, USENIX Association, July 2020.

[58] K. C. Cruz, L. J. R. Talon, V. Ostria, M. R. Dicang, and R. B. Baquirin, "An experimental study on the shortest fittest job first (sfjf) scheduling algorithm using dynamic queues," in *Proceedings of 2020 6th International Conference on Computing and Data Engineering*, ICCDE '20, (New York, NY, USA), p. 19–23, Association for Computing Machinery, 2020.

[59] J. Chen, D. Wang, and W. Zhao, "A task scheduling algorithm for hadoop platform," *College of Computer Science, Beijing University of Technology*, 2013. Email: bjut_chen@126.com, {wangdan, zhaowb}@bjut.edu.cn.

[60] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Preemptive thread block scheduling with online structural runtime prediction for concurrent gpgpu kernels," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), p. 483–484, Association for Computing Machinery, 2014.

[61] N. Goel and R. B. Garg, "A comparative study of cpu scheduling algorithms," 2013. arXiv:1307.4165.

[62] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, p. 46–61, Jan. 1973.

[63] A. Bertossi, L. Mancini, and F. Rossini, "Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 9, pp. 934–945, 1999.

[64] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *[1989] Proceedings. Real-Time Systems Symposium*, pp. 166–171, 1989.

[65] K. G. John Gittins and R. Weber, *Main Ideas: Gittins Index*, ch. 2, pp. 19–53. John Wiley  Sons, Ltd, 2011.

[66] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "Dl2: A deep learning-driven scheduler for deep learning clusters," *arXiv preprint arXiv:1909.06040*, 2019.

[67] Z. Mo, H. Xu, and W. C. Lau, "Fast and fair training for deep learning in heterogeneous gpu clusters," in *Proceedings of the 39th ACM International Conference on Supercomputing*, ICS '25, (New York, NY, USA), p. 324–338, Association for Computing Machinery, 2025.

[68] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. arXiv:1512.03385.

[69] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

[70] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. arXiv:1409.1556.

[71] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[72] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 142–150, 2011.

[73] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," 2015. arXiv:1512.02595.

[74] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An asr corpus based on public domain audio books," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5206–5210, 2015.

[75] L. Zhu, "Thop: Pytorch-opcounter." `https://pypi.org/project/thop/`, 2022. A tool for computing the number of FLOPs and parameters of PyTorch models.

[76] NVIDIA Corporation, "Nvidia a100 tensor core gpu architecture." `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020. NVIDIA Ampere architecture whitepaper, accessed 2026-03-21.

[77] NVIDIA Corporation, "Nvidia h100 tensor core gpu." `https://www.nvidia.com/en-us/data-center/h100/`, 2024. Accessed: 2026-03-21.

[78] NVIDIA Corporation, "Nvidia tesla v100 gpu architecture." `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017. NVIDIA Volta architecture whitepaper, accessed 2026-03-21.

[79] S. Kim, "Deepspeech2: Pytorch implementation." `https://github.com/sooftware/deepspeech2`, 2020. Accessed: 2026-04-08.

[80] IBM, "Backfill scheduling in ibm spectrum lsf." https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=jobs-backfill-scheduling, 2025. Accessed: 2026-03-21.

[81] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, (New York, NY, USA), p. 50–56, Association for Computing Machinery, 2016.

[82] J. H. Holland, *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[84] R. Chab, F. Li, and S. Setia, "Algorithmic techniques for gpu scheduling: A comprehensive survey," *Algorithms*, vol. 18, no. 7, 2025.

[85] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[86] Meta, "Llama-2-7b-hf." https://huggingface.co/meta-llama/Llama-2-7b-hf, 2023. Hugging Face model repository, accessed April 11, 2026.

[87] C. Lou and X. Jin, "Efficient far memory-aware scheduling with famas," *IEEE Transactions on Networking*, pp. 1–15, 2025.

[88] S. Choi, T. Kim, J. Jeong, R. Ausavarungnirun, M. Jeon, Y. Kwon, and J. Ahn, "Memory harvesting in multi-gpu systems with hierarchical unified virtual memory," in *USENIX ATC*, pp. 625–638, 2022.

# Appendices

## A Why We Do Not Include TopDRL as a Baseline

Although the work in [26] by Hajer et al. incorporates workload complexity (e.g., FLOPs), it is not directly applicable as a baseline in our setting due to a fundamental difference in problem formulation.

Specifically, their approach treats job execution time as an intrinsic property of each job, which is provided as an input to the scheduler. In contrast, in our environment, execution time is not a fixed attribute, but is instead determined by the scheduling decision itself. The effective execution speed of a job depends on the allocated GPU configuration, including the number and type of GPUs, as well as their placement within the cluster topology.

In particular, hardware heterogeneity leads to significant variation in compute throughput across different GPU types, while communication overhead introduces additional performance variability depending on placement (e.g., intra-node vs. inter-node allocation). As a result, the same job may exhibit substantially different execution times under different allocation strategies.

Therefore, execution time in our setting is an outcome of the scheduling policy rather than a predefined input. This violates a key assumption of [26], making it difficult to reproduce their method faithfully without fundamentally altering its underlying model. Consequently, we do not include it as a baseline.

## B Why We Do Not Include Meta-Heuristic Baselines

Meta-heuristic algorithms, such as Genetic Algorithms (GA) [82] and Simulated Annealing (SA) [83], are widely used for solving combinatorial optimization problems. However, we do not include them as baselines in our evaluation for several reasons.

First, meta-heuristic methods incur significantly higher computational complexity compared to both optimization-based and learning-based approaches. In GPU scheduling, each candidate solution corresponds to a complete assignment of jobs to resources, and evaluating its quality requires estimating system performance under that allocation. While FFT evaluates a limited number of candidate assignments in a single pass per scheduling round, meta-heuristic methods require iterative search over a population of candidate solutions. This results in a computational cost on the order of $\mathcal{O}(I \cdot P \cdot \text{eval})$, where $I$ is the number of iterations and $P$ is the population size. In practice, this leads to orders of magnitude more candidate evaluations than FFT, making meta-heuristic methods significantly more expensive in our setting.

Second, meta-heuristic methods are inherently round-based and must be re-executed whenever the system state changes. In dynamic scheduling scenarios with continuous job arrivals and evolving resource availability, this implies that the optimization process must be repeatedly invoked. Running such iterative algorithms at fine-grained time scales is computationally prohibitive, while applying them at coarse scheduling intervals introduces the same responsiveness limitations as FFT, but with substantially higher overhead.

Third, unlike FFT, which deterministically evaluates and selects from candidate assignments in a single pass, meta-heuristic methods rely on stochastic search and repeatedly evaluate similar candidate solutions across iterations. This redundant exploration further increases computational overhead without guaranteeing improved solution quality under strict time constraints.

In addition, evaluating the quality of each candidate assignment requires simulating its impact on the current system state. In practice, this involves maintaining an isolated copy of the cluster state (e.g., via deep copy or equivalent mechanisms) to avoid interfering with the actual execution state. Each candidate must be evaluated on such an independent state representation, which introduces additional memory overhead and computational cost.

As the cluster size and workload scale increase, the system state becomes more complex, and the cost of maintaining and evaluating multiple such state instances grows accordingly. This further limits the scalability of meta-heuristic methods, making them difficult to generalize to large-scale heterogeneous GPU clusters.

Finally, prior studies have shown that while meta-heuristic methods such as GA/SA may achieve slightly better performance than simple greedy heuristics in heterogeneous

cluster environments, they require minutes of computation time per run [84]. This fundamentally limits their applicability in online scheduling settings.

In practice, such high computational cost forces meta-heuristic methods to operate in a round-based manner with relatively coarse scheduling intervals. However, as discussed earlier, the length of the scheduling round has a significant impact on key performance metrics such as JCT and deadline miss rate.

Moreover, due to the dynamic nature of GPU clusters, the system state may change substantially during the optimization process (e.g., job completions or new arrivals). As a result, the scheduling decisions derived from an outdated system state can become suboptimal by the time they are applied.

For these reasons, meta-heuristic approaches are not well suited for online GPU scheduling scenarios, and are therefore not included as baselines in our evaluation.

# C Extended Workload Configuration Including Llama 2

Although Llama 2 [85, 86] is a widely used large language model in recent years, we do not include it in the main synthetic dataset presented in the thesis.

The primary reason is to ensure consistency in workload profiling and job length estimation. In our framework, workload characterization relies on profiling-based estimation, which requires access to open-source models and feasible execution on a reference GPU. In the main experiments, representative NLP workloads such as GPT-2 are calibrated using NeuSight [22]. However, this tool does not support Llama 2.

Moreover, for large-scale models such as Llama 2, accurate memory profiling becomes impractical under our setup. Our reference GPU has limited memory capacity (40 GB), which is insufficient to run Llama 2 under realistic configurations for offline measurement. As a result, only artificially reduced settings (e.g., very small batch size or sequence length) would be feasible, which are not representative of real-world deployments.

In addition, training or serving Llama 2 at scale typically requires model parallelism to distribute model parameters across multiple GPUs. In this work, we focus on a data-parallel training paradigm, under which all workloads are assumed to follow a consistent execution model. In contrast, model-parallel approaches introduce fundamentally different execution patterns and communication structures, which fall outside the scope of our current modeling framework.

Including Llama 2 under these constraints would therefore introduce inconsistencies in workload calibration and compromise the validity of performance comparisons. To maintain methodological consistency, we exclude Llama 2 from the primary experimental setting.

Nevertheless, given its practical importance, we include Llama 2 in this appendix as an extended workload configuration. For Llama 2 jobs, we estimate the computational cost using the same method described in Section 3.4.1. Specifically, the job length is approximated by

$$\text{FLOPs} \approx 6MD,$$

where $M$ denotes the number of model parameters and $D$ denotes the number of processed tokens. The memory estimation for Llama 2 is also performed consistently with the methodology described in Section 3.4.1, with extrapolation based on available profiling data.

Following the same methodology described in the main text, we obtain the results shown in Table 7.

We observe that all three metrics, namely mean JCT, maximum waiting time, and deadline miss rate, increase under the extended workload setting that includes Llama 2. This is likely because Llama 2 is substantially larger than GPT-2. In our setting, the

Table 6: Extended configuration of representative training workloads in the synthetic dataset, including Llama 2.

| Job Type | Model | Dataset | Dataset Size | Batch Size | Epochs | Sequence Length / Input Length | GPUs |
|---|---|---|---|---|---|---|---|
| Image Classification | ResNet18 [68] | ImageNet [12] | $\leq$ 11,060,223 | 32–512 | 1–20 | N/A | 1–4 |
| Image Classification | Inception [69] | ImageNet [12] | $\leq$ 11,060,223 | 32–512 | 1–20 | N/A | 1–5 |
| Image Classification | VGG16 [70] | ImageNet [12] | $\leq$ 11,060,223 | 32–512 | 1–20 | N/A | 4–6 |
| Natural Language Processing | GPT-2 [71] | IMDB50K [72] | $\leq$ 50,000 | 2–128 | 1–20 | 64–128 tokens | 6–8 |
| Natural Language Processing | Llama 2 [86] | IMDB50K [72] | $\leq$ 50,000 | 16–64 | 1–20 | 64 tokens | 6–8 |
| Automatic Speech Recognition | DeepSpeech-2 [79] | LibriSpeech [74] | $\leq$ 104,014 | 8–128 | 5–20 | 10–20 s (audio) | 3–4 |

| Metric | Heuristic Models | | | | | RL | Opt. | Ours |
|---|---|---|---|---|---|---|---|---|
| | SJF | FIFO | Backfill | Tiresias | EDF | $DL^2$ | FFT | SARL |
| Mean JCT (TU) | 8635.3 | 8816.3 | 8902.2 | 9173.5 | <u>8525.9</u> | 11061.5 | 10033.1 | **7241.7** |
| Max Wait (TU) | 5246.8 | 4289.8 | **3843.6** | 4469.6 | 8042.6 | 5107.8 | 16480.3 | <u>4083.9</u> |
| Miss Rate (%) | 48.4 | 40.0 | 36.6 | 34.5 | <u>31.4</u> | 58.0 | 56.0 | **25.2** |

Table 7: Comparison of different scheduling methods under the normal preemption cost setting. Results are averaged over 10 test seeds. **Bold** indicates the best result, and <u>underline</u> indicates the second best. Lower values indicate better performance for all metrics.

Llama 2 model has 7B parameters, whereas GPT-2 has approximately 1.5B parameters, resulting in longer estimated job lengths and higher resource demands. Consequently, the inclusion of Llama 2 makes the overall scheduling problem more challenging, leading to higher JCT, waiting time, and miss rate across methods. However, the overall trend remains consistent with that reported in the main methodology chapter, and SARL still achieves the best overall performance among the compared schedulers.

# D Memory Disaggregation and UVM as Future Directions

While this thesis focuses on scheduling policies under conventional GPU memory constraints, recent advances in memory disaggregation and unified memory mechanisms suggest promising directions for future research. In particular, techniques such as remote memory access and Unified Virtual Memory (UVM) introduce new opportunities for alleviating memory bottlenecks in multi-GPU systems.

Existing studies have demonstrated the potential of memory sharing in CPU-centric environments. For instance, FaMAS [87] enforces partial local memory usage and offloads excess demand to a remote memory pool, effectively increasing system concurrency and reducing job completion time. These results indicate that memory disaggregation can fundamentally reshape resource constraints in distributed systems.

However, extending such mechanisms to GPU environments remains challenging. Deep learning workloads exhibit significantly higher memory bandwidth requirements compared to CPU workloads, making traditional remote memory access (e.g., RDMA) less effective due to its limited bandwidth and high latency. As a result, large-scale GPU memory disaggregation is still an open problem.

An alternative approach is to leverage hardware-supported mechanisms such as Unified Virtual Memory (UVM), GPUVM, and GPU Direct Storage. These techniques enable GPUs to access memory beyond their local capacity, either from peer GPUs or host memory, without explicit data movement management. This abstraction introduces a new form of memory elasticity, which could potentially be exploited by scheduling algorithms.

Despite these advantages, UVM-based memory sharing introduces several non-trivial trade-offs. Memory accesses to remote locations incur additional latency and may suffer from NUMA effects. Furthermore, the performance of UVM depends heavily on hardware topology (e.g., NVLink vs. PCIe) and access patterns, making it difficult to model accurately in scheduling decisions.

Recent work such as HUVM [88] demonstrates that carefully designed memory managers can improve utilization by coordinating data placement across GPUs and host memory. However, these systems typically operate at the memory management layer and remain largely decoupled from scheduling policies.

**Implications for Scheduling.** These observations suggest that future schedulers should not treat memory as a static per-GPU constraint, but rather as a dynamic and shareable resource. In particular, integrating memory-aware mechanisms into scheduling decisions introduces several research opportunities:

- **Memory-aware job placement:** Scheduling decisions could consider not only GPU availability but also the global memory state, enabling jobs to utilize remote

or shared memory when necessary.

- **Joint scheduling and memory management:** Instead of treating memory management and scheduling as separate layers, a unified framework could jointly optimize both aspects, potentially improving overall system efficiency. Recent work such as FaMAS [87] has already demonstrated the effectiveness of decoupling memory from compute resources in CPU environments by introducing a disaggregated memory pool. Although such designs have not yet been widely adopted in GPU systems, they highlight a promising direction for extending memory-sharing paradigms to GPU clusters.

- **Topology-aware scheduling:** Given the heterogeneous bandwidth and latency across NVLink, PCIe, and RDMA, schedulers could incorporate communication cost models when allocating GPUs and memory resources.

- **Learning-based policies:** The complex trade-offs introduced by memory sharing (e.g., latency vs. capacity) make this problem well-suited for reinforcement learning approaches, such as extending SARL to incorporate memory-related state and actions.

**Connection to This Work.** When large-scale GPU memory sharing or borrowing becomes practically feasible, scheduling policies should evolve accordingly. In such environments, memory can no longer be treated as a strictly local per-GPU constraint. Instead, schedulers may need to account for shared or remotely accessible memory resources when making placement and preemption decisions. This shift would introduce a new scheduling dimension, where the trade-off between memory capacity, communication overhead, and execution efficiency must be jointly considered.

Overall, incorporating memory disaggregation into scheduling remains an open and promising research direction. We leave the integration of memory-sharing upon UVM mechanisms as future work.

### D.1 UVM

Traditional GPU programming treats CPU and GPU memory as separate physical spaces, which requires explicit memory allocation and data movement. As a result, applications are typically constrained by the local physical memory of each GPU, and out-of-memory failures may occur when the working set exceeds device capacity.

To alleviate this limitation, modern GPU vendors have introduced Unified Virtual Memory (UVM), which provides a unified address space across CPUs and GPUs. Under UVM, data can be migrated transparently between local GPU memory, host memory,

and, in some systems, peer GPUs. This reduces programming complexity and enables a degree of memory elasticity for memory-intensive workloads.

However, UVM does not eliminate the cost of remote memory access. Its effectiveness still depends on hardware support, device compatibility, and interconnect performance. Accessing remote memory may incur substantial latency and bandwidth penalties, especially when host memory is involved.

Despite these limitations, UVM is highly relevant to scheduling research. It suggests that GPU memory need not always be modeled as a strictly local constraint. Instead, future schedulers may need to jointly consider computation, communication, and remotely accessible memory resources.

A representative example is HUVM, proposed by Choi et al.[88], which improves memory utilization in NVLink-connected multi-GPU servers by borrowing underutilized memory from neighboring GPUs. Their results show that limited remote GPU memory can still provide meaningful performance gains for memory-intensive workloads. Nevertheless, such mechanisms operate mainly at the memory-management layer and remain largely invisible to existing schedulers.

This observation motivates an important future direction for our work. While node-local memory sharing may improve utilization within a machine, a cluster-level scheduler must still decide where jobs should be placed and how communication overhead should be controlled. Integrating UVM-aware memory sharing into semantic-aware scheduling therefore remains a promising topic for future research.