Enhancing Deep Learning Schedulers with Large Language Models

Anonymous Authors, Paper #474

Abstract

Deep learning (DL) schedulers are pivotal in optimizing resource allocation in GPU clusters. However, existing DL schedulers face significant challenges: high profiling overhead, unreliable duration estimation, inadequate failure handling, and limited observability. To this end, we propose SCHEDMATE, a novel LLM-powered framework that enhances existing DL schedulers without requiring modifications to user code or scheduling frameworks. SCHEDMATE strategically extracts and utilizes information from three often overlooked sources: code scripts, runtime logs, and historical jobs, to improve scheduling decisions across the entire job lifecycle. It features three modules: a Scheduling Advisor that analyzes code and history to eliminate profiling and improve duration estimation, a Metric Tracker that non-intrusively monitors logs for real-time observability, and a Failure Handler that diagnoses failures and automates recovery from infrastructure issues. Our implementation integrates seamlessly with state-of-the-art schedulers. Evaluations on production traces show SCHEDMATE reduces average job completion times (JCT) by up to 1.91×, substantially cuts profiling overhead, and improves failure recovery, demonstrating the practical impact of overlooked data for DL scheduling.

1 Introduction

Deep Learning (DL) has experienced unprecedented growth in recent years, with large language models (LLMs) like GPT-4 [3], LLaMA-3 [31], and Qwen-2.5 [46] pushing the boundaries of AI capabilities across various domains. This growth drives substantial demand for computational resources, particularly GPU clusters [19, 20, 24]. DL schedulers are critical in orchestrating the execution of diverse and resource-intensive workloads within these clusters.

Various works have been proposed to efficiently optimize job completion time and overall resource utilization [15, 16, 19–21, 23, 35, 38, 50, 51, 54]. However, most of them are designed with simplified assumptions and face unforeseen or undesirable challenges:

- **C1**: ***High Profiling Overhead***. To make optimal scheduling decisions, job profiling is widely used by schedulers [21, 23, 33, 35] to gather workload features, hardware metrics, and progress information. However, the profiling process is time-consuming and resource-intensive. For example, the

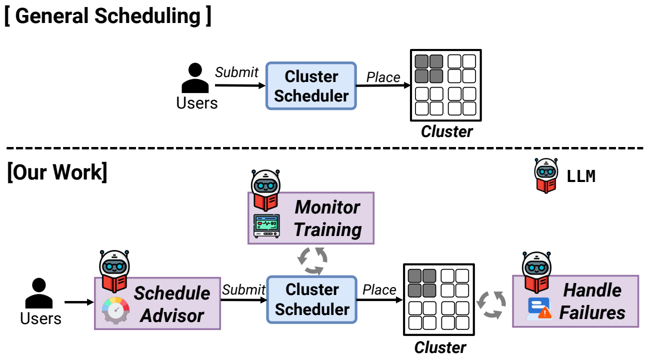


Figure 1: Existing DL schedulers workflow (top) and scheduling with SCHEDMATE (bottom).

state-of-the-art scheduling system Sia [23] requires extensive per-job profiling across heterogeneous GPUs and different batch size configurations to fit a throughput model for each job. Additionally, as shown in Figure 2, the majority of jobs in DL clusters are short-term. For instance, the LLM training dedicated cluster Acme [20] has a median job duration of 2 minutes. Profiling each job incurs an overhead nearly equivalent to actually finishing the job, which significantly amplifies the impact of profiling on cluster efficiency. Besides, both distributed training of LLMs and its profiling require hundreds of GPUs, incurring unacceptable GPU time overhead.

- **C2**: ***Unreliable Duration Estimation***. Job duration is important for optimizing scheduling decisions in many DL scheduling policies [11, 19, 21, 33, 35, 38, 48]. Some works [33, 35, 37] estimate a job duration by simply calculating $step\_time \times total\_steps$. Others estimate the duration by training a ML model using historical job metadata. However, they are unreliable in two aspects: (1) **Job cancellations and failures**: many jobs encounter early-exit due to manual cancellations or failures, as shown in Figure 2 b&c, where up to half of the jobs terminate with incomplete status (Acme [20], Philly [24], and Helios [19]). As a result, the first method can mislead scheduling decisions, causing sub-par scheduling performance and inefficient resource utilization. (2) **Inadequate features**: the ML-based methods only leverage basic metadata (e.g. job name, username, submission time) for model training due to limit datasources of schedulers, constraining the accuracy of duration estimation.
- **C3**: ***Inadequate Failure Handling***. Job failures frequently occur during training and cause significant delays in model

development [20, 24, 31, 56]. However, many state-of-the-art DL schedulers [21, 23, 35, 38] usually ignore failure handling. This oversight can lead to inefficient resource utilization and delays in training completion, particularly for large-scale distributed training jobs like LLMs [20]. For instance, the pre-training of LLaMA-3.1 [31] encounters 419 failures, with training failing on average every 3 hours. Fast recovery can significantly improve resource efficiency.

- **C4**: ***Limited Observability***. DL jobs are black-boxes to most real-world DL schedulers [20, 24, 48], delivering only limited observability. Schedulers usually only have access to basic scheduling metadata (e.g. job name, username, submission time) and hardware metrics. These limitations constrain the scheduler's ability to make optimal scheduling decisions. For example, they cannot detect performance issues or identify the task (e.g. computer vision, NLP, robotics) of a job. Improving the observability of DL jobs can help schedulers make more informed scheduling decisions, especially to non-intrusive schedulers [21].

Given the above challenges, we propose SCHEDMATE, a novel framework designed to enhance the performance of DL schedulers. Our key insight is that the critical information needed to overcome these limitations already exists in often overlooked data sources: training logs, source code, and historical jobs. This insight directly addresses the core challenges: *Source code*, available immediately upon job submission, contains rich workload characteristics (model architecture, hyperparameters, dataset specifications) that can eliminate high profiling overhead and improve duration estimation. *Runtime logs*, generated during execution, provide real-time training progress, performance metrics, and error messages that enhance observability and enable efficient failure handling. By strategically extracting and analyzing this information, we can significantly improve scheduling decisions of existing DL schedulers.

SCHEDMATE enhances scheduling performance through three LLM-powered modules that extract and interpret information from source code and logs. As Figure 1 shows, SCHEDMATE integrates into the standard job scheduling workflow with: (1) The *Scheduling Advisor*, which analyzes job source code to extract workload characteristics, creates vector embeddings, and identifies similar historical jobs for informed scheduling decisions—addressing challenges **C1** and **C2** by eliminating extensive profiling and improving workload prediction; (2) The *Metric Tracker*, which non-intrusively extracts valuable metrics from training logs using embedding-based classification and LLMs, solving **C4** and enabling dynamic scheduling adjustments; and (3) The *Failure Handler*, which employs binary search to locate error messages, identify root causes, and automatically recover jobs from infrastructure-related issues—addressing **C3**, minimizing downtime, and improving resource utilization. SCHEDMATE significantly enhances existing DL schedulers without requiring modifications to user code or frameworks.

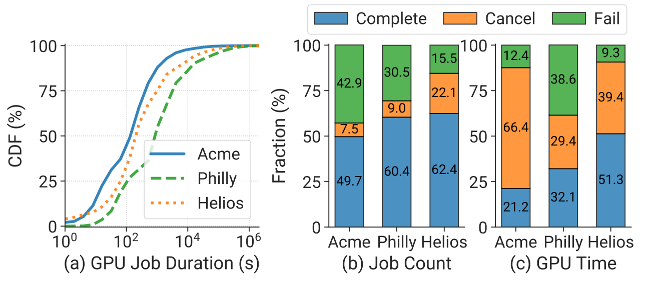


Figure 2: **Background**: DL workload characteristics across Microsoft Philly [24], SenseTime Helios [19] and Acme [20] clusters. (a) CDF of the job duration. (b, c) Final statuses of jobs in terms of quantity and utilized GPU resources.

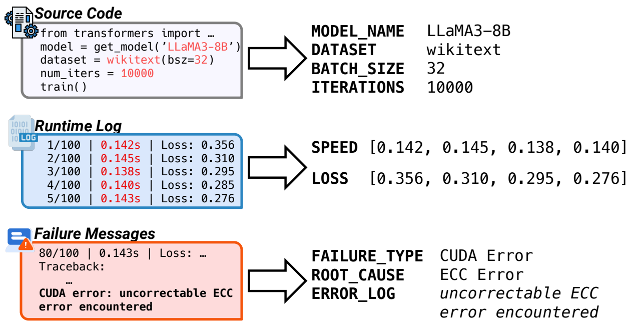
We implement SCHEDMATE as a plug-and-play module compatible with several state-of-the-art DL schedulers, such as Sia [23] and Lucid [21]. Our evaluation demonstrates that SCHEDMATE significantly improves scheduling performance, reducing job completion times by up to 48% compared to the state-of-the-art DL scheduler Lucid. Additionally, we perform extensive simulations and 64-GPU physical experiments to evaluate the efficiency of SCHEDMATE on various production cluster traces [19, 20, 24]. Our results demonstrate that SCHEDMATE can effectively address the challenges **C1**∼**C4** and improve the scheduling performance of existing DL schedulers.

2 Background and Motivation

2.1 DL Workload Scheduling

Existing DL Schedulers. DL training jobs are usually submitted to multi-tenant GPU clusters [19, 20, 24, 48]. Figure 1 shows common paradigms of DL workload scheduling [14, 48, 55]. Initially, users submit their jobs to a scheduler, which is responsible for managing the jobs for execution in the cluster. The scheduler makes scheduling decisions based on certain scheduling policies, such as First-In-First-Out (FIFO) and Shortest Job First (SJF) [14]. To accomplish cluster-wide goals like maximizing resource utilization or minimizing job completion time (JCT), some advanced features are adopted in the policies of DL schedulers, such as *job-packing* [21, 51], which runs multiple jobs on the same GPU(s), *elastic-training* [23, 38], which dynamically adjusts the job's resource allocation, and *heterogeneous-aware policies* [5, 23, 35], which further consider the GPU heterogeneity.

Workload Estimation. Effective DL scheduling requires estimating workload characteristics, such as duration and resource utilization. Schedulers gather necessary metrics via pre-profiling (briefly running jobs beforehand, leveraging their iterative nature [21, 35]) or online profiling (collecting metrics during execution [23, 38]). Two main duration estimation approaches are used: (1) *Config-based* methods calculate $step\_time \times total\_steps$ [11, 33, 35, 37, 38], but are unreliable under high failure/cancellation rates. (2) *History-based* methods exploit recurring job submission patterns [19, 20, 48] by using historical data, either training predictive models on metadata [19, 21, 48] or identifying similar past jobs [22, 36].



```
from transformers import …
model = get_model('LLaMA3-8B')
dataset = wikitext(bsz=32)
num_iters = 10000
train()
```

```
MODEL_NAME   LLaMA3-8B
DATASET      wikitext
BATCH_SIZE   32
ITERATIONS   10000
```

```
1/100 | 0.142s | Loss: 0.356
2/100 | 0.145s | Loss: 0.310
3/100 | 0.138s | Loss: 0.295
4/100 | 0.140s | Loss: 0.285
5/100 | 0.143s | Loss: 0.276
```

```
SPEED [0.142, 0.145, 0.138, 0.140]

LOSS  [0.356, 0.310, 0.295, 0.276]
```

```
80/100 | 0.143s | Loss: …
Traceback:
    …
CUDA error: uncorrectable ECC
error encountered
```

```
FAILURE_TYPE  CUDA Error
ROOT_CAUSE    ECC Error
ERROR_LOG     uncorrectable ECC
              error encountered
```

Figure 3: Examples of the information available in source code and runtime logs.

Prevalence of Failures. Recent studies [20, 24, 25, 31] report that DL training jobs suffer from frequent failures, especially infrastructure failures. Acme [20] reports around 2 infrastructure failures per day on average during LLM development. ByteDance [25] experienced over 100 failures during a multi-week LLM pretraining task. As shown in Figure 2(c), it is obvious that only approximately 20∼50% of resources are consumed by jobs that are finally complete, demonstrating the importance of failure handling.

2.2 Motivation and Opportunities

Workload Similarity. Workload similarity of different DL jobs comes from two factors. Firstly, recurrent jobs, which has been studied in several works [19, 36, 48]. Secondly, different jobs may also share similar workload characteristics and duration distribution, even if they are not identical. For instance, different LLM pretraining jobs may share similar workload. Leveraging this similarity can potentially enhance workload estimation. However, existing schedulers only leverage basic metadata, which fails to fully explore this information.

Overlooked Information for Schedulers. Existing DL schedulers fail to fully exploit a wealth of valuable information generated from different sources. This includes (1) source code of jobs, (2) runtime logs, and (3) historical jobs. These data sources are available in most local or DL clusters, and can be easily accessed by schedulers. Figure 3 provides examples of first two types of information available in both source code and logs. By leveraging them, we demonstrate that it is possible to effectively address key challenges (**C1**-**C4**) and significantly enhance the performance of existing schedulers.

- ***Workload-related Metadata from Source Code.*** One insight of our work is that the source code of jobs contains rich information about the workload characteristics, such as model architecture, training task, and dataset settings. With the running command/script and working directory of the job, we can easily extract the workload metadata from the source code. This workload-related metadata can be used for jobs similarities analysis, based on which we can improve both the duration estimation and resource utilization prediction. Thus addressing the challenge **C1** and **C2**.

- ***Metrics from Logs.*** Non-intrusive deep learning schedulers,

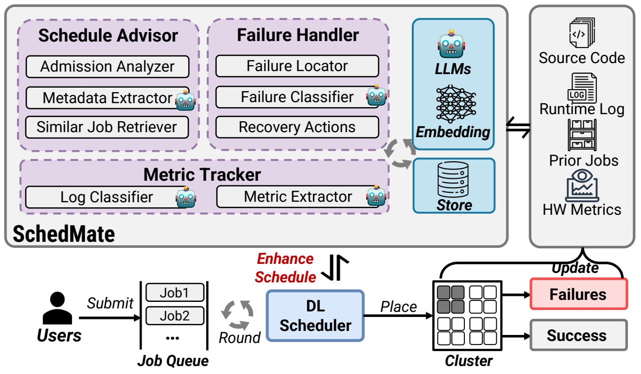


Figure 4: **System Overview of SCHEDMATE** The system integrates with four data sources (prior jobs, source code, runtime log, and hardware metrics). The modules attached with robot symbols utilize LLMs.

such as Lucid, reduce deployment costs by avoiding modifications to user code or frameworks. However, such an approach limits access to real-time training metrics, such as loss rates and throughput, which are crucial for optimal scheduling decisions. Potentially, with these metrics, schedulers can further optimize scheduling performance, e.g., identify underperforming jobs or adjust resource allocations based on actual training progress. Addressing this observability challenge **C4** by incorporating training metrics can enable schedulers to make more informed decisions, dynamically reallocate resources, and improve overall system efficiency while maintaining a non-intrusive approach.

- ***Root Cause from Failure Messages.*** Among the job failures in DL cluster, the program or user script issues, such as syntax errors and code bugs, are the most common [20, 24]. In contrast, infrastructure failures, while less frequent, tend to be more severe and exhibit longer runtime-to-failure (RTF) times [20]. Regarding failure recovery, programmatic issues necessitate manual intervention, whereas most infrastructure failures can typically be resolved without requiring code modifications [20, 24, 26]. This distinction inspired our design of a system that leverages LLMs to analyze job logs, identify failure types, and automatically recover from infrastructure failures, thereby addressing challenge **C3** (§3.3).

Leveraging LLMs. All the aforementioned data sources are unstructured or semi-structured texts, making it challenging to extract useful information. Fortunately, LLMs excel at processing such unstructured data. By utilizing LLMs, we can effectively extract crucial information from them, thereby overcoming limitations faced by existing scheduling approaches.

3 SchedMate Overview and Design

Design Principles. We design SCHEDMATE based on three core principles: **(1)** ***Non-intrusiveness***. SCHEDMATE operates non-intrusively, requiring no code or process modifications. It leverages existing datasources for enchancing scheduling performance. This ensures adoptability in various environments, especially local development clusters [19,20,24,48], and different scheduling policies. **(2)** ***Cost-effectiveness***. We minimize LLM overhead by constraining token consumption and model scale while maintaining performance through efficient embedding-based filtering and targeted prompting. **(3)** ***Robustness***. The semi-structured nature of the utilized datasources introduces additional challenges to LLMs. Each component handles diverse data distributions (e.g., varying log formats, code styles, metadata) where rule-based methods fail.

System Overview. Figure 4 presents the architecture of SCHEDMATE. It enhances DL schedulers by integrating information from four key data sources: source code, runtime logs, historical job data (prior jobs), and hardware metrics. SCHEDMATE consists of three main modules: the *Scheduling Advisor*, the *Metric Tracker*, and the *Failure Handler*. The *Scheduling Advisor* analyzes source code and historical data to provide workload insights for improved scheduling. The *Metric Tracker* monitors runtime logs non-intrusively to extract performance metrics, offering real-time observability. The *Failure Handler* analyzes logs to diagnose job failures and facilitate recovery. These modules leverage LLMs and embedding techniques, supported by a KV store.

Retrieval-Augmented Scheduling. To improve the accuracy of workload estimation and provide other insights for scheduling, we propose to utilize the workload metadata to retrieve similar jobs using the *Scheduling Advisor*. For efficient metadata extraction from source code, we developed an LLM-based agent using specialized filesystem tools. The agent reasons to navigate the file structure, select files to read, and summarize workload metadata. The detailed design and workflow of the metadata extraction process are discussed in §3.1.

Progress-aware Decision Making. The *Metric Tracker* uses a two-stage approach to extract performance metrics from job logs. First, the *Log Classifier* uses an embedding model to categorize and filter relevant log lines. Then, the *Metric Extractor* uses an LLM (with techniques like JsonFormer [1]) to parse filtered lines and extract metrics. This provides observability for real-time scheduler optimizations (§3.2).

LLM-based RCA & Failure Handle. Infrastructure failures significantly impact model development [20,24]. We propose an automated method for root cause analysis (RCA) and remediation. We use efficient embedding-based binary search for initial root cause location, followed by LLM-based failure classification. The system then automatically selects recovery actions for infrastructure failures. This reduces manual intervention, resource idleness, and training delays (§3.3).

3.1 Scheduling Advisor

The *Scheduling Advisor* provides workload insights by retrieving similar historical jobs, avoiding intensive profiling.

Workflow. As shown in Figure 5, when a job is submitted, the *Admission Analyzer* initially matches the job with recent jobs. Otherwise, the *Metadata Extractor* extracts the workload metadata from the source code using an LLM-based agent. The resulting metadata is then leveraged to retrieve similar historical jobs, providing insights into workload characteristics and enabling more informed scheduling decisions.

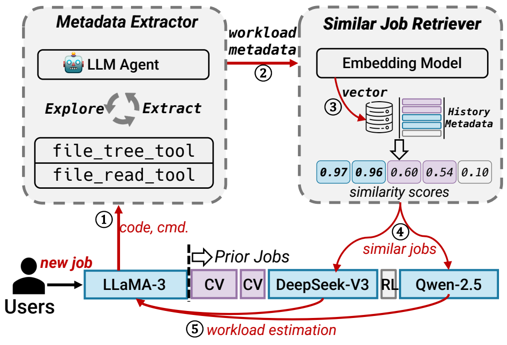


Figure 5: **Scheduling Advisor Workflow.** The LLM-based agent leverages two tools to support the efficient metadata extraction from source code. Length of each job refers to the duration. The red line indicates the data flow path.

Workload Metadata. We describe what we actually want to extract from the source code. As projects can be large, we extract only the most relevant workload metadata. We focus on extracting the following metadata: (1) *Model Architecture*, which includes the model name, model type, and task type (e.g. NLP, CV); (2) *Dataset Settings*, which includes the dataset name, dataset size, and data augmentation techniques; and (4) *Training Configuration*, which includes training framework, distributed training settings, and hardware configuration. We predefine core metadata fields for broad utility but allow the agent to dynamically extract job-specific fields.

Admission Analyzer. Upon submission, the *Admission Analyzer* first tries matching the job with the user's recent jobs based on working directory, command, and git branch. If no match is found, it is sent to the *Metadata Extractor* for further analysis. If historical jobs are found, the job is directly treated as a resubmission, and the matched jobs are used to estimate the job's workload. This reduces the load of the following components, *Metadata Extractor* and *Similar Job Retriever*.

Metadata Extractor. It is designed to analyze the job's source code and extract the workload metadata. Workload metadata is often scattered across files, making extraction from fixed locations (e.g., `main.py`) difficult. Processing the entire codebase is impractical due to its size.

We address this using a tool-based ReAct-style agent [53]. We adopt two tools to provide observability into the filesystem: `file_tree_tool` and `file_read_tool`. The former outputs the file tree using the Unix `tree` command. We selectively output relevant file types (e.g., `.py`, `.sh`) and exclude transient data (e.g., logs, cache). The `file_read_tool` opens a file and sends the entire content to the agent. This tool allows the agent to access file content.

A condensed version of the prompt is shown in Figure 6. The agent is injected with four prompts: (1) *task requirements prompt*, which specifies the basic background and details of the task; (2) *ReAct prompt*, which provides guidance tools

Prompt for Metadata Extraction

You are a helpful AI assistant. Answer the following questions as best you can. You have access to the following tools: `TOOLS INSTRUCTION`
Task Requirements. Follow the ReAct-style instruction [53] to find the answer based on the repository's path, command, and launch configuration. Response in a structured format.
Question.: What is the training, model, and dataset metadata?

```
training command: python -m train experiment=pile/gpt
repository path: /path/to/project
launch configuration: ngpu=8, ...,nnodes=1
```

Workload Metadata Sample

```
model_config*       training_config*    dataset_config*
  model_name*: GPT    iters*: 80000       train*: pile
  task_type*: NLP     iter_type*: step    valid*: pile-val
  d_model: 768        batch_size: 256     num_workers: 4
  n_layer: 12         lr: 0.001           ...
```

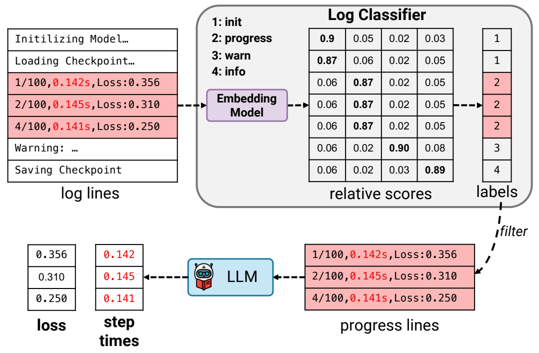
Figure 6: Prompt template of *Metadata Extractor* and a workload metadata sample. Fields with * are required.

manual and the thought-action-observation response rule; (3) *metadata scheme,* which defines the fields of the metadata, such as the model name, dataset settings, and hyperparameters; and (4) *formatting instruction*, that ensures the extracted data is organized in a JSON format, facilitating easy integration with other system components. We predefine a few universal output fields (marked * in Figure 6) common in DL workflows. We encourage the agent to extract relevant dynamic job-specific metadata.

As shown in Figure 5, the agent starts with the job's working directory and command. It then uses the `file_tree_tool` to survey the file structure, identifying potentially relevant files based on their names. For each identified file, the agent utilizes `file_read_tool` to access and analyze its contents. This process continues iteratively, with the agent reasoning about the gathered information at each step and deciding whether to explore further or conclude the search. Typically, the agent begins with the launch script, which can be determined by the running command. The extraction process concludes when the agent determines it has collected sufficient metadata or exhausted all relevant sources. Finally, the agent organizes all gathered information into a structured JSON format, providing a comprehensive metadata summary for the job. This iterative and adaptive approach allows the agent to efficiently extract workload metadata.

Notably, due to randomness in jobs and LLM outputs, we don't directly use specific fields of the extracted metadata. Instead, we use a similarity-based embedding model.

Similar Job Retrieval. To match extracted metadata with historical jobs, we use a similarity-based approach. First, workload metadata is converted to a string and then to a dense vector via an embedding model. We then compute the *cosine similarity* between the current job's vector and those of historical jobs. Jobs with similarity scores above a threshold (T_{score}) are considered matches, and we select the *top-k* jobs



```
Initilizing Model…
Loading Checkpoint…
1/100,0.142s,Loss:0.356
2/100,0.145s,Loss:0.310
4/100,0.141s,Loss:0.250
Warning: …
Saving Checkpoint
```

0.9	0.05	0.02	0.03	1
0.87	0.06	0.02	0.05	1
0.06	**0.87**	0.02	0.05	2
0.06	**0.87**	0.02	0.05	2
0.06	**0.87**	0.02	0.05	2
0.06	0.02	**0.90**	0.08	3
0.06	0.02	0.03	**0.89**	4

```
1/100,0.142s,Loss:0.356
2/100,0.145s,Loss:0.310
4/100,0.141s,Loss:0.250
```

loss	step times
0.356	0.142
0.310	0.145
0.250	0.141

Figure 7: **Metric Tracker Workflow**. The progress lines are blended with some initialization and warning logs.

with the highest similarity scores as the most relevant similar jobs. This semantic matching efficiently identifies jobs with similar characteristics, enhancing retrieval-augmented scheduling. For example, as shown in Figure 5, the *Similar Job Retriever* retrieves the top-2 similar LLM training jobs based on the metadata of the current LLM job. The retrieved jobs are then used to estimate the workload of the current job, such as duration and resource utilization.

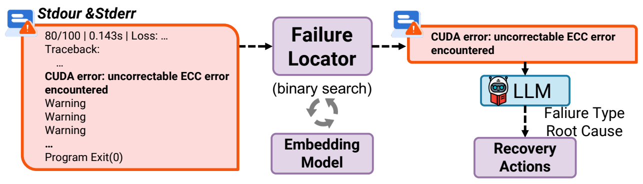
3.2 Metric Tracker

The *Metric Tracker* extracts critical training performance metrics (e.g., step time, loss) from job logs without requiring system modifications. This addresses a fundamental challenge for non-intrusive schedulers: gaining visibility into training progress metrics that are typically embedded within verbose logs. However, these 'progress logs' are often mixed with irrelevant logs (e.g., initialization, warnings). Direct LLM analysis of entire logs is computationally prohibitive due to their size. Our solution employs a novel two-stage approach: first, an efficient embedding-based filter rapidly identifies and isolates relevant progress lines from noise; second, an LLM parses these filtered lines to extract structured metrics. This design maintains low latency and high robustness across diverse logging formats, which is a critical requirement for production deployment.

Log Classifier. Job logs contain diverse information and *Log Classifier* is responsible for categorizing one log line or a chunk of logs into one of these five categories: (1) error log, (2) warning log, (3) training progress, (4) other normal output like initialization and debug information. Classification starts by converting log lines/chunks and category text into dense vectors using an embedding model, capturing semantics for comparison. The classification process computes the similarity score S_{ij} as the cosine similarity between log line vector $\mathbf{v}_i$ and category vector $\mathbf{u}_j$:

$$S_{ij} = \frac{\mathbf{v}_i \cdot \mathbf{u}_j}{||\mathbf{v}_i|| \cdot ||\mathbf{u}_j||} \tag{1}$$

This score quantifies the semantic alignment between each log line and potential categories, normalizing for vector magnitude. Each log line is assigned to the category yielding



```
80/100 | 0.143s | Loss: ...
Traceback:
    ...
CUDA error: uncorrectable ECC error
encountered
Warning
Warning
Warning
...
Program Exit(0)
```

```
CUDA error: uncorrectable ECC error
encountered
```

Figure 8: **Failure Handler Workflow**. The emphasized log line is the first failure we aims to locate.

the highest similarity score, enabling efficient and accurate filtering of relevant training progress information from noise.

Metric Extractor. The *Metric Extractor* is designed to extract metrics from logs. Developers usually utilize it to monitor the training (e.g. "`[10:25:30] Epoch 1/10: Training loss: 2.345, Step time: 0.18s`"). Varied log formats make direct regex extraction challenging. Moreover, real-world logs often contain noise (e.g., warnings, failures). To address this issue, we propose a two-stage method to extract the desired metrics. We first categorize log lines in reverse order using the *Log Classifier*. We process logs iteratively until finding N training progress lines. Analyzing a few tens of lines is typically sufficient to obtain accurate step times while maintaining efficiency, which can be done in less than 1s. In the second stage, an LLM extracts step times from filtered lines. We also employ JsonFormer [1] to ensure structured floating-point output from the LLM. By filtering out irrelevant lines in the first stage, the input size of the LLM is reduced. As embedding models are much faster than LLMs, this method is more efficient than direct LLM parsing.

Partial Information Discussion. *Metric Tracker* may fail due to absence of progress lines within the logs. Therefore, we recommend that users ensure the integration functions without causing scheduling performance degradation in this scenario of partial information.

3.3 Failure Handler

In distributed training environments, we focus on addressing **infrastructure failures** that significantly impact job completion. Unlike software errors that require developer intervention, infrastructure failures (e.g., GPU, network, or node issues) can be automatically remediated by isolating faulty components and restarting jobs. The *Failure Handler* employs sophisticated log analysis to specifically distinguish infrastructure failures from other error types, enabling automatic recovery. This module overcomes one key challenge: identifying root causes within cascading error patterns where initial failures trigger numerous secondary errors. Our solution combines efficient binary search for precise failure localization with LLM-based classification for automated remediation.

Challenges. Leveraging LLMs to identify the root cause of failures in DL jobs faces significant challenges. Firstly, error patterns are complex [20, 24], with a single initial failure triggering numerous cascading errors. Secondly, log volumes are substantial. Figure 9(a) shows that full logs from Acme reaching 256K lines with 50% surpassing 16K. Moreover,

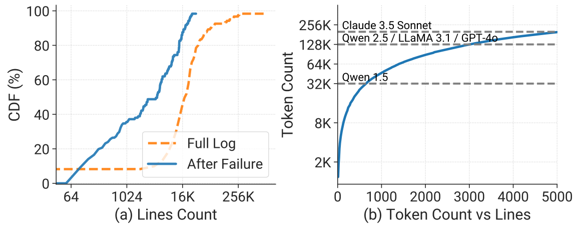


Figure 9: **Log Volume v.s. Token.** (a) CDF of full log and post-failure logs' line count. (b) Token count versus log lines, marking four LLMs' context.

even post-failure logs span thousands of lines with only 40% lower than 1K ($\sim 45K$ tokens). However, Figure 9(b) highlights that even a few thousand lines of error logs can easily surpass the context window limitations widely used LLMs, hindering comprehensive analysis.

Failure Locator. To handle noisy logs efficiently, we employ a binary search approach combined with the *Log Classifier*. We first split the entire log into chunks (each with 20 lines) and apply binary search to locate the first occurrence of an error message, which likely indicates the root cause. Our approach leverages a key insight: in typical failure logs, entries before the root cause are non-error logs (e.g., progress lines and initialization), while entries after include cascading error messages. By classifying each chunk as either error or non-error using our *Log Classifier*, we can efficiently pinpoint the transition point where errors first appear.

Algorithm 1 details this binary search process. With a time complexity of $O(\log N)$ where N is the number of log chunks, this method dramatically outperforms linear scanning approaches that would require $O(N)$ operations. In practice, this enables us to locate root causes in logs with tens of thousands of lines in just a few *Log Classifier* invocations, making the process highly efficient even for large-scale failures.

Failure Classification. After locating the error message, we use the LLM to parse the very log chunk around the located position (e.g. 500 lines), and classify the failure into one of the predefined failure types. The categories[1] include infrastructure failure, framework errors, user script errors/bugs, and other failures. If infrastructure failure is identified, the LLM will further classify the error into one of the predefined categories, including (a) GPU failure, (b) NVLink failure, (c) Network failure, (d) Node failure, and (e) Other failures. We set an error context windows size L_{err} for this failure classification process, which achieves a tradeoff between the accuracy of error message classification and the efficiency of the LLMs generation. Besides, we employ JsonFormer [1] to ensure JSON-format output with two fields: *Error Type*, and *Faulty Component*

Automated Recovery Actions. Upon detecting faulty component, SCHEDMATE autonomously selects and executes recovery actions from a predefined action set. For example, when a GPU failure is identified, the system first runs `nccltest` to

[1]This classification method refers to Acme [20]

Algorithm 1 Binary search method for filtering error lines

```
Require: Log $L$
Require: Embedding Model $E$
Ensure: Error Line Index
1: $S \leftarrow$ split $L$ into chunks
2: // Classes: 0-failure, 1-warning, 2-progress, 3-others
3: $log\_classifier \leftarrow init\_log\_classifier(\text{E})$
4: $start \leftarrow 0$
5: $end \leftarrow \text{len}(S) - 1$
6: while $start \leq end$ do
7:     $mid \leftarrow (start + end)//2$
8:     if $start \geq end$ then
9:         if $log\_classifier(S[start]) \leq 2$ then
10:            return $start$
11:        else
12:            return -1 // No failure found
13:    $mid\_class \leftarrow log\_classifier(S[mid])$
14:    if $mid\_class \leq 2$ then
15:        $end \leftarrow mid$
16:    else
17:        $start \leftarrow mid + 1$
18: return -1
```

pinpoint the faulty node. It then isolates the node, provisions a backup node, and restarts the job. This automated remediation pipeline eliminates manual intervention, significantly reduces downtime. If the action fails to recover the job, the system will notify the user and provide a detailed explanation of the failure, including the error type and the identified possible faulty infrastructure as a reference.

4 Integration Case Studies

4.1 Case Study 1: Non-intrusive Scheduling with Lucid

Lucid [21] is a non-intrusive DL scheduler that utilizes the job packing technique to minimize the JCTs. It profiles each job for a short time limit T_{prof}, gathers hardware metrics, and uses this information to determine the pairs of jobs to be packed. T_{prof} is set to surpass the cold-start time of jobs, encompassing the initialization and data movement time, to obtain hardware metrics after the training actually starts. In the meantime, it should be kept short to avoid high costs.

In practice, Lucid faces several challenges. Firstly, a typical model training usually undergoes two stages [29]: the warmup stage, during which the resource utilization and training throughput gradually increase, and the steady stage, where they turn stable. Therefore, even if the T_{prof} covers the actual model training, it might fall in the warmup stage and the collected metrics are unreliable. This would result in an *underestimation* of the job's resource utilization in Lucid's profiler. Empirical results in Lucid's paper show that two jobs with lower GPU utilization have less interference and are more likely to be packed. Consequently, this underestimation can result in heavy jobs being mistakenly packed together, leading to more interference and potential slowdowns in JCTs. In addition, large models are becoming popular. The cold-start time for their training is notably long due to the time required for checkpoint loading and data preparation [13, 20]. While simply increasing the T_{prof} for every job to ensure covering the steady stage can help mitigate this issue, it is not feasible due to the expensive overhead. Secondly, Lucid employs a workload estimate model to predict job duration. However, it only utilizes the basic scheduling metadata such as user ID, job name, and the number of GPUs. These metadata are less informative compared to the information from source codes and logs, which are used in SCHEDMATE. Finally, due to the interference effect, the packing mechanism can lead to a significant slowdown on packed jobs in some cases even with a robust algorithm. Lucid lacks the mechanism to deal with this issue. We propose to integrate SCHEDMATE into Lucid to address these challenges.

Bypass Profiling. We integrate our *Scheduling Advisor* into Lucid's profiler. As the jobs are running, we collect both the job duration and hardware metrics, including GPU utilization, and GPU memory footprint, and save them in a store. We use a KV store to store the job metadata and hardware metrics of all jobs. For each new job, we extract its metadata and use the *Scheduling Advisor* to retrieve $top-3$ most similar historical jobs ($similarity\_threshold = 0.90$). If $k > 0$, we estimate hardware metrics by calculating the average values of similar jobs. If $k = 0$, it falls back to the Lucid's profiler for profiling. Finally, the *Scheduling Advisor* feeds the estimated hardware metrics or profiler results to Lucid's job packing module to make packing decisions.

Enhanced Duration Estimation. We also integrate the *Scheduling Advisor* for duration estimation. We calculate the average duration of matched similar jobs as the estimated duration for the new job. If $k = 0$, it falls back to Lucid's ML-based workload estimator.

Interference-aware Packing Cancelation. To overcome the slowdowns of packed jobs, we propose an non-intrusive interference-aware packing cancelation mechanism that adopts the *Metric Tracker*. Specifically, we only trigger the *Metric Tracker* twice whenever a job is packed. We take two jobs, A and B, for example. Initially, Job A is running in the cluster. Then, Job B is scheduled and packed with Job A. We first read Job A's log generated previously and input it to the *Metric Tracker* to obtain TP_{before}, the throughput of Job A before packing. Then, after a certain period of time, we collect the newly generated log of Job A and trigger the *Metric Tracker* again to get TP_{after}, the throughput of Job A after packing. At this stage, we ensure that Job A has completed several tens of training steps after packing. Finally, if we detect that the slowdown rate ($\frac{TP_{after}}{TP_{before}}$) of Job A is lower than a threshold (0.65 in this case study), the scheduler stops Job B and returns it back to the job queue for rescheduling. Considering that the model training progress of the evicted jobs will be lost, we restrict that the evicted jobs cannot be packed

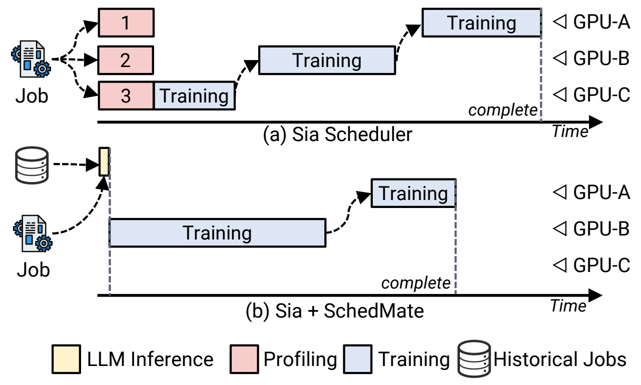


Figure 10: Job lifecycle in Sia and Sia+SCHEDMATE. The yellow block refers to the processing of *Scheduling Advisor*. GPU-{A,B,C} represents different types of GPUs.

again to avoid frequent eviction. Note that the eviction check is skipped for the specific job pairs where the job's progress cannot be tracked.

4.2 Case Study 2: Elastic Scheduling with Sia

Sia [23] is the state-of-the-art heterogeneity-aware, elastic-enabled scheduler designed to enhance JCTs and resource utilization in heterogeneous clusters. Sia begins each job by profiling it and then adopts a profile-as-you-go approach, using this information to make scheduling decisions. It records all throughput information under all placements and batch sizes encountered by each job. However, if the initial profiling process is conducted for every job, it can be both costly and time-consuming. To address this limitation, we have integrated SCHEDMATE into Sia, minimizing the profiling overhead by identifying resubmissions or similar jobs and bypassing their profiling process.

Preprofiling-free Bootstraping. We modify Sia's profiler. For each job, instead of profiling on GPus, we use the *Scheduling Advisor* to retrieve similar jobs in the historical data. If similar jobs are found, we bypass the profiling phase and directly utilize the throughput information of the matched jobs to fit the performance model. If no match is found, we profile the job in the original way. Once the job is scheduled, we continue to apply Sia's profile-as-you-go method, updating the job's throughput information.

Figure 10 illustrates this process. Vanilla Sia scheduler requires profiling on each GPU type before training, consuming additional resources. In contrast, our system leverages historical job data to bypass profiling, allowing training to start immediately and taking advantage of previous profiling results, which reduces overhead and improves cluster efficiency.

4.3 Case Study 3: Standalone Scheduler

We implement SCHEDMATE as a standalone scheduler following the vanilla Shortest Job First (SJF) policy. In this configuration, we employ both the *Scheduling Advisor* and *Failure Handler* modules. The standalone SCHEDMATE follows the vanilla Shortest Job First (SJF) policy. The scheduler

Table 2: **Physical Evaluation**: Comparison of policy performance between physical and simulated environments.

Policy	**Cluster**	**Avg. JCT**	**Makespan**
Lucid+SCHEDMATE	Physical	**3.37h**	**15.7h**
	Simulation	3.48h	14.1h
Lucid	Physical	4.39h	17.2h
	Simulation	4.34h	18.0h

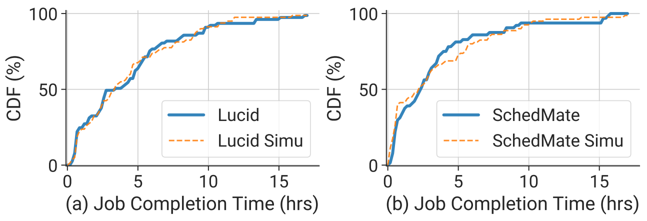


Figure 11: **Simulator Fidelity Validation.** JCT CDFs comparing simulation results (dashed lines) against physical cluster execution (solid lines) for (a) Lucid and (b) Lucid+SCHEDMATE.

maintains a database to store the scheduling metadata (submit time, end time, etc.) and workload fingerprint, extracted by Metadata Extractor, of historical jobs.

For each incoming job, the *Scheduling Advisor* retrieves the three most similar historical jobs and calculates their average duration to estimate the new job's runtime. The scheduler then sorts the jobs based on these estimated durations and schedules them accordingly. In the event of a job failure, the *Failure Handler* is triggered to analyze the root cause of the failure, take actions to mitigate the issue, and resume the job. This implementation allows us to evaluate SCHEDMATE's core capabilities in job duration estimation, efficient scheduling, and failure management independently of existing schedulers.

5 Evaluation

5.1 Experiment Setup

Implementation. We implement SCHEDMATE on top Ray [34]-based scheduling system with approximately 5,500 lines of code. SCHEDMATE is implemented as a library and can be integrated into existing schedulers. We deploy the core LLM as an OpenAI-compatible API service using vLLM [27]. For the embedding models, we use the FlagEmbedding library [6]. We use Redis [41] for storing the vectors, historical job metadata, and performing similarity searching. The system incorporates real-time GPU monitoring using NVIDIA Management Library [4], providing hardware metrics for profiling.

Testbed and Models. We conduct our physical experiments on 8 nodes in a SLURM [55] cluster. Each node is equipped with 8 NVIDIA A800-80GB GPUs and 2TB memory. For software, we use Python 3.9, CUDA 12.2, PyTorch 2.4.0, RAY 2.6.3, and vLLM 0.5.5 [27]. In this section, if not specified, we use BGE-m3 [6] embedding model and Qwen-2.5-7B-Instruct [45] (enable GPTQ [12] Int8 quantization) as the core models. Additionally, we deploy the embedding model

Table 1: Representative DL model training workloads for scheduler evaluation, categorized by scale.

Scale	Model	Dataset	Batch Size	Task	#Param.
Large (>1B)	LLaMA-3 [31]	Alpaca [44]	1∼4	Instruction Fine-tuning	8B
	Qwen-2.5 [52]	Alpaca	1∼4	Instruction Fine-tuning	14B
Medium (100M - 1B)	ViT-L/16 [10]	ImageNet-1k [8]	16∼64	Image Classification	307M
	BERT-Base [9]	SQuAD v1.1 [39]	16∼64	Fine-tuning	110M
Small (<100M)	ResNet50 [17]	ImageNet-1k [8]	32∼128	Image Classification	25.6M
	Unet2D [40]	BraTS [42]	8∼64	Image Segmentation	30M
	EfficientNet-B4 [43]	ImageNet-1k	32∼128	Image Classification	19M

on CPU and LLM on a single GPU using vLLM. We compare the efficiency and accuracy of different models in §5.5.

Traces. We use the following traces from various sources for comprehensive evaluation of SCHEDMATE:

- *Mars*: A production trace we collect from a 1,024 GPU cluster, containing 500 LLM training jobs (7B-100B parameters) with code, logs, and hardware metrics. These jobs utilize the same codebase with about 40K LoC.
- *Public Traces*: Widely-used public DL cluster traces including Acme [20], Philly [24], Helios [19] (Saturn and Venus), and Sia-trace [23]. Used for end-to-end physical and simulator evaluations.
- *Synthetic Traces*: We generated 100 different jobs from 20 opensource GitHub [2] repositories covering diverse tasks and models in domains, such as CV, NLP, and audio. These repositories reflect varied development practices (e.g., code organization, logging, frameworks).

Workloads. In the physical experiments, we use the models, datasets, and batch sizes in Table 1. In the simulation experiments of standalone SCHEDMATE (§5.3), we use a sampled one-month trace from Acme without changing the workload, mainly LLM training jobs. In the Lucid (§5.3) and Sia(§5.3) simulation experiments, we follow the workload recipes of the original papers. To simulate real world development practices, we set a portion of the job to exit early.

5.2 Evaluation on a Physical Cluster

We evaluate Lucid and Lucid+SCHEDMATE (settings in §4.1) on a physical cluster comprising 8 nodes, each equipped with 8 NVIDIA A800-80GB GPUs (64 GPUs total). Our implementation extends the RAY [34] scheduler to incorporate Lucid and SCHEDMATE policies, utilizing RAY's fractional GPU capabilities for job packing. Each job is submitted as a RAY task. This approach allows deployment on existing clusters without altering the underlying resource manager.

The evaluation workload consists of 82 jobs sampled from the Acme trace [20]. We assign workloads from Table 1 based on the job's original GPU request size: *large* (>8 GPUs), *medium* (4-8 GPUs), and *small* (<4 GPUs). Job requests exceeding the cluster capacity are capped at 64 GPUs. The slowdown threshold is set to 0.5. We repeat 3 times on each approach to eliminate randomness. Lucid's profiler reserves one node, and profiles each job for 100s. SCHEDMATE operates as a RAY actor on a dedicated GPU, where we deploy Qwen-2.5-7B-Instruct and BGE-m3 as core models. Results in Table 2 show that Lucid+SCHEDMATE reduces the average JCT by 23.2% compared to Lucid on the physical cluster. This improvement comes from SCHEDMATE's ability to cancel subpar packing decisions and enhanced duration estimation. We also observe that the average latency of *Scheduling Advisor* is 9s, which is acceptable compared to the profiling overhead.

We also verify the fidelity of our simulator. We use the simulator in 5.3 to process the same traces and settings and compare the result with the ground truth result. The average error rate of both average JCT and makespan is less than 3.7%, which indicates the high fidelity of our simulator.

5.3 Simulation-based Evaluation

Case Study 1: Non-intrusive scheduling with Lucid. We conduct an end-to-end simulation on the performance of Lucid+SCHEDMATE, Lucid, and Horus, QSSF, on four traces: Acme, Philly, Saturn, and Venus. Figure 12(a,b,c) presents the CDF curves of JCTs on different traces. Lucid+SCHEDMATE consistently outperforms other policies. Figure 12(d) presents the average JCT improvements of Lucid+SCHEDMATE against Lucid. Lucid+SCHEDMATE demonstrates superior performance, improving $1.23\times \sim 1.91\times$ compared to Lucid alone. The key source of improvements is its ability to dynamically detect slowdowns of job packing and predict workload using *Workload Metadata*. Specifically, among the traces, the performance gain of SCHEDMATE is more significant on the Acme trace, owing to the fact that Acme trace contains heavier workloads, which leads to a significant slowdown in job packing. The interference-aware eviction policy of SCHEDMATE is more effective in such traces. Besides, Acme has a high-skewed workload distribution [20], where most of the jobs are short-term jobs, introducing challenges to duration estimation. In addition, the improvements in other traces are smaller because the duration distribution is more concentrated in Saturn and Philly, and the jobs are relatively lightweight. Many short jobs are finished during profiling. The interference of jobs is also rather lower, leaving less room for the improvement of the eviction policy.

Case Study 2: Elastic Scheduling with Sia. As mentioned in §4.2, Sia+SCHEDMATE can reduce the profiling overhead through *Preprofiling-free Bootstraping*. To test the effectiveness of SCHEDMATE in elastic scheduling scenar-

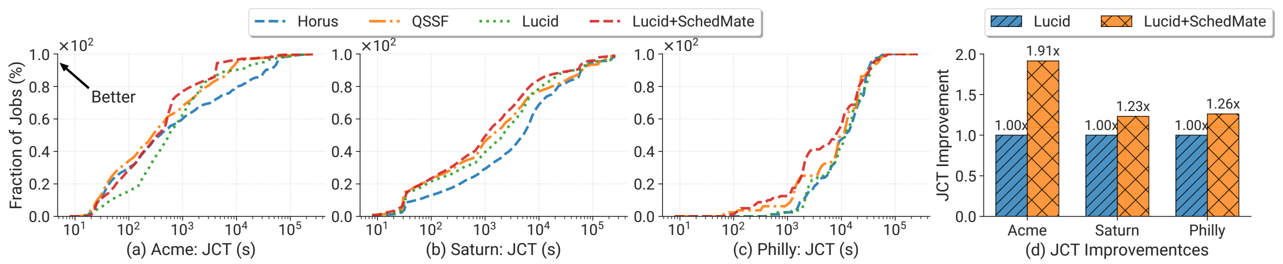


Figure 12: **Case Study 1.** The CDF curves of JCTs using different scheduling policies (a,b,c) and (d) Avg. JCT improvement of Lucid+SCHEDMATE against Lucid on four traces: Acme, Philly, Saturn, and Venus.

Table 3: **Case Study 2.** Comparison of the performance of Sia and Sia+SCHEDMATE across different traces.

Trace	Policy	Avg. JCT	p99 JCT	Makespan
Philly	Sia	0.74h	10.60h	14.7h
	Sia+SCHEDMATE	**0.59h**	**10.12h**	**14.4h**
Helios	Sia	0.83h	12.0h	15.7h
	Sia+SCHEDMATE	**0.72h**	**10.7h**	**14.8h**
Sia-trace	Sia	0.64h	4.25h	12.5h
	Sia+SCHEDMATE	**0.53h**	**4.12h**	**12.4h**

ios, we integrate SCHEDMATE into Sia's simulator and conduct evaluation using identical workload and heterogeneous cluster settings in Sia's paper [23]. Table 3 presents the results. We observe that our system consistently outperforms Sia. Specifically, SCHEDMATE reduces the average JCT by 13.3% $\sim$ 20%. Furthermore, the p99 JCT and makespans also show consistent improvements. The improvements derive from the benefits of *Scheduling Advisor* in mitigating profiling overhead while maintaining the accuracy of the scheduling decisions. In summary, this case study demonstrates that integrating SCHEDMATE with Sia (**Sia+SCHEDMATE**) significantly enhances scheduling efficiency.

Case Study 3: Standalone Scheduling. We evaluated the performance of SCHEDMATE as a standalone scheduler, which utilizes *Scheduling Advisor* for duration estimation and *Failure Handler* for automatic failure recovery. We sample jobs of Acme between June and August 2023. Specifically, to evaluate the failure recovery performance, we identify resumed failed jobs from the trace and consider failures in the simulator, which is the first simulator considering this factor. We compared standalone SCHEDMATE against several baseline schedulers: Tiresias [16] (a preemptive scheduler), Quasi-Shortest-Service-First (QSSF) [19] (ML to prioritize short jobs), FIFO, and Horus [54]. Figure 13 shows that standalone SCHEDMATE achieves a 25.7% improvement in average JCT over QSSF and 16% over Tiresias, primarily due to its superior duration estimation capabilities provided by the *Scheduling Advisor*. Furthermore, incorporating the *Failure Handler* yields an additional ~12.5% reduction in average JCT compared to SCHEDMATE without failure handling, highlighting its effectiveness in mitigating the impact of job failures. Overall, this case study demonstrates that standalone SCHEDMATE significantly enhances scheduling efficiency through LLM-

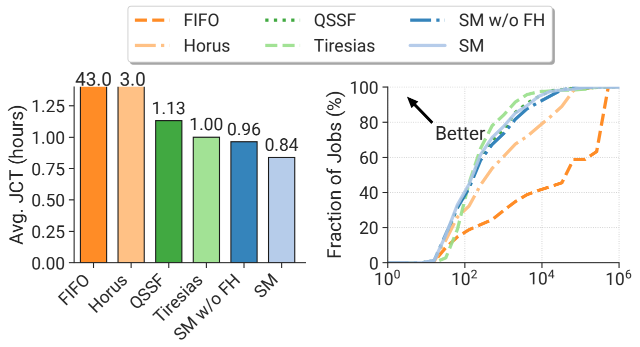


Figure 13: **Case Study 3: Standalone SCHEDMATE Evaluation.** (Left) Job Completion Times (JCTs) and queuing delays. (Right) CDF curves of JCTs using different policies. **SM**: SCHEDMATE. **FH**: *Failure Handler*.

based workload prediction and failure management.

5.4 Module-Specific Performance Analysis

Scheduling Advisor Evaluation. We compare the performance of workload estimation of *Scheduling Advisor* against an ML-based estimator from Lucid. We use the Mars trace to evaluate both methods. Specifically, the Lucid estimator trains an GA^2M model [21] for duration estimation. Figure 14 illustrates the results across SCHEDMATE, oracle SCHEDMATE, and Lucid. Due to the sensitivity of relative error when ground truth values are small, we truncate errors in (a) and (b). For duration estimation, SCHEDMATE achieves relative errors less than 100% for 85.7% of estimations, significantly outperforming Lucid at 27.7%. Notably, the curve shows a sharp increase as relative error approaches 100%. Regarding SM utilization estimation, SCHEDMATE performs slightly below Lucid's profiler, with average relative errors of 28.1% and 19.3% respectively. However, SCHEDMATE's method requires less than 10 seconds on a single GPU per job, whereas Lucid demands profiling per job for a period of time ($T_{prof} = 100s$ in this case). Furthermore, oracle SCHEDMATE presents a promising performance in both tasks, demonstrating the potential of our retrieval-based method.

Failure Handler Evaluation. We evaluate our failure handler on 300 failed jobs sampled from Mars, among which 75 are infrastructure failures. We set *failure handler w/o locator* as the baseline, which simply parses the last 500 log lines

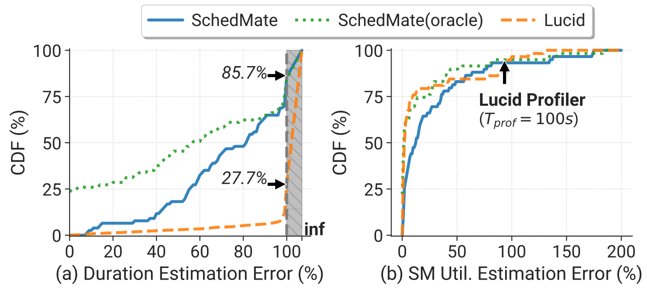


Figure 14: CDF curves of relative error for (a) duration estimation and (b) SM utilization estimation of SCHEDMATE, oracle SCHEDMATE, and Lucid.

Table 4: **Failure Handler Evaluation**. We evaluate the component in identifying infrastructure failures against baselines. Failure Handler utilizes Qwen-2.5 models.

Method	Model	<u>F1-score</u>	Precision	Accuracy
Failure Parser	7B	**67.7**	**75.7**	**90.1**
	14B	**65.1**	**77.1**	**90.0**
	32B	<u>68.2</u>	<u>78.4</u>	<u>90.7</u>
Failure Parser w/o Locator	7B	11.1	75.0	83.8
	14B	62.5	47.9	81.8
	32B	**69.0**	**81.1**	**90.9**
RCACopilot [7]	FastText	43.0	50.0	87.0

without attempting to locate the failure message, relying on the assumption that the relevant information is likely to be near the end of the log. For the *failure handler*, we parse the <u>200</u> log lines around the located failure message. Results are summarized in Table 4. We observe that the *Failure Handler* method consistently outperforms or matches the baseline across different model sizes, suggesting reliable identification of infrastructure failures with minimal false positives. While the baseline only shows competitive performance with the 32B model, our method achieves comparable results with smaller models, indicating better efficiency. Specifically, the 7B model achieves a precision of 95% while requiring 40% less computational resources compared to the 32B model. This demonstrates that the failure locator effectively narrows down the relevant log lines, enabling smaller models to perform well without sacrificing accuracy, thereby reducing both latency and resource consumption.

We also compare our system against RCACopilot [7], a recent embedding-based approach. It achieves only 43.0 F1-score. We believe this is due to the complex failure patterns, which are not well captured by the embedding model.

Metric Tracker Evaluation. The *Metric Tracker* resolves the noisy log challenges in real-world scenarios. We tested it on a synthetic dataset of 1,000 logs, with 50% containing massive irrelevant data. We set pure LLM method as the baseline, which processes the log without filtering. We perform *step time extraction* using both methods. As shown in Figure 15, *Metric Tracker* achieved an 84.3% success rate and an RMSRE of 0.42, significantly outperforming the pure LLM approach, whose success rate drops to 59.2%. These results highlight the *Metric Tracker*'s superior resilience to noisy log

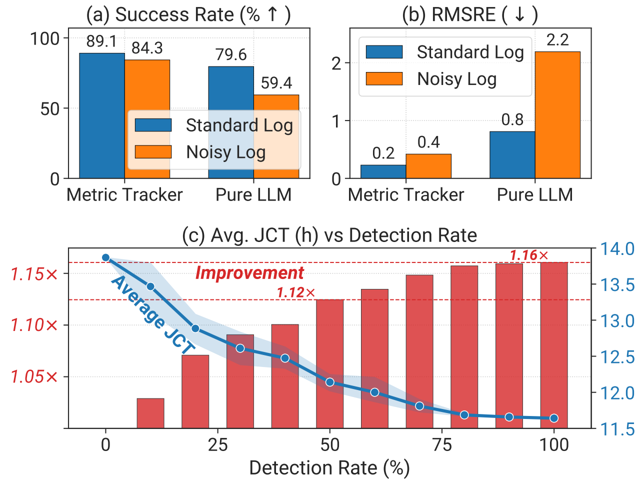


Figure 15: **Metric Tracker Evaluation.** Performance comparison between *Metric Tracker* and a pure LLM under standard and noisy conditions. (a) Success rate evaluation. (b) Rmsre comparision. (c) Impact of slowdown detection rate on average JCT (red) and its improvement (blue) in Lucid+SCHEDMATE (each repeated five times).

data. Besides, we also observe that our method has a lower latency with **8.7**s per log compared to 9.8s per log in pure LLM. These results demonstrate both the efficiency of the *Metric Tracker*, and robustness in handling noisy data, which is common in real-world DL job logs.

5.5 Micro-benchmarks

Ablation Study on Partial Information. The non-intrusiveness of SCHEDMATE indicates the absence of certain information. We perform an ablation study on the Lucid+SCHEDMATE's packing eviction technique. In practice, we can only detect the slowdown of jobs when they actually output the step times in their logs. To simulate different levels of partial information, we manually control the **detection rate** from 0 to 1.0. The detection rate refers to the percentage of detected slowdown jobs ratio. As shown in , the JCTs of Lucid on a sampled **Acme** trace with interference-aware evicting under different slowdown detection rates are presented. We assign the workloads in Table 1 and set the slowdown threshold to 0.50. The JCTs under different settings are presented in Figure 15(c). We observe that the average JCT decreases as the detection rate increases. The improvement achieves 12.5% when the detection rate is 0.5. When the detection rate is above 0.6, the performance gain is limited. *Metric Tracker* relies on the assumption that most jobs would output the metrics in the log. In addition, the result in Table 15 shows *Metric Tracker* achieves a 91.6% success rate using a Qwen-2.5-7B model. In summary, relying on users' logs to detect the slowdown of jobs seems to be a risky approach. However, we prove that in real-world scenarios, partial information can achieve considerable performance gain.

Impact of Different LLMs. Figure 17 presents the performance of various LLMs across different scales for the three

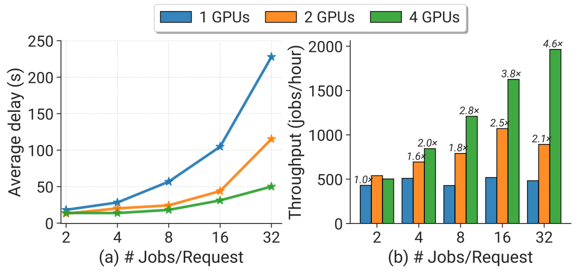


Figure 16: Performance scaling with job intensity and GPU count of the *Scheduling Advisor* using a Qwen-1.5-14B-Int8 model (a) Average delay vs. jobs per request. (b) Throughput and improvement ratios for different numbers of GPUs.

modules in SCHEDMATE. We observe that model performance improves with increased model size, with Qwen2.5-32B achieving the best overall results among open-source models. Smaller models (0.5B and 1.5B) failed to perform the SA task, indicating that a certain level of model complexity is required for effective metadata extraction. Interestingly, the closed-source GPT-4o model demonstrates superior performance in the SA task but unexpectedly underperforms in MT and FH tasks compared to larger Qwen models. In addition, despite competitive performance in MT tasks, LLaMA-3.1-8B struggles significantly in the FH task and does not match the same scale Qwen models in the other two tasks. The results highlight a trade-off between model size, task performance, and computational requirements, with models in the 7B to 14B parameter range offering a good balance of effectiveness and efficiency for SCHEDMATE's tasks.

Scalability and Overhead We sample 200 jobs from *Mars* and test the *Scheduling Advisor*, which takes the heaviest load among the three modules, under varying GPU counts and request intensity. We deploy Qwen-1.5-14B-Int8 model on each GPU. Note that for testing scalability on heavier workload, we only sample large-scale jobs that use the same project with over **40K LoC**.

Figure 16 (a) shows that when the request intensity is less than 4, the Avg. delay remains less than 20s, acceptable for one GPU and no significant benefit for more. When request intensity ≥ 8, delay of one GPU increases significantly, while using more GPUs maintains lower delays. Figure 16 (b) illustrates the throughput improvements. We set the *1GPU2job throughput* as the baseline and mark the throughput improvement. It's obvious that single GPU throughput remains steady ($\sim$480 jobs/hour), while the 2-GPU and 4-GPU settings increase. Specifically, the 2-GPU setting achieves over 2$\times$ throughput when request intensity reaches 16, and 4-GPU up to 4.6$\times$ ($\sim$2000 jobs/hour) throughput over the baseline setting when 32 jobs per request. In summary, we also recommend large-scale cluster operators to dynamically assign GPUs according to the actual submission rate. For instance, for PAI [48] with periodical load (300 $\sim$ 1200 jobs/h), while single GPU is suitable for low load periods, to manage peak load periods, one should allocate 4 or more GPUs.

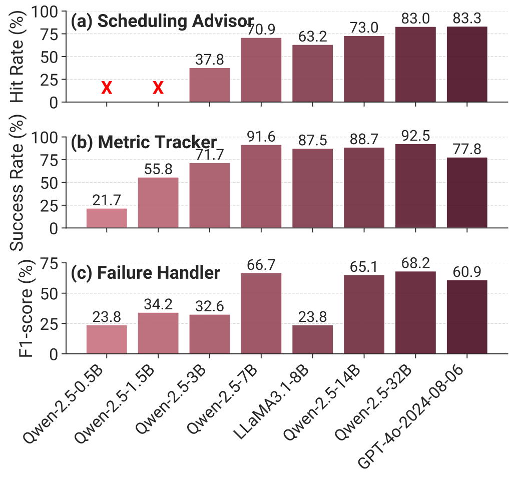


Figure 17: **Evaluation on different LLMs.** We involve 7 open-source models and one closed-source model, GPT-4o. The red cross mark means that the model fails in the task.

6 Discussion and Related Work

Supporting Other Workloads. While SCHEDMATE targets DL training, its core idea and techniques, especially the *Scheduling Advisor*, are adaptable to other domains like big data analytics and serverless computing, provided jobs' source code is accessible. Adapting SCHEDMATE can potentially help address domain-specific challenges, such as data locality for big data or dynamic scaling for serverless functions. Future work can explore these adaptations and extend our approach to emerging paradigms like edge computing, broadening SCHEDMATE's impact.

Related Work. Learning-based approaches, including reinforcement learning and LLMs, are increasingly employed to optimize system performance across various domains [18, 21, 26, 49]. LLMs have been applied to specific system tasks like log parsing [28, 32], failure diagnosis [7, 30, 47], and network management [49]; however, existing research has not explored their potential for deep semantic understanding of Deep learning job characteristics derived from both source code and runtime logs to directly improve scheduling decisions. Some prior work addresses related areas but differs from SCHEDMATE's focus on enhancing DL scheduling performance. Systems like 3Sigma [36] utilize metadata for similar job retrieval, lacking semantic workload analysis. ML-based failure handling tools like RCACopilot [7] lack scheduling specific capabilities and show lower performance compared then LLMs-based *Failure Handler*. SCHEDMATE distinguishes itself by leveraging LLM analysis to extract DL semantics, monitoring metrics, and automating failure handling, thus enhancing DL scheduler performance.

7 Conclusion

SCHEDMATE leverages LLMs to enhance DL job scheduling by extracting insights from source code and logs. It addresses critical challenges in existing schedulers, significantly improving scheduling performance across various scenarios.

References

[1] GitHub - 1rgs/jsonformer: A Bulletproof Way to Generate Structured JSON from Language Models — github.com. https://github.com/1rgs/jsonformer. [Accessed 24-08-2024].

[2] Github. https://github.com, 2023.

[3] Gpt-4v(ision) system card. 2023.

[4] Nvidia management library (nvml). https://developer.nvidia.com/management-library-nvml, 2024. Accessed: 2024-10-16.

[5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.

[6] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation, June 2024. arXiv:2402.03216 [cs].

[7] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the 19th EuroSys Conference*, EuroSys '24. Association for Computing Machinery, 2024.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '19, 2019.

[10] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[11] Abdullah Bin Faisal, Noah Martin, Hafiz Mohsin Bashir, Swaminathan Lamelas, and Fahad R. Dogar. When will my ML job finish? toward providing completion time estimates through Predictability-Centric scheduling. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 487–505, Santa Clara, CA, July 2024. USENIX Association.

[12] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[13] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Locality-Enhanced Serverless Inference for Large Language Models, January 2024.

[14] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *CoRR*, abs/2205.11913, 2022.

[15] Wei Gao, Xu Zhang, Shan Huang, Shangwei Guo, Peng Sun, Yonggang Wen, and Tianwei Zhang. Autosched: An adaptive self-configured framework for scheduling deep learning training workloads. In *Proceedings of the 38th ACM International Conference on Supercomputing*, ICS '24, page 473–484, New York, NY, USA, 2024. Association for Computing Machinery.

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

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, 2016.

[18] Qinghao Hu, Harsha Nori, Peng Sun, Yonggang Wen, and Tianwei Zhang. Primo: Practical learning-augmented systems with interpretable models. In *2022 USENIX Annual Technical Conference*, USENIX ATC '22. USENIX Association, 2022.

[19] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[20] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei

Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, 2024.

[21] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23. Association for Computing Machinery, 2023.

[22] Akshay Jajoo, Y. Charlie Hu, Xiaojun Lin, and Nan Deng. A case for task sampling based learning for cluster job scheduling. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 19–33. USENIX Association, 2022.

[23] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23. Association for Computing Machinery, 2023.

[24] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 947–960. USENIX Association, 2019.

[25] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[26] Taeyoon Kim, Suyeon Jeong, Jongseop Lee, Soobee Lee, and Myeongjae Jeon. Sibylla: To retry or not to retry on deep learning job failure. In *2022 USENIX Annual Technical Conference*, USENIX ATC '22, pages 263–270. USENIX Association, 2022.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23. Association for Computing Machinery, 2023.

[28] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23. IEEE Press, 2023.

[29] Shijian Li, Robert J. Walls, and Tian Guo. Characterizing and Modeling Distributed Training with Transient Cloud GPU Servers. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 943–953, Singapore, Singapore, November 2020. IEEE.

[30] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyin Xu. Configuration validation with large language models. *CoRR*, abs/2310.09690, 2023.

[31] LlamaTeam. The llama 3 herd of models, 2024.

[32] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

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

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 561–577. USENIX Association, 2018.

[35] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 481–498. USENIX Association, 2020.

[36] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery, 2018.

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

Article 3 in EuroSys '18, pages 1–14, New York, NY, USA, April 2018. Association for Computing Machinery.

[38] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[39] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016.

[40] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

[41] Salvatore Sanfilippo. Redis: An in-memory database. https://redis.io, 2009.

[42] Amber L Simpson, Michela Antonelli, Spyridon Bakas, Michel Bilello, Keyvan Farahani, Bram Van Ginneken, Annette Kopp-Schneider, Bennett A Landman, Geert Litjens, Bjoern Menze, et al. A large annotated medical image dataset for the development and evaluation of segmentation algorithms. *arXiv preprint arXiv:1902.09063*, 2019.

[43] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *ICML '19*, pages 6105–6114, 2019.

[44] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[45] Qwen Team. Introducing qwen1.5, February 2024.

[46] Qwen Team. Qwen2.5: A party of foundation models, September 2024.

[47] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Lunting Fan, Lingfei Wu, and Qingsong Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. *CoRR*, abs/2310.16340, 2023.

[48] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 945–960. USENIX Association, 2022.

[49] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, volume 33 of *ACM SIGCOMM '24*, page 661–678. ACM, August 2024.

[50] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 595–610. USENIX Association, 2018.

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

[52] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report, 2024.

[53] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[54] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, and Peter Garraghan. Horus: Interference-aware and prediction-based scheduling

in deep learning systems. *IEEE Transactions on Parallel and Distributed Systems*, 33:88–100, 2022.

[55] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Berlin Heidelberg, 2003.

[56] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022.