PromptTuner: SLO-Aware Elastic System for LLM Prompt Tuning

Anonymous Author(s)

ABSTRACT

Prompt tuning has become a prominent strategy for enhancing the performance of Large Language Models (LLMs) on downstream tasks. Many IT enterprises now offer Prompt-Tuning-as-a-Service to fulfill the growing demand for prompt tuning LLMs on downstream tasks. Their primary objective is to satisfy users' Service Level Objectives (SLOs) while reducing resource provisioning costs. Nevertheless, our characterization analysis for existing deep learning resource management systems reveals that they are insufficient to optimize these objectives for LLM prompt tuning workloads.

In this paper, we introduce PromptTuner, an SLO-aware elastic system to optimize LLM prompt tuning. It contains two innovations. (1) We design a *Prompt Bank* to identify efficient initial prompts to expedite the convergence of prompt tuning. (2) We develop a *Workload Scheduler* to enable fast resource allocation to reduce the SLO violation and resource costs. In our evaluation, PromptTuner reduces SLO violations by 4.0× and 7.9×, and lowers costs by 1.6× and 4.5×, compared to INFless and ElasticFlow respectively.

1 INTRODUCTION

Large Language Models (LLMs) are becoming prevalent in many scenarios owing to their exceptional capabilities [16, 34, 60]. LLM developers employ a compelling and widely embraced technique known as *prompt tuning*, to customize LLMs for diverse applications, particularly agentic ones, without altering the model weights [7, 19, 54, 59]. However, the manual process of prompt tuning is time-consuming and resource-intensive [26, 75, 86], driving many IT companies to offer Prompt-Tuning-as-a-Service to enable automatic prompt tuning within seconds to minutes [8, 9, 15]. In this business model, users furnish initial prompts and downstream datasets and select the base LLMs. Subsequently, the service provider must efficiently allocate GPUs to optimize the prompts for the given datasets, handling tens of thousands of LLM prompt tuning (LPT) requests per day [15], returning the finalized prompts to users.

The service provider has several considerations when serving users' LPT requests. First, users concentrate on the accuracy[1] and the latency of their LPT requests. They will specify the Service Level Objectives[2] (SLOs) of the targeted accuracy and latency [6, 8, 9, 15]. Second, the service provider rents top-grade GPU resources from clouds [3–5] to handle users' LPT requests. Given the increasing number of LPT requests and the considerable cost of renting GPUs, there is a pressing need to design systems that optimize resource allocations for LPT workloads. Such optimization aims to reduce resource costs for service providers while fulfilling SLOs for users.

We present a workload characterization summary of LPT workloads in §2.2, and find that they exhibit training-like and inference-like features. A straightforward approach is to leverage prior studies in cluster management systems for training and inference workloads to address LPT demands. However, our empirical study in §3 shows that they are ineffective in managing LPT workloads. First, previous SLO-aware systems for Deep Learning (DL) training [22, 31, 35] oversubscribe a **fixed**-sized GPU cluster to guarantee SLOs, resulting in increased resource costs. Also, the commonly adopted frequent resource allocation could incur nearly one-minute resource allocation overhead for LLMs [41, 69] and pose a significant barrier to enforcing minutes-level latency SLOs for LPT workloads. Second, prior inference systems [64, 77, 82, 83] adopt two techniques: (1) they autoscale the quantity of GPUs needed to reduce resource costs; (2) they pre-load the DL runtime (e.g., CUDA/framework runtime) and model weights in the GPU memory for a time period to reduce the allocation overhead and optimize the SLO attainment. However, these solutions adhere to a **fixed** GPU allocation, normally assigning one GPU for each job, compromising the adaptability required to meet varying levels of SLOs for LPT jobs. As shown in §3.2, even with the incorporation of multi-GPU allocation into DL inference systems, they still struggle to serve LPT jobs effectively. Overall, prior training and inference systems exhibit deficiencies in realizing SLO satisfaction and cost reduction simultaneously for LPT.

Additionally, a unique feature of LPT workloads is overlooked by existing DL cluster management systems and LPT services: their model convergence is highly *sensitive to the initial prompts* (§2.2). This sensitivity suggests the significant variance in the number of iterations required to achieve the targeted accuracy given different initial prompts. For example, a well-curated initial prompt demands fewer tuning iterations than a poor one, thereby mitigating SLO violations and reducing resource costs. Practically, LLM developers adopt two initialization methods. First, the current practice of LPT services is manual initialization. Users are asked to craft initial prompts by themselves [17, 79]. Alternatively, users are recommended to reuse publicly available prompts directly [1, 8]. However, both practices rely on human expertise, substantial GPU resources, and time for these laborious trial-and-error processes. Second, some LLM studies [80, 86] and LLM services [2] propose induction initialization to guide LLMs to automatically generate initial prompts without human expertise. However, the quality of the generated initial prompt heavily relies on the performance of the LLM itself [80, 86] (evaluated in §6.3). Despite the potential benefits, few systematic efforts exist to automatically and efficiently identify initial prompts for a given LPT job.

To bridge these gaps, we design PromptTuner, an SLO-aware elastic cluster management system dedicated to LPT. PromptTuner consists of two designs. First, we design a *Prompt Bank* as a query engine to automatically and efficiently search the initial prompt for a given LPT job. The observation that prompts optimized for one LPT task can serve as effective initial prompts for another task with high similarity motivates the design of the Prompt Bank [67, 72]. As public prompts optimized for various tasks are noticeably increasing [27], we collect thousands of high-quality prompts as the initial prompt candidates for incoming LPT jobs. We adopt a two-layer data structure that enables quick search of an effective initial prompt, reducing the selection time to under 10 seconds.

[1] We use accuracy as a universal term to denote any evaluation metric.
[2] The definition of SLO is explained in §4.2.

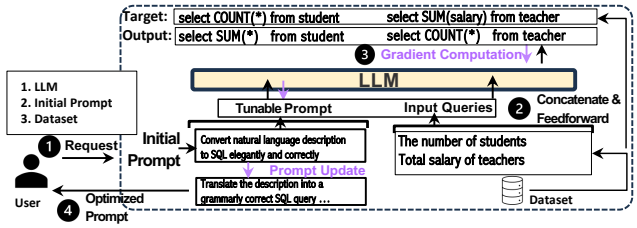


Figure 1: An example of LLM prompt tuning. The user first prepares the LLM, the initial prompt, and the task-specific dataset, which consists of a batch of input queries and target responses. During the execution stage, it optimizes the tunable prompt starting from the initial prompt on the given dataset.

Second, we design a *Workload Scheduler* that supports fast and elastic GPU allocation for LPT workloads to meet SLOs and reduce resource costs. The Workload Scheduler allows LPT jobs based on the same LLM to reuse the GPUs from a warm GPU pool comprising GPUs with the same job-specific pre-loaded LLM runtime and weights, providing rapid GPU allocation. The Workload Scheduler maintains a dedicated warm GPU pool for each LLM and dynamically adjusts the pool size by adding GPUs from a shared cold GPU pool. It consists of two algorithms to manage these GPU pools. The first delivers fast multi-GPU allocation from the warm GPU pools to LPT jobs, reducing the considerable initialization overhead for multi-GPU execution (§3.2). The second one dynamically adjusts the number of GPUs for each warm GPU pool to balance the trade-off between SLO attainment and resource costs in the dynamic traffic of LPT jobs. Furthermore, the Workload Scheduler intelligently decides whether to route incoming LPT requests to the Prompt Bank or execute them directly to prevent resource contention.

We implement `PromptTuner` atop Knative, and select three LLMs (GPT2-Base [18], GPT2-Large [18], Vicuna-7B [23]) to compare `PromptTuner` with the state-of-the-art SLO-aware DL inference system INFless [77] and training system ElasticFlow [35] on a cluster of 32 NVIDIA A100-80GB GPUs. `PromptTuner` reduces the SLO violation by up to 4.0× (INFless) and 7.9× (ElasticFlow), and reduces the cost by up to 1.6× (INFless) and 4.5× (ElasticFlow). We also conduct experiments on LLaMA-30B and DeepSeek-R1-Distill-Qwen-7B (Qwen7B-R1) and perform large-scale experiments in a 96-GPU cluster to demonstrate its superiority under heavy workload settings. Our contributions are as follows:

- ★ We perform an in-depth characterization analysis for LPT workloads and conduct detailed empirical studies to uncover the inefficiencies of existing systems to handle LPT.
- ★ We present `PromptTuner`, an elastic system for LPT workloads that can guarantee SLOs for users and reduce resource costs for service providers.
- ★ We perform extensive evaluations to validate the efficiency of the *Prompt Bank* and the *Workload Scheduler*.

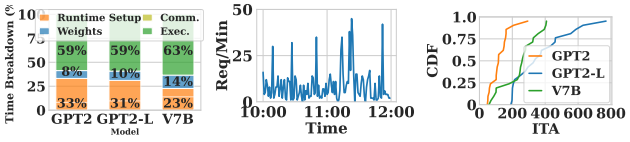
2 LPT WORKLOAD CHARACTERIZATION

2.1 Prompt Tuning

Prompt tuning is an approach to obtain high-quality responses for a specific task from an LLM by attaching a prompt prefix (simply referred to as prompt), saving the high cost of retraining the model weights. An LPT job optimizes a prompt that elicits the best response from the LLM when prepended to an input query. Figure 1 shows an example of the task of converting the natural language to SQL language using the gradient-based LPT algorithm [50]. The user sends an LPT request containing the LLM, the initial prompt ("*Convert natural language description to SQL elegantly and correctly*"), and the task-specific dataset consisting of input queries and corresponding target responses. Some LPT service providers [1, 8] recommend the user to specify their initial prompt based on their expertise (❶). To execute an LPT job, the LPT system feeds this set of input queries into the LLM (❷). The system runs the given LPT algorithm to compute the loss between the generated output sentences and targeted responses. Then it backpropagates the textual gradients and updates them on the tunable prompt (❸). After multiple iterations, the optimized prompt is generated: "*Translate the description into a grammatically correct SQL query optimized for speed and accuracy*", and returned to the user (❹).

Table 1: The average score (↑) of prompting techniques over tasks.

Prompting Techniques	GPT-3.5	GPT-4	Vicuna-7B	LLaMA-30B	Qwen7B-R1
Few Shot	29.8	37.0	14.9	20.8	34.1
Prompt Tuning	70.1	75.8	80.5	84.1	85.7



(a) Time Breakdown. (b) Trace Pattern. (c) ITA CDF.

Figure 2: Characteristics of LPT workloads: (a) The end-to-end LPT job execution time breakdown across different LLMs. (b) A 2-hour LPT workload trace from a cluster. (c) The Iteration-To-Accuracy (ITA) distribution of various initial prompts with the SAMSUM dataset [33] across different LLMs.

Prevalence of LPT Workloads. Today, LPT workloads emerge as an important GPU consumer, making prompt-tuning services an essential business practice [8]. When a user sends an LPT request, the system registers it as an LPT job and schedules each LPT job to run on GPUs while maintaining the strict SLOs the users impose.

The prevalence of LPT workloads manifests in three aspects. First, many prompt-tuning services [1, 8, 9, 15, 29, 61, 68] serve to expand LLMs across various fields, making the LLM prompt market trendy and growing. LLM developers daily produce tens of thousands of prompt-tuning requests [8, 9, 15] and claim a significant number of high-grade GPUs [10, 11] to respond to these LPT requests quickly. Second, LLM developers utilize curated prompts to guide commercial LLM services in surpassing human-engineered prompts on downstream tasks [86]. As shown in the second and third columns of Table 1, prompt tuning surpasses few-shot prompting techniques across 10 LLM tasks [36], delivering an average improvement of 2.5× and 1.8× with GPT-3.5 and GPT-4, respectively. Third, LLM developers rely on prompt tuning methods to enhance the accuracy of open-source LLMs on specific tasks [14, 21, 73]. In the forth, fifth and sixth columns of Table 1, prompt tuning achieves an average score improvement of 5.4×, 4.0×, 2.2× across various tasks on Vicuna-7B, LLaMA-30B, and Qwen7B-R1, respectively. Open-source LLMs provide access the output at the logits layer, leading to higher accuracy compared to commercial LLM services.

2.2 Prompt Tuning Workload Characteristics

Next, we study the LPT workload characteristics, which can guide the design of an efficient LPT cluster management system. We experiment with three popular LLMs (GPT2-Base, GPT2-Large, Vicuna-7B) and the SAMSUM dataset [33] on a server of 8 Nvidia A100-80GB GPUs. We identify some common characteristics that LPT workloads share with training and inference workloads, which have been extensively studied by prior works [38, 42, 64, 65, 76, 82].

Synchronous Cross-GPU Communication. Similar to DL training workloads, executing an LPT job requires iterations of feed-forward/backward passes, followed by a synchronous exchange of prompt gradients after each iteration. However, the cross-GPU communication of LPT is much lower than that of DL training. Figure 2a shows the time breakdown of three LPT workloads: the communication overheads are within 0.4-0.5% of the total execution time. Hence, LPT workloads can enjoy a nearly linear throughput increase when the number of allocated GPUs is increased.

Dynamic Traffic. LPT is a user-facing service, featuring highly volatile dynamic traffic. We analyze a trace of LPT jobs sampled from a 64-GPU cluster in a large institute (anonymized). Figure 2b presents the LPT job arrival time for prompt-tuning Vicuna-7B within two hours. We observe large spikes of LPT traffic, with the maximum number of requests per minute being 5× the mean. Such a pattern indicates that an efficient LPT system needs highly reactive autoscaling.

High GPU Allocation-to-Computation Ratio. The dynamic nature of LPT workloads requires fast provisioning of GPUs, similar to inference workloads [64, 65, 82]. We measure the GPU allocation overhead (including container setup, framework initialization, and GPU runtime creation), which accounts for 37-41% of the total execution time. This indicates the need for fast GPU provisioning and reuse across LPT jobs.

High Sensitivity to Initial Prompts. We observe that the convergence speed of the LPT workload highly depends on the choice of the initial prompt. We measure the convergence speed with the Iterations-To-Accuracy (ITA) metric using 20 randomly selected prompts on the SAMSUM dataset [33] for different LLMs. Figure 2c shows the cumulative distribution function (CDF) of the ITA metric. The median and maximum ITA values are 1.7-4.5× higher than the minimum ITA, indicating the significance of selecting an effective initial prompt at the beginning of LPT. Given the availability of substantial public prompts [27], we identify the possibility of finding and reusing them as initial prompts for specific tasks.

Characterization Summary. Table 2 summarizes the characteristics of LPT, DL training, and inference workloads. First, LPT workloads require synchronous communication after each iteration, similar to DL training. Second, LPT workloads are highly dynamic and suffer from lengthy GPU allocation delays, similar to DL inference workloads. Meanwhile, LPT workloads have a unique feature: their processing time highly depends on the choice of the initial prompts.

3 CHARACTERIZATION OF THE EXISTING CLUSTER MANAGEMENT SYSTEMS

As LPT workloads share similar execution features with DL training and inference workloads, a natural strategy is to extend existing cluster management systems for training and inference to LPT. In this section, we quantitatively evaluate the efficiency of state-of-the-art inference and training systems using the same experimental setup as in §6.1. We use the first 20 minutes of the trace in Figure 2b to run the prompt-tuning jobs based on the Vicuna-7B model.

Table 2: Comparison of LPT, DL inference and training workloads.

Characteristics	LPT	Inference	Training
Synchronous Cross-GPU Comm.	✓	✗	✓
Dynamic Traffic	✓	✓	✗
High Allocation Overhead	✓	✓	✗
Prompt Sensitivity	✓	✗	✗

3.1 Inefficiency of DL Training Systems

Prior works have proposed many system designs [22, 31, 32, 35, 78] that optimize the execution of DL training workloads. These systems provision a fixed-size GPU cluster, further referred to as a GPU pool, and frequently allocate GPUs from this pool to jobs to maximize GPU utilization.

We evaluate the efficiency of the state-of-the-art SLO-aware training system ElasticFlow [35]. It dynamically adjusts the number of allocated GPUs for each job to improve the job throughput and SLO attainment. However, in ElasticFlow, the resource costs per time unit remain fixed for all statically provisioned GPUs, regardless of actual usage. Figure 3a shows the GPU cluster utilization of ElasticFlow. On average, ElasticFlow only achieves 56% GPU cluster utilization, almost doubling the GPU cluster's cost.

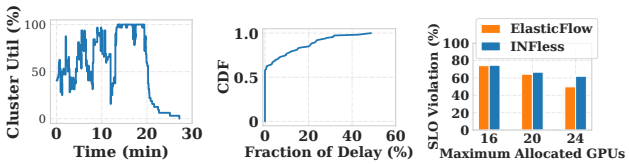
> **Inefficiency 1**: The static provisioning of a fixed-size GPU cluster in existing DL training systems results in a high resource cost when running LPT workloads.

3.2 Inefficiency of DL Inference Systems

Existing inference systems [64, 77, 82, 83] often feature a serverless autoscaling architecture. Upon receiving an inference job, they typically allocate a GPU-equipped container, also called an instance that is a unit of scaling, from a large pool of available GPUs to the provider for each incoming job. To alleviate the lengthy GPU container startup overheads, providers keep idle instances ready to serve any future inference jobs of the same model, occupying pricey GPU memory for a prolonged time. These systems implement autoscaling to adjust the number of instances for each model according to changes in the inference traffic.

Although these designs avoid the static resource provisioning of the training systems, their performance suffers from other limitations. First, they scale the resources of each model independently without considering a globally optimal schedule. Second, prior systems [64, 77, 82, 83] are limited to allocating one GPU for each instance of a model. Last, they lack support for synchronous cross-GPU communication, which is required for LPT jobs.

We select INFless [77], a representative DL inference system, for our evaluation. However, running an LPT job on a single instance is insufficient to improve the job throughput and meet the emergent latency SLOs. To address this limitation, we extend INFless to support synchronous cross-GPU communication via Memcached [12], as commonly used in serverless systems [44, 74]. The implementation details of multi-GPU execution can be found in §5.1. Thus, a



(a) ElasticFlow. (b) INFless. (c) SLO violations.

Figure 3: Characterization of existing DL systems: (a) The cluster utilization (%) (y-axis) in ElasticFlow over time (x-axis). (b) The CDF (y-axis) illustrates the fraction (x-axis) of waiting delay in the end-to-end latency caused by the instance initialization.(c) SLO violation (%) of ElasticFlow and INFless across varying maximum GPUs.

single LPT job can use multiple instances, i.e., multiple GPUs, to accelerate its completion. Nonetheless, in INFless and other inference systems, some instances may need tens of seconds to initialize, thereby incurring long waiting time for the LPT job when running across multiple instances. Figure 3b depicts that instance initialization contributes on average 11% to the end-to-end LPT job latency, and up to 50% in the worst case.

> **Inefficiency 2**: The initialization techniques for a single instance adopted by DL inference systems incur substantial delays for the initialization of multi-GPU instances.

Unsurprisingly, ElasticFlow and INFless show substantially high SLO violation rates due to the abovementioned inefficiencies. Figure 3c shows the SLO violation (%) – up to 70% – occurring when executing the LPT workload on top of ElasticFlow and INFless with varying maximum numbers of allocated GPUs. These results demonstrate that existing cluster management systems are unsuitable for LPT workloads, calling for designing a new system tailored to the LPT workload characteristics in §2.2.

4 SYSTEM DESIGN

We introduce `PromptTuner`, an SLO-aware elastic cluster management system for LPT workloads. We begin with the design insights and overview, followed by the illustration of its two key components: Prompt Bank and Workload Scheduler.

4.1 Design Insights

The design of `PromptTuner` is motivated by two insights. Our first insight is that *LPT tasks can reuse the prompts optimized for similar tasks as their initial prompt to reduce the number of tuning iterations needed to achieve the desired accuracy.* Extensive empirical studies on transfer learning [67, 72] and theoretical analysis [71] affirm that reusing prompts can considerably accelerate the model convergence. However, the key to successfully reusing prompts lies in automatically and promptly identifying the most effective ones.

Our second insight is that *LPT workloads can reuse the runtime of the jobs based on the same LLMs.* Indeed, many LPT jobs load the same runtime state into the GPUs before execution [20, 66]. This state includes the CUDA and DL framework dependencies and model weights. Reusing this state can substantially reduce the GPU allocation overhead (§2.2). However, the key to effectively reusing the runtime lies in mitigating the substantial delays for LPT jobs demanding multiple GPUs, as emphasized in §3.2.

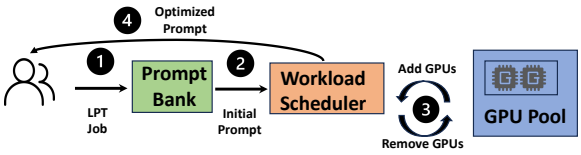


Figure 4: The workflow of `PromptTuner`. It consists of two key components: (1) The Prompt Bank identifies an effective initial prompt for an incoming LPT job at a minimal cost; (2) The Workload Scheduler dynamically adds GPUs from the GPU pool for each LPT job to reduce SLO violation while minimizing resource costs.

Table 3: Job attributes description in `PromptTuner`.

Attributes	Description
`Model`	The LLM model name.
`Termination Condition`	The job completion criteria, including a maximum number of iterations and an accuracy target.
`Deadline`	The anticipated time by which the LPT workload should be completed.
`Dataset`	A path (e.g., AWS S3) where data samples are stored.
`Hyperparam`	Including initial prompt and parameters such as batch size, optimization algorithm.
`Prompt`	The optimized prompt.

4.2 System Overview

`PromptTuner` contains two key components: the Prompt Bank leverages *prompt reusing* to identify effective initial prompts for incoming LPT jobs (§4.3); the Workload Scheduler harnesses *runtime reusing* to rapidly allocate GPUs to each LPT job, maintaining the SLO requirement while reducing resource costs (§4.4).

Figure 4 shows the workflow of `PromptTuner`. First, the user submits an LPT job to the service provider (❶). The Prompt Bank identifies the effective initial prompt for this job (❷). Next, the Workload Scheduler dynamically adds/removes GPUs from/to the GPU pool based on the GPU demand of incoming traffic. The Workload Scheduler also dynamically adjusts the amount of GPUs for each job periodically (❸). Finally, the LPT service provider returns the optimized prompt to the user upon the LPT job completion (❹).

A job in `PromptTuner` is equivalent to an RPC request sent by an LPT service user followed by the RPC response from the system. Table 3 summarizes the job attributes and descriptions. The first five attributes are job parameters specified by users. The last parameter is the response with an optimized prompt that the system returns to the user. The SLO of a job is defined as the maximum time during which the LPT job meets the termination condition.

4.3 Prompt Bank

The Prompt Bank realizes *prompt reusing* to improve the ITA performance of incoming LPT jobs. It contains a set of prompts shared by all LPT jobs for selection as their initial prompts. We aim to *balance* the speedup benefits of identifying initial prompts and latency cost of the query. To this end, we engineer the Prompt Bank as a query engine with a two-layer data structure. It enables efficient lookup operations for new LPT jobs and facilitates the seamless insertion and replacement of new initial prompt candidates. Below we detail the process of constructing the data structure and performing lookup, insertion, and replacement operations on it. The notations used in this section are defined in Table 4.

4.3.1 Data Structure Construction. We first assemble thousands of prompt candidates from public sources [8, 27] into a comprehensive set, which can maximize the likelihood of selecting effective initial

Table 4: Summary of Notations in the Prompt Bank.

Sym.	Definition
$\mathcal{D}_{\text{eval}}$	The evaluation dataset
d_i^{in}	The input query sample
d_i^{tgt}	The target response sample
concat	The operation to concatenate two text sequences
$\mathcal{L}$	The loss between the output and target sample
C	The total number of prompt candidates in the Prompt Bank
K	The number of clusters for algorithm K-medoid
C_{sim}	The cluster with the representative prompt that is closest to the new initial prompt

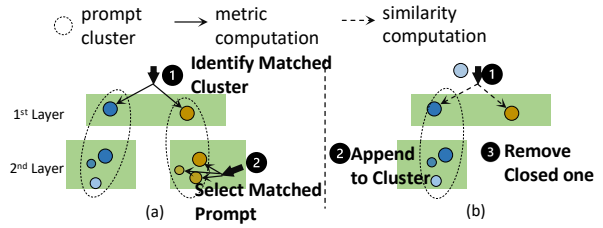


Figure 5: The illustration of performing (a) lookup, and (b) insertion & replacement on the two-layer data structure.

prompts. To identify an effective initial prompt for a given LPT job, a brute-force search over the entire prompt candidate set is computationally intensive, often taking hours. Our empirical study (Figure 10a in §6) and existing LLM research [67] demonstrate the prevalence of prompt similarity. This provides an opportunity to exclude unnecessary assessment of extensive poor prompt candidates and improve query efficiency [27].

To this end, we build a two-layer data structure for the prompt candidate set. Inspired by [47], we divide all the prompt candidates into clusters based on their activation feature similarity. We begin by using an LLM (e.g., Vicuna-7B) to extract the activation features of each prompt candidate. Then, we measure the prompt similarity based on the cosine distance between activation features. We also discuss other similarity metrics in §5.2. Finally, we adopt K-medoid clustering to group prompts with similar activation features into one cluster. Figure 5 (a) illustrates an example of this data structure. The first layer retains each cluster's medoid, further referred to as the *representative prompt* of the cluster. The second layer stores each prompt within these clusters. Hereafter, we detail how to perform the lookup, insertion & replacement operations.

4.3.2 Lookup. The lookup operation aims to identify an effective initial prompt for a given LPT job on this two-layer data structure. For each initial prompt candidate p, we introduce a metric $\texttt{score}(p)$, which is computed as the average loss on evaluation samples without requiring additional tuning on the training samples. We formulate $\texttt{score}(p)$ as follows:

$$\texttt{score}(p) = \frac{1}{\mathcal{D}_{\text{eval}}} \sum_{(d_i^{\text{in}}, d_i^{\text{tgt}}) \in \mathcal{D}_{\text{eval}}} \mathcal{L}\left(\text{concat}(p, d_i^{\text{in}}), d_i^{\text{tgt}}\right). \quad (1)$$

A smaller score value indicates a better initial prompt. We only use a small number of evaluation samples (e.g., 16) for prompt assessment. This requires minimal effort for labeling if the evaluation dataset is missing. Without performing any tuning, we can select the prompt with the minimal score as the most effective one.

The two-layer data structure facilitates the reduction of the number of prompt candidates needed to perform the metric computation in Eqn. 1. Figure 5 (a) illustrates the process of lookup operation. First, we identify the matched cluster by computing each representative prompt's score at the first layer. We identify the cluster with the lowest score. Next, we select the matched initial prompt by calculating the score for each prompt of the matched cluster at the second layer. We pick up the prompt with the lowest score as the optimal one. Assuming that each cluster contains the same number of prompt candidates, this two-layer data structure reduces the number of metric computations from C to $K + C/K$. Ideally, the minimal number of metric computations is $2\sqrt{C}$ when the optimal cluster is $K = \sqrt{C}$. Empirically, the two-layer data structure can reduce the overhead of the lookup operations by up to 40× while retaining the performance (§6.3).

4.3.3 Insertion & Replacement. When the service provider inserts a new initial prompt candidate, Figure 5 (b) shows the process of the insertion and replacement operation. First, we identify a similar cluster. We extract the activation features of the new candidate and measure the cosine distance of activation features between the new candidate and each cluster's representative candidate at the first layer. Different from the lookup operations, we do not involve metric computations (Eqn. 1) in this step. The cluster that attains the minimal cosine distance is denoted as C_{sim}. Second, we append this initial prompt into C_{sim} at the second layer. Third, the replacement operation is triggered when the number of initial prompt candidates exceeds the threshold (e.g., 3000) after insertion. We need to select one prompt candidate to remove it. To maximize the diversity of prompt candidates within the cluster, we choose the prompt candidate that has the minimal cosine distance to the representative prompt of C_{sim} and remove it to realize the replacement.

4.3.4 Two-layer Structure Discussion. The prevalent similarities among prompts suggest that clustering similar prompts can avoid unnecessary score assessment with minor speedup benefit loss. The study in §6.3 indicates that a two-layer data structure can identify effective initial prompts within 10 seconds. Additionally, we construct a three-layer structure using K-medoid clustering, but encounter convergence issues with Vicuna-7B and experience exorbitant construction overhead (up to tens of minutes). A two-layer structure can be constructed in five minutes without convergence issues across different LLMs, making it a more suitable choice.

4.4 Workload Scheduler

The Workload Scheduler realizes *runtime reusing* to mitigate the exorbitant GPU allocation overhead (§2.2), thus reducing the SLO violation and minimizing the resource cost. Figure 6 shows the overview of the Workload Scheduler, which manages two types of GPU pools, namely a single *shared cold* GPU pool and a set of *per-LLM warm* GPU pools. Each warm pool contains GPUs initialized to serve jobs for one specific LLM, i.e., each GPU has a pre-loaded PyTorch/CUDA runtime and LLM weights. The shared cold GPU pool contains GPUs without any pre-loaded GPU context[3].

Managing the per-LLM warm pools independently from the shared cold pool significantly reduces GPU allocation overhead

[3]Although cloud providers are free to use the GPUs from the cold pool for any jobs operating in their datacenter, for simplicity, we assume that the size of the cold GPU pool is fixed and GPUs can be allocated without any delays in time.

Table 5: Summary of notations in the Workload Scheduler.

Sym.	Definition
i	The index of LPT job
l	The index of LLM
k	The index of GPU in a warm GPU pool
a	The number of allocated GPUs
L	The number of LLMs.
R_l	The number of GPUs for each LLM l's warm GPU pool
A	The number of allocated GPUs in a warm GPU pool for each job
B	The number of GPUs added from the cold GPU pool for each warm GPU pool
T_i^{slo}	The SLO of job i
$T_i^{\text{warm}}(a)$	The estimated completion time of job i when assigned with a GPUs in a warm GPU pool
T^{cold}	The overhead of adding GPUs from the cold GPU pool to a warm GPU pool
$\mathcal{P}$	All pending LPT jobs
E_l	The list to store earliest timestamps for each GPU in the LLM l's warm GPU pool
E	A list to store each LLM's E_l

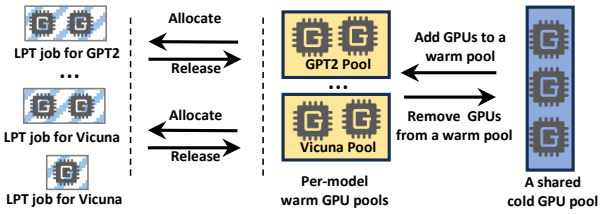


Figure 6: The Workload Scheduler consists of a single shared cold GPU pool and a set of per-LLM warm GPU pools. It rapidly allocates GPUs from the warm GPU pools to LPT jobs to optimize the SLO attainment. It also dynamically adjusts the number of GPUs added from the shared cold GPU pool to the warm GPU pools based on traffic and GPU availability.

without statically provisioning a large fixed-size cluster, as in ElasticFlow (§3.1). When the scheduler allocates GPUs to an LPT job from the corresponding warm pool, the job can start execution immediately, avoiding the delays of pre-loading the required runtime and LLM weights. Thus, the Workload Scheduler facilitates *runtime reusing* of LPT jobs of the same LLM. Since many users use the same LLMs [23, 62, 70], the system can operate efficiently while minimizing the operational cost by keeping only a small number of warm pools with a minimal set of GPUs. Unlike the GPUs in the warm pools, the GPUs in the cold pool do not impose any cost, so the providers can allocate them to any service running in datacenters.

For each incoming job in the pending queue, the Workload Scheduler determines the number of GPUs to be allocated based on its SLO and predicted execution time. It then allocates GPUs from the warm pool corresponding to the LLM type defined in the job attributes. To secure the SLO compliance, we predict the upper bound of job execution time as a product of the number of maximum remaining iterations and maximum time cost per iteration under given allocated GPUs with additional GPU allocation overhead. LPT jobs release their allocated GPUs to the corresponding warm pools upon completion. The scheduler monitors each pool's GPU usage, adding more GPUs from the cold pool to the warm pools of the LLMs that experience high demand and removing excessive GPUs from the warm pools of the LLMs.

Next, we detail two key resource allocation algorithms for LPT execution and one algorithm for the execution of Prompt Bank. Table 5 defines the notations used in this section.

4.4.1 GPU Allocation from a Warm Pool. This algorithm optimizes the SLO attainment by determining the number of GPUs in the warm GPU pools allocated to each job in the pending queue. Upon an LPT job's arrival, the scheduler adds it to the pending queue. Then, the scheduler periodically adjusts the GPU allocation for each job in the queue, allocating more GPUs from the corresponding warm pool whenever needed to achieve the job's SLO. Algorithm 1 illustrates this process. It starts by sorting LPT jobs in the pending queue based on their SLOs, and then progressively increases the number of allocated GPUs for each LPT job to meet its SLO until the warm pool is depleted (Lines 7-9).

Algorithm 1 GPU allocation from a warm pool.

1: **Input:** R_l that is the number of GPUs in the LLM l's warm pool, $\mathcal{P}_l$ that is the pending queue for LLM l.
2: **Output:** A that is the number of GPUs allocated to each job in the pending queue.
3:
4: Sort jobs based on T_i^{slo} in the ascending order
5: **for** each job i in $\mathcal{P}_l$ **do**
6: Set initial allocation $A_i = 1$
7: **while** $T_i^{\text{warm}}(A_i) > T_i^{\text{slo}}$ **and** $A_i \leq R_l$ **do**
8: $A_i = A_i + 1$ // *Allocate A_i GPU to the job*
9: **end while**
10: **if** $T_i^{\text{warm}}(A_i) \leq T_i^{\text{slo}}$ **then**
11: $R_l = R_l - A_i$ // *Update the number of GPUs in the warm GPU pool*
12: **else**
13: $A_i = 0$
14: **end if**
15: **end for**

Algorithm 2 GPU allocation from the cold pool.

1: **Input:** L LLM, pending queue with jobs $\mathcal{P}$, earliest timestamps of GPUs in the warm GPU pools E.
2: **Output:** The number of allocated GPUs B to each LLM's warm GPU pool.
3:
4: Sort $\mathcal{P}$ based on T_i^{slo} in the ascending order
5: **for** each job i and the corresponding LLM l in $\mathcal{P}$ **do**
6: // *Assess if the system can meet the job's SLO by delay its execution*
7: **if** DelaySchedulable(E, i, l) **then**
8: **continue**
9: **end if**
10: Set the initially allocated GPU number $A_i = 1$
11: // *Determine how many GPUs are needed to satisfy the job's SLO*
12: **while** $T_i(A_i) + T_l^{\text{cold}} > T_i^{\text{slo}}$ **and** $T_i^{\text{slo}} < T^{\text{cold}}$ **do**
13: $A_i = A_i + 1$
14: **end while**
15: **if** $T_i(A_i) + T_l^{\text{cold}} \leq T_i^{\text{slo}}$ **then**
16: // *Update the number of GPUs in each LLM's warm pool*
17: $B_l = B_l + A_i$
18: // *Update the earliest timestamps of GPUs in the warm GPU pools.*
19: Repeat A_i times to push back $T_i^{\text{warm}}(A_i) + T_l^{\text{cold}}$ into E_l.
20: **end if**
21: **end for**
22:
23: **function** DelaySchedulable(E, i, l)
24: $k = 1$, T^{cur} = current timestamp
25: Sort E_l in the ascending order
26: **while** $k \leq E_l.len$ **and** $T_i(k) - T^{\text{cur}} + E_{l,k} > T_i^{\text{slo}}$ **do**
27: $k = k + 1$
28: **end while**
29: **if** $k < E_l.len$ **and** $T_i(k) - T^{\text{cur}} + E_{l,k} \leq T_i^{\text{slo}}$ **then**
30: $E_{l,1:k} = T_i(k) + E_{l,k} - T^{\text{cur}}$
31: Sort E in the ascending order
32: **return** True
33: **end if**
34: **return** False
35: **end function**

4.4.2 GPU Allocation from the Cold Pool. The Workload Scheduler can periodically add and remove GPUs from the cold GPU pool to the corresponding warm GPU pool, following the demand for the corresponding LLM. The main objective of the algorithm is to

ensure each warm pool has the minimum number of GPUs required to ensure that the jobs can achieve their SLOs while minimizing the resource cost, which is proportional to the number of GPUs used by the jobs and present in the warm pools. Hence, the algorithm prioritizes jobs with shorter SLOs, delaying the execution of the jobs with longer SLOs and the jobs projected to miss SLOs.

Algorithm 2 details the steps that allocate GPUs from the cold pool to the warm pools. First, the algorithm sorts all pending jobs based on its SLO. Second, it identifies the LPT job i, scheduling of which can be delayed while still meeting its SLO by calling the DelaySchedulable function (Line 23-35). Third, if the system cannot meet the job's SLO, the algorithm progressively allocates more GPUs from the cold GPU pool to the job until it can ensure that SLO is met. The algorithm takes the GPU allocation overhead T_l^{cold} into account while determining whether the system can meet the job's SLO (Line 12). Last, if the system can meet the job i's SLO, the algorithm accumulates the number of added GPUs from the cold GPU pool to the corresponding warm GPU pool (Line 16-20).

The DelaySchedulable function determines if a job's SLO can be met by delaying its execution to a future moment when enough GPUs would be released by completing jobs to the warm pool instead of immediately adding more GPUs to the warm pool. We use $E_{l,k}$ to record the earliest timestamp when k GPUs in a warm GPU pool for LLM l are available. This information is obtained from predicting the completion time of each running LPT job, along with the subsequent release of GPUs to the respective warm pool. Additionally, to reduce the resource cost, the Workload Scheduler removes the GPUs from a warm pool if they do not serve any jobs for a time window, the size of which is set to one minute (§6.3).

Why DelaySchedulable function. Many SLO-aware resource allocation policies [35, 64, 77] expect to schedule jobs promptly to ensure SLO compliance for jobs. Delaying the execution of the jobs might risk the SLO violation. Owing to the speedup benefits provided by the Prompt Bank, many LPT jobs are completed earlier, leaving many GPUs idle. The Workload Scheduler takes advantage of this by strategically delaying the execution of LPT requests without requiring launching additional GPU resources. This allows the system to meet SLOs for more LPT requests with fewer GPUs.

4.4.3 Latency Budget for Prompt Bank. The Workload Scheduler needs to allocate GPUs to perform the Prompt Bank. Despite we support the sequential execution of Prompt Bank and LPT (§ 5.2) and reduce the overhead of the Prompt Bank within 10 seconds, it is possible that this overhead compromises SLO compliance for short requests. We empirically observe that the Prompt Bank can yield a 1.2-4.7× speedup compared to the induction initialization [80], an automatic prompt initialization baseline (detailed in §6.1). Therefore, we set a latency budget of 20% of the latency SLO to execute the Prompt Bank, ensuring that the minimum speedup benefits still outweigh the overhead of the Prompt Bank.

5 IMPLEMENTATION

5.1 Multi-GPU Execution

We implement LPT jobs with 2000 lines of Python code atop Transformers 2.4.1 and PyTorch 2.1 and deploy them as containerized GPU Knative functions to pre-load the LPT runtime and LLM weights in the GPU. Each Knative function accepts a set of parameters described in Table 3 and responds to users with the optimized prompt. We adopt the prompt-tuning algorithm in [49]. Note that `PromptTuner` is general and can support other implementations of LPT jobs and algorithms.

An LPT job demands multiple function instances to deliver multi-GPU execution. We implement the multi-GPU execution atop LambdaML [44], which employs Memcached as the storage channel to realize the synchronous cross-GPU communication between function instances. Each function instance belonging to an LPT job is assigned an IP address and port to connect with other function instances, incurring at most a 2-second overhead. The storage channel incurs negligible communication overhead due to its small size.

5.2 Prompt Bank

We implement the Prompt Bank with ~1000 lines of Python code atop Transformers 2.4.1 and PyTorch 2.1. It is also deployed as a Knative function with one GPU, which accepts parameters, including the dataset and initial prompt described in Table 3, and returns the optimized initial prompt for subsequent prompt-tuning.

Offline Phase. For each LPT job, we use the corresponding LLM to extract the activation features of gathered prompts and empirically set the number of clusters in the two-layer data structure as 50. Moreover, we employ Scipy 1.10.1 to execute K-medoid clustering. Despite exploring alternative distance metrics, including Manhattan and Euclidean distances, we encounter convergence issues. The lack of convergence may stem from imbalances in the numerical value scales within the activation features of various prompts. The storage size remains under 5 GB for each LLM. We have detailed the insertion and replacement operation in §4.3. If the service provider introduces a new LLM, it needs to re-extract the activation features of all gathered prompts to construct the two-layer data structure.

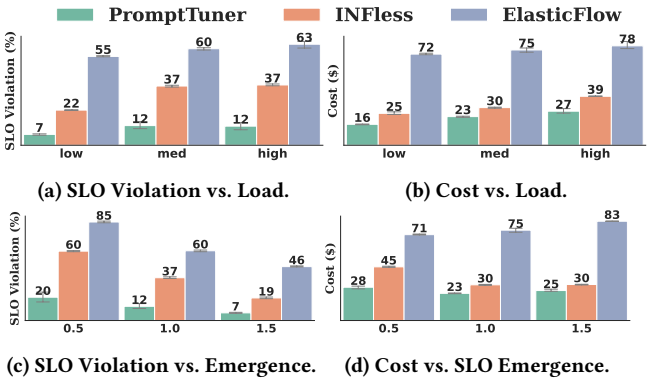
Online Phase. The Workload Scheduler utilizes the latency budget to decide whether to perform the Prompt Bank for incoming request. We notice that the implementation and runtime of the Prompt Bank and LPT can be shared. Hence, we incorporate the Prompt Bank into the corresponding LPT job. In other words, the Prompt Bank and LPT job run sequentially on their assigned GPUs.

5.3 Workload Scheduler

Pre-loaded Runtime. Each job requires multiple function instances for multi-GPU execution. Knative provides an autoscaling mechanism to maintain function instances serving future requests.

GPU Allocation from a Warm Pool. This algorithm aims to perform rapid GPU allocation from a warm pool to an LPT job. Hence, we conduct the round-based GPU allocation every 50 milliseconds, which is negligible compared to minutes-level latency SLO. It operates within the distributed control plane of Knative to assign GPUs in the warm GPU pools to corresponding LPT jobs.

GPU Allocation from a Cold Pool. This algorithm is implemented inside the distributed control plane of Knative, and the interval is set as 50 milliseconds to add and remove GPUs for warm pools promptly. Moreover, this algorithm tracks the profiled information, including the allocation overhead for each LLM and the job throughput. It then continuously updates this information to the scheduler to avoid high estimation errors.



(a) SLO Violation vs. Load. (b) Cost vs. Load.

(c) SLO Violation vs. Emergence. (d) Cost vs. SLO Emergence.

Figure 7: End-to-end performance under different loads (a-b) and different SLO emergence (c-d).

6 EVALUATION

6.1 Experimental Setup

Testbed. We set up `PromptTuner` in a physical GPU cluster. Each GPU server has eight NVIDIA A100-80GB GPUs and one 200Gbs HDR InfiniBand. It features an Intel Xeon 8369B 2.90GHz CPU with 64 cores, 256 GB RAM, and PCIe-III. `PromptTuner` provisions at most 4 GPU servers. We adopt Memcached 1.5.22 to set up an Elastic Cache service for communication among GPU servers. **Workload Construction.** We evaluate three representative LLMs (GPT-Base, GPT-Large, Vicuna-7B) on 12 datasets, as shown in Table 6. To further increase the diversity of LPT workloads, we sample each dataset into ten exclusive partitions and construct 120 tasks for each LLM. For each LPT task in Table 6, we measure the average accuracy over 20 initial prompts randomly selected from the Prompt Bank as the target accuracy. This primarily ensures that the evaluated LPT jobs, using different initial prompts in the prompt sensitivity analysis of §2.2, can reach such accuracy. We also evaluate LLaMA-30B and Qwen7B-R1[4] to present the system efficiency in the context of large-scale models and long-sequence inputs.

In our experiments, we sample three 20-minute LPT traces from our anonymous institute's data center to construct low (41/55/42), medium (77/71/65), and high (99/85/76) loads for different LLMs (GPT2-B/GPT2-L/V7B). We sample 59 and 70 requests for LLaMA-30B and Qwen7B-R1 as medium load. These traces include the submission time, the number of allocated GPUs, and the duration of each LPT job. The job durations vary from a few seconds to several minutes. We follow the minute granularity of the submission time attribute to invoke the request with an exponential distribution. The product of the job duration and number of allocated GPUs is used to assign LPT tasks for each job. It randomly chooses one task in Table 6 to match the GPU time of such a job. We set each job's SLO as its duration multiplied by a parameter S added by the resource allocation overhead. We denote S as SLO emergence. A small S indicates a more emergent SLO.

Baselines. `PromptTuner` is the first SLO-aware system for LPT workloads. We choose two state-of-the-art DL cluster management systems as the baselines: (1) **INFless** [77]: this is an efficient SLO-aware and cost-effective system for DL inference. It supports traffic-based autoscaling and runtime reusing. To ensure a fair comparison, we reinforce INFless with the multi-GPU execution and Prompt Bank. (2) **ElasticFlow** [35]: this is an SLO-aware DL training system. It dynamically adjusts the number of GPUs for each job. However, it does not support runtime reusing. The Prompt Bank is also incorporated into ElasticFlow.

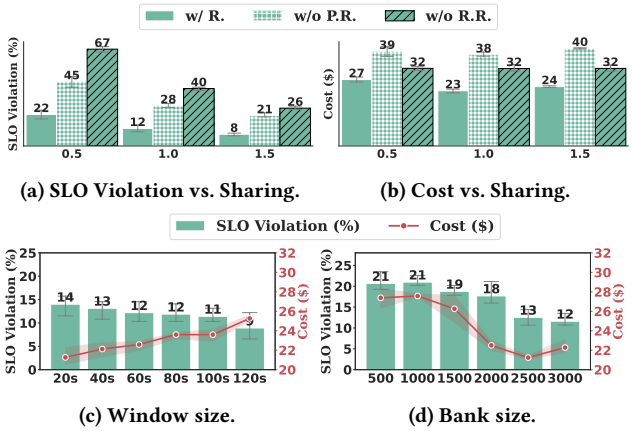
To evaluate the quality of initial prompts from the Prompt Bank, we consider two baselines: (1) **Ideal**: this is the prompt with the best ITA performance. For easy computation, we use `score` to shortlist 20 prompts and select the best one based on their ITA performance. However, it is computationally infeasible in practice. (2) **Induction** [80]: it is an automatic prompt initialization method that leverages a set of demonstrative examples to guide the LLM to generate an appropriate initial prompt. However, it only works for simple tasks, and the LLM should possess strong capabilities.

Evaluation Metrics. We consider two evaluation metrics: (1) the ratio of workloads that meet the SLOs. We use the SLO violation as the metric. (2) The total resource cost. We estimate the cost based on the price of the AWS `p4de.24xlarge` instance. The storage costs are billed on GB/hour (AWS elastic cache). We take the minimal possible price for storing transferred data, accounting for the small communication time. For the Prompt Bank, we choose ITA to demonstrate the high quality of selected initial prompts.

[4]We use soft prompt tuning algorithm [50] and task GSM8K [24] to evaluate Qwen7B-R1. The selected initial textual prompt is positioned before the soft prompt.

Table 6: LPT tasks and targeted accuracy: [B] and [R] refer to the `bleu` score and `rouge` score respectively.

Task Description	Dataset	Accuracy	Task Description	Dataset	Accuracy
Dialog	DA [49]	54 [B]	Summarization	CNNDM [49]	34 [B]
	PC [84]	19 [B]		SAMSUM [33]	46 [B]
Question Answer	COQAQG [63]	51 [B]		XSUM [49]	40 [B]
	QUORA [45]	21 [B]	Story Generation	CMV [39]	26 [R]
Text Generation	WIKIBIO [49]	70 [R]		WP [28]	20 [R]
	WIKIP [13]	22 [R]		ROC [56]	25 [R]



(a) SLO Violation vs. Sharing. (b) Cost vs. Sharing.

(c) Window size. (d) Bank size.

Figure 8: Feature evalutions: (a-b) The impact of prompt reusing (P.R.) and runtime reusing (R.R.) on SLO violation and cost over different SLO levels. (c-d) SLO violation and cost of `PromptTuner` under varying window sizes (c) and prompt bank sizes (d).

6.2 End-to-end Performance

We compare the end-to-end performance of `PromptTuner` with two baselines (INFless and ElasticFlow) under various environments

Table 7: Heavy Workload Evaluation.

Heavy Setting	Metric	PromptTuner	INFless	ElasticFlow
LLaMA-30B	SLO Violation ↓(%)	28.4	38.9	82.3
	Cost ↓ ($)	38.8	46.4	69.4
Qwen7B-R1	SLO Violation ↓(%)	23.1	36.2	74.9
	Cost ↓ ($)	30.7	42.8	70.1
Large-Scale	SLO Violation ↓ (%)	25.4	57.1	78.2
	Cost ↓ ($)	57.2	65.9	99.1

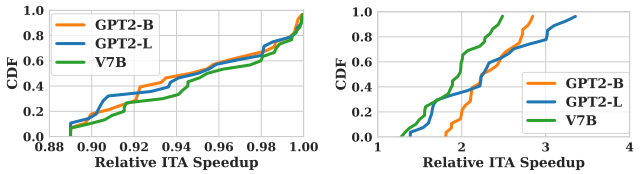
in a physical cluster. Our empirical evaluation in Figure 7 simultaneously serves requests for three LLMs in this experiment. First, Figures 7a and 7b present the SLO violation and cost of these systems under different job loads, respectively. PromptTuner achieves 15-25% SLO violation reduction compared to INFless and 48-51% SLO violation reduction compared to ElasticFlow. Interestingly, the increased loads provide more opportunities to perform *runtime reusing*. Thus, the SLO violation does not increase significantly from medium to high loads. The heavy job load increases the SLO violation and cost, and PromptTuner demonstrates higher superiority than baselines under heavier job loads.

Second, we explore the SLO violation and cost of these systems in different emergencies of SLOs, focusing on a medium job load for simplicity. As shown in Figures 7c and 7d, PromptTuner consistently outperforms baseline systems with at least 10% SLO violation reduction across varying SLO levels. When the SLO emergence is set as 0.5, more LPT jobs are executed on multiple GPUs. Thus, INFless is more likely to suffer from the long waiting delay incurred by the instance initialization, as discussed in 3.2. Hence, INFless even achieves very high SLO violation as ElasticFlow. In terms of resource cost, compared to INFless, PromptTuner reduces the expenses by 38%, 23%, and 17% at SLO levels $S = 0.5$, 1.0, and 1.5, respectively. Compared to ElasticFlow, the cost savings of PromptTuner are even more pronounced: up to 70% at $S = 1.5$. In summary, PromptTuner stands out for its superior performance in both SLO violation reduction and cost efficiency.

LLaMA-30B Evaluation. A single replica of LLaMA-30B is hosted across four GPUs, with tensor parallelism employed to facilitate prompt tuning. Due to limited GPU availability, the experiment for LLaMA-30B is conducted separately. Table 7 compares the SLO violation rates and resource costs of PromptTuner with two baseline methods. PromptTuner reduces the SLO violation rate by 1.36-2.90× and resource costs by 1.20-1.79× compared to another two baselines. These results suggest that PromptTuner sustains its superior performance when managing heavy LLM workloads.

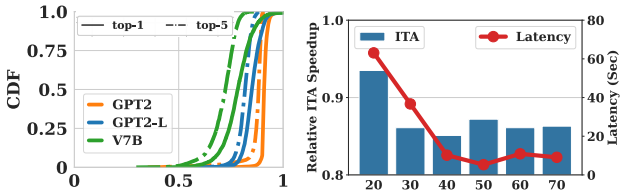
Qwen7B-R1 Evaluation. The maximum sequence length for Qwen7B-R1 is 32,768 tokens. To accommodate this, we utilize four GPUs to host a replica of Qwen7B-R1 using tensor parallelism. Furthermore, we employ a cluster of 32 GPUs to evaluate the performance of Qwen7B-R1. Table 7 compares the SLO violation rates and resource costs of PromptTuner with two baseline methods. PromptTuner reduces the SLO violation rate by 1.56-3.24× and resource costs by 1.39-2.28× compared to another two baselines. These results suggest that PromptTuner sustains its superior performance when handling long-sequence samples.

Scalability Evaluation. We measure the performance of PromptTuner in large-scale GPU clusters. Because of limited available GPUs, we perform one experiment for each system on a cluster of up to 96 GPUs. We increase Figure 7c's medium job loads proportionally



(a) Score versus Ideal. **(b) Score versus Induction.**

Figure 9: Distributions of relative ITA speedup of score candidate to (a) ideal candidate; (b) induction candidate.



(a) Prompt Similarity. **(b) Varying K.**

Figure 10: Performance of the *two-layer structure*: (a) Distribution of prompt similarity; (b) latency and average relative TTA of varying numbers of groups.

to match the maximal amount of provisioned GPUs. Table 7 compares the SLO violation and resource costs of PromptTuner with the other two baselines. The performance gain of PromptTuner over other baselines is enlarged with the increase of provisioned GPUs. With more workloads and GPUs, PromptTuner can exploit dynamic resource allocation to obtain better scheduling decisions. Additionally, the average/maximal scheduling overhead is 13/67 ms, making it not a performance bottleneck in PromptTuner. The small scheduling overhead strengthens our belief that PromptTuner can attain satisfactory performance in a large-scale GPU cluster.

6.3 Evaluation of Key Components

Prompt & Runtime Reusing. Figures 8a and 8b show the benefits of *prompt reusing* (P.R.) and *runtime reusing* (R.R.) to SLO guarantee and cost-effectiveness over different SLO levels. First, prompt reusing can reduce SLO violations by 13-23% and cost savings by 30-40%. For the stringent SLO, the Prompt Bank (i.e., prompt reusing) saves GPU time by satisfying more SLOs of LPT jobs. In relaxed SLO scenarios, the Prompt Bank can reduce the number of GPUs allocated to warm GPU pools. PromptTuner particularly benefits from *runtime reusing* by mitigating the GPU allocation overhead, enhancing SLO attainment. However, the cost savings from *runtime reusing* are not comparable to that of *prompt reusing*.

Impact of Allocation from Warm GPU pool. The GPU allocation from a warm GPU pool enables simultaneous multi-GPU allocation, effectively mitigating the initialization overhead associated with multi-GPU instances, as discussed in § 3.2. We implement a baseline policy (without the warm allocator) that immediately allocates warm GPUs to each function instance (§5.1), disregarding the constraints of simultaneous allocation within the same LPT request. The second and third columns of Table 8 demonstrate that our proposed allocation policy reduces the SLO violation rate by a factor of 2.24, with only a modest increase in cost at an SLO level of 1.0 and a medium job load. These results suggest that the warm GPU allocator has a positive impact on SLO attainment.

Table 8: Impact of key components in Workload Scheduler.

Metric	Workload Scheduler	w/o Warm Allocator	w/o DelaySchedulable	w/o Latency Budget
SLO Violation ↓ (%)	12.4	27.8	15.6	16.3
Cost ↓ ($)	22.9	20.9	26.6	23.2

Impact of DelaySchedulable Function. The DelaySchedulable function delays the execution of certain LPT requests to fully utilize GPUs from the warm GPU pool, reducing the GPU amount in a warm GPU pool while minimizing the SLO violation rate. As shown in the second and fourth columns of Table 8, this function reduces the SLO violation rate and resource costs by 1.27× and 1.16×, respectively, at an SLO level of 1.0 and a medium job load. This empirically demonstrates the effectiveness of the DelaySchedulable function in optimizing both SLO violations and resource costs.

Impact of Latency Budget. To evaluate the effectiveness of the latency budget, we establish a baseline in which the Prompt Bank is triggered for each incoming request. As shown in Table 8, the latency budget leads to a reduction in both SLO violations and costs by 1.31× and 1.02× respectively. The latency budget proves to be a beneficial operation in `PromptTuner`.

Window Size of Allocation from Cold GPU Pool. We investigate how the window size of the cold GPU allocator affects the performance of `PromptTuner`. A smaller window size causes GPUs to be removed from the warm GPU pool frequently, increasing the SLO violation. A larger window size may make `PromptTuner` less responsive to traffic, increasing resource costs. Figure 8c presents various window sizes and shows that setting 60 seconds strikes a satisfactory balance between the SLO violation and cost.

Varying Prompt Bank Size. Due to the heavy evaluation costs of Prompt Bank, we only choose GPT2-Base, GPT2-Large, and Vicuna-7B to investigate. We analyze the impact of the number of prompt candidates on the scheduling performance of `PromptTuner`. We set the maximum size as 3,000 due to the limited number of free-of-use high-quality prompts. Figure 8d depicts that the SLO violation and cost vary over different sizes of the Prompt Bank. A larger Prompt Bank incurs larger execution overhead, while a smaller one may reduce the potential speedup benefits derived from effective initial prompts. When the size drops to 2,000, both SLO violations and costs increase significantly, highlighting the importance of maintaining prompt diversity of the Prompt Bank.

Score Metric. We term *score candidate*, *ideal candidate*, and *induction candidate* as the prompts selected by proposed metric (Eqn. 1), ideal baseline, and induction baseline, respectively. Figure 9a shows the distributions of relative ITA performance between the score candidate and ideal candidate from 120 LPT tasks of three LLMs. The ITA performance of most score candidates exceeds 90% of that of ideal candidates. Figure 9b presents the distributions of relative ITA performance between the score candidates and induction candidates. The score candidates outperform the induction candidates and yield at least 1.81×, 1.38×, 1.28× ITA speedup for GPT-Base, GPT-Large, and Vicuna-7B, respectively. GPT-B achieves the highest ITA speedup benefits (1.8–2.8×), as its generality is weak compared to Vicuna-7B, leading to less effective initial prompt generation by itself. Conversely, V-7B achieves a minimum ITA speedup of 1.28× compared to induction candidates. This analysis highlights that our `score` method can identify near-optimal initial prompts, delivering superior ITA performance over induction initialization across various tasks and LLMs.

Two-layer Data Structure. Figure 10a shows the CDF of top-1 (solid line) and top-5 (dashed line) cosine similarity of the activation features in our curated prompt candidate set across varying LLMs. This high similarity motivates us to design a two-level data structure to group similar prompt candidates. Furthermore, we verify whether clustering similar prompt candidates degrades the ITA performance of the identified initial prompt and reduces the selection latency. We fix the number of evaluation samples to 16 and the LLM to GPT2-Base. Figure 10b shows the impact of the cluster counts on the relative ITA speedup compared to the ideal candidate and the average selection latency. Using more groups does not cause considerable ITA performance loss. For GPT2-Large and Vicuna-7B, the impact of cluster counts on ITA speedup presents a similar trend. Also, we are concerned about the latency overhead and set the number of clusters as 50 for `PromptTuner`. Then the average latency is 5.3 seconds for GPT2-Base, 6.1 seconds for GPT2-Large, and 9.2 seconds for Vicuna-7B, respectively. As a reference, it takes approximately 2.5, 2.9, and 4.5 hours for GPT-Base, GPT-Large, and Vicuna-7B, respectively, when K is set to 1. Overall, the two-layer data structure efficiently balances speedup benefits with the overhead of initial prompt selection.

7 RELATED WORKS

LLM Systems. The significance of LLMs attracts researchers to design specialized systems to support their execution. Many LLM systems [40, 43, 57, 58, 85] focus on automatic discovery of parallelism strategies for deploying LLM training on thousands of GPUs. In our scenario, LPT has significantly smaller communication overhead and GPU requests (at most tens) than LLM training. Thus, these strategies are not well-suited to LPT. Many LLM inference works [20, 30, 46, 55, 66, 81] mainly address the mismatch between the computation-bound prefill phase and the memory-bound decoding phase, along with the heavy KV cache. However, since many prompt tuning algorithms do not involve the decoding phase and KV cache management, `PromptTuner` cannot directly adopt solutions proposed by these inference systems. Instead, `PromptTuner` utilizes prompt sharing to accelerate LPT.

Parameter-Efficient Fine-Tuning. Recent works design various parameter-efficient fine-tuning methods for LLMs, including LoRA [37], Prefix-tuning [50], P-Tuning [51], and Prompt tuning [48, 52]. Prefix-tuning and P-tuning are extensions of prompt tuning, and `PromptTuner` can treat them as LPT workloads to determine the GPU allocation. The LLM community has also introduced other methods like gradient-based approaches [49, 50], zero order [53], and reinforcement learning [25] to optimize the prompts.

8 CONCLUSION

This paper presents `PromptTuner`, an SLO-aware elastic system for managing LPT workloads. We take advantage of *prompt reusing* to develop the Prompt Bank for expediting LPT workloads. We also exploit the *runtime reusing* to reduce the GPU allocation overhead for resource elasticity. Our extensive experiments demonstrate the superiority of `PromptTuner` in SLO attainment and cost reduction.

REFERENCES

[1] https://promptbase.com/. Accessed: 2025-01.
[2] https://claude.ai/. Accessed: 2025-01.
[3] Alibaba Cloud. https://www.alibabacloud.com/. 2023.
[4] Amazon Web Services. https://aws.amazon.com/. 2025.
[5] Azure Cloud. https://azure.microsoft.com/. 2025.
[6] mergeflow. https://mergeflow.com/.
[7] Midjourney. https://www.midjourney.com.
[8] Promptthero. https://promptthero.com/.
[9] Promptperfect. https://promptperfect.jina.ai/home.
[10] Nvidia a100. https://www.nvidia.com/en-sg/data-center/a100/, 2022.
[11] Nvidia h100. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper, 2022.
[12] Memcached. https://memcached.org/, 2025.
[13] Wikiplots. https://github.com/markriedl/WikiPlots, 2025.
[14] Toyin D. Aguda, Suchetha Siddagangappa, Elena Kochkina, Simerjot Kaur, Dongsheng Wang, and Charese Smiley. Large language models as financial data annotators: A study on effectiveness and efficiency. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 10124–10145, Torino, Italia, May 2024. ELRA and ICCL.
[15] Portkey AI. Portkey ai, 2025. Accessed: 2025-01.
[16] Anthropic. Claude ai. https://claude.ai/, 2025.
[17] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023.
[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.
[19] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*, 2023.
[20] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547*, 2023.
[21] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023.
[22] Zhaoyun Chen, Wei Quan, Mei Wen, Jianbin Fang, Jie Yu, Chunyuan Zhang, and Lei Luo. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2020.
[23] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.
[24] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
[25] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. *arXiv preprint arXiv:2205.12548*, 2022.
[26] Viet-Tung Do, Van-Khanh Hoang, Duy-Hung Nguyen, Shahab Sabahi, Jeff Yang, Hajime Hotta, Minh-Tien Nguyen, and Hung Le. Automatic prompt selection for large language models. *arXiv preprint arXiv:2404.02717*, 2024.
[27] Fatih Erikli. Awesome chatgpt prompts. https://github.com/f/awesome-chatgpt-prompts/, 2022.
[28] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia, July 2018. Association for Computational Linguistics.
[29] FlowGPT. Flowgpt: Prompt engineering for the future. https://flowgpt.com/, 2024. Accessed: 2025-01.
[30] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–153. USENIX Association, 2024.
[31] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, 2021.
[32] Wei Gao, Zhisheng Ye, Peng Sun, Tianwei Zhang, and Yonggang Wen. Unisched: A unified scheduler for deep learning training jobs with different user demands. *IEEE Transactions on Computers*, 2024.
[33] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. SAMSum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pages 70–79, Hong Kong, China, November 2019. Association for Computational Linguistics.
[34] Google. Gemini ai. https://gemini.google.com/, 2025.
[35] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 266–280, 2023.
[36] Or Honovich, Uri Shaham, Samuel R Bowman, and Omer Levy. Instruction induction: From few examples to natural language task descriptions. *arXiv preprint arXiv:2205.10782*, 2022.
[37] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
[38] 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.
[39] Xinyu Hua and Lu Wang. PAIR: Planning and iterative refinement in pre-trained transformers for long text generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 781–793, Online, November 2020. Association for Computational Linguistics.
[40] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
[41] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.
[42] 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, 2019.
[43] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems*, MLSys '19, 2019.
[44] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, pages 857–871, 2021.
[45] Ashutosh Kumar, Kabir Ahuja, Raghuram Vadapalli, and Partha Talukdar. Syntax-guided controlled generation of paraphrases. *Transactions of the Association for Computational Linguistics*, 8:329–345, 2020.
[46] 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*, pages 611–626, 2023.
[47] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. Modelkeeper: Accelerating dnn training via automated training warmup. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.
[48] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.
[49] Junyi Li, Tianyi Tang, Gaole He, Jinhao Jiang, Xiaoxuan Hu, Puzhao Xie, Zhipeng Chen, Zhuohao Yu, Wayne Xin Zhao, and Ji-Rong Wen. TextBox: A unified, modularized, and extensible framework for text generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*, pages 30–39, Online, August 2021. Association for Computational Linguistics.
[50] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.
[51] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602*, 2021.
[52] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023.
[53] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *arXiv preprint arXiv:2305.17333*, 2023.

[54] Manus. Manus. https://manus.im/, 2025-04.

[55] ModelTC. Lightllm. https://github.com/ModelTC/lightllm/, 2025.

[56] Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. A corpus and cloze evaluation for deeper understanding of commonsense stories. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 839–849, San Diego, California, June 2016. Association for Computational Linguistics.

[57] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[58] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[59] OpenAI. Gpt-4 technical report, 2023.

[60] OpenAI. Chatgpt. https://chatgpt.com/, 2025.

[61] Promptrr. Promptrr. https://promptrr.io/, 2024. Accessed: 2025-01.

[62] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[63] Siva Reddy, Danqi Chen, and Christopher D. Manning. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019.

[64] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference*, USENIX ATC '21, 2021.

[65] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, 2020.

[66] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[67] Yusheng Su, Xiaozhi Wang, Yujia Qin, Chi-Min Chan, Yankai Lin, Huadong Wang, Kaiyue Wen, Zhiyuan Liu, Peng Li, Juanzi Li, Lei Hou, Maosong Sun, and Jie Zhou. On transferability of prompt tuning for natural language processing. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3949–3969, Seattle, United States, July 2022. Association for Computational Linguistics.

[68] TextCortex. Ai prompt marketplace - textcortex. https://textcortex.com/templates/ai-prompt-marketplace, 2024. Accessed: 2025-01.

[69] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[70] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: open and efficient foundation language models, 2023. *URL https://arxiv. org/abs/2302.13971*.

[71] Nilesh Tripuraneni, Michael Jordan, and Chi Jin. On the theory of transfer learning: The importance of task diversity. *Advances in Neural Information Processing Systems*, 33:7852–7862, 2020.

[72] Tu Vu, Tong Wang, Tsendsuren Munkhdalai, Alessandro Sordoni, Adam Trischler, Andrew Mattarella-Micke, Subhransu Maji, and Mohit Iyyer. Exploring and predicting transferability across nlp tasks. *arXiv preprint arXiv:2005.00770*, 2020.

[73] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompting the coding ability of large language models with the interactive chain of repair, 2024.

[74] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM*, pages 1288–1296, 2019.

[75] Shuyang Wang, Somayeh Moazeni, and Diego Klabjan. A sequential optimal learning approach to automated prompt engineering in large language models. *arXiv preprint arXiv:2501.03508v1*, January 2025.

[76] 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, 2022.

[77] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[78] Zichao Yang, Heng Wu, Yuanjia Xu, Yuewen Wu, Hua Zhong, and Wenbo Zhang. Hydra: Deadline-aware and efficiency-oriented scheduling for deep learning jobs on heterogeneous gpus. *IEEE Trans. Computers*, 2023.

[79] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

[80] Qinyuan Ye, Maxamed Axmed, Reid Pryzant, and Fereshte Khani. Prompt engineering a prompt engineer. *arXiv preprint arXiv:2311.05661*, 2023.

[81] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[82] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622*, 2023.

[83] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[84] Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2204–2213, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[85] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[86] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.