Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving

Shan Yu[1], Jiarong Xing[2,3*], Yifan Qiao[2], Mingyuan Ma[4], Yangmin Li[5], Yang Wang[6], Shuo Yang[2], Zhiqiang Xie[7], Shiyi Cao[2], Ke Bao[8], Ion Stoica[2], Harry Xu[1], Ying Sheng[1*]

[1]UCLA [2]UC Berkeley [3]Rice University [4]Harvard University
[5]Carnegie Mellon University [6]Intel [7]Stanford University [8]LMSYS

Abstract

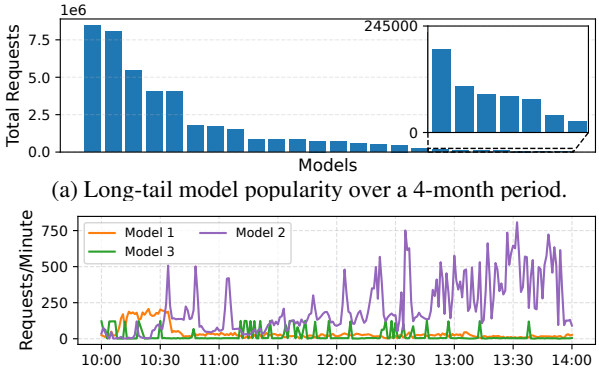
Serving large language models (LLMs) is expensive, especially for providers hosting many models, making cost reduction essential. The unique workload patterns of serving multiple LLMs (*i.e.*, multi-LLM serving) create new opportunities and challenges for this task. The long-tail popularity of models and their long idle periods present opportunities to improve utilization through GPU sharing. However, existing GPU sharing systems lack the ability to adjust their resource allocation and sharing policies at runtime, making them ineffective at meeting latency service-level objectives (SLOs) under rapidly fluctuating workloads.

This paper presents Prism, a multi-LLM serving system that unleashes the full potential of GPU sharing to achieve both cost efficiency and SLO attainment. At its core, Prism tackles a key limitation of existing systems—the lack of *cross-model memory coordination*, which is essential for flexibly sharing GPU memory across models under dynamic workloads. Prism achieves this with two key designs. First, it supports on-demand memory allocation by dynamically mapping physical to virtual memory pages, allowing flexible memory redistribution among models that space- and time-share a GPU. Second, it improves memory efficiency through a two-level scheduling policy that dynamically adjusts sharing strategies based on models' runtime demands. Evaluations on real-world traces show that Prism achieves more than 2× cost savings and 3.3× SLO attainment compared to state-of-the-art systems.

1 Introduction

Serving large language models (LLMs) incurs substantial costs [9], especially for inference providers (*e.g.*, Google [15], AWS [8], Hyperbolic [26], Novita AI [4], Together AI [6]) that host thousands of base models and user-submitted fine-tuned models [5]. These models vary in sizes (∼1B–100B+ parameters) and workloads, requiring a large GPU fleet to meet their performance requirements. As a result, reducing inference costs while maintaining performance (*e.g.*, latency objectives) has become a major focus for providers to make their solutions more competitive and profitable.

Our in-depth analysis of four production traces (§3.1) reveals that serving multiple LLMs (*i.e.*, multi-LLM serving) introduces unique workload patterns that create both opportunities and challenges for cost-efficient serving. In particular, we identify four key characteristics that must be considered.



(a) Long-tail model popularity over a 4-month period.

(b) Request rates over a 4-hour period in a day.

Figure 1: The workload characteristics of multi-LLM serving from a representative service provider, Hyperbolic [26].

C1: Long-tail model popularity. The popularity of models follows a long-tail distribution. Figure 1a shows the number of requests received by models served by a popular provider, Hyperbolic [26]. As shown, a small number of highly popular models account for the majority of requests, while a large number of models in the tail receive <100 requests per hour. *C2: Frequent idle periods.* Models often have interleaving idle periods with no incoming requests. In this trace, over 60% of models experience more than 1,000 idle periods, each longer than 10 seconds. *C3: Rapid workload fluctuations.* Workload patterns change rapidly over time. Figure 1b shows the request rates of three models in the trace, illustrating that the workload can fluctuate by more than 5× in just one minute. *C4: Diverse service-level objectives (SLOs).* Providers typically assign different latency SLOs (∼1–10+ seconds) to models based on customer requirements and workloads [40,43,55]. Maintaining these SLOs is essential, as violations can result in customer compensation and reputational damage [12].

Because of C1 and C2, the most common practice today—dedicating a single or a group of GPUs to each model—often leads to significant resource underutilization, as individual models in the popularity tail or during idle periods leave GPU resources unused. A simple back-of-the-envelope calculation on the above trace shows that this serving strategy would require around 120 H100-80G GPUs in total, but most would remain idle, with an average compute utilization of just 1.3% and memory utilization of 28.4%.

Several solutions have been proposed to improve utilization by sharing GPUs across multiple LLMs, as summarized in Ta-

*Corresponding authors.

	Dedicated	S-Pa.	Muxs.	QLM	Ours
Space sharing by model colocation	✗	✓	✓	✗	✓
Time sharing by model swapping	✗	✗	✗	✓	✓
Runtime sharing policy adaptation	✗	✗	✗	✗	✓
Cost saving & meeting SLOs	✗	✗	✗	✗	✓
# GPUs needed for 8 models	8	7	5	8	2
SLO attainment with 2 GPUs	-	39%	51%	45%	99%

Table 1: Comparison of GPU sharing systems. S-Pa.: static GPU partition; Muxs.: MuxServe. The last two rows show the number of GPUs needed for 8 models to achieve 99% SLO attainment and the SLO attainment when limited to 2 GPUs.

ble 1. First, MIG [38] and fractional GPUs [1, 7, 46] statically partition GPU resources into slices, allowing multiple models to run together using space sharing. MuxServe extends this by allowing models colocated on the same GPU group to share resources without fixed per-model limits. QLM [40] attempts to share GPUs in time by swapping models in and out.

However, these approaches statically allocate resources or use a fixed sharing policy, making them ineffective at handling dynamic workloads (C3). As a result, they fall short of achieving cost efficiency while meeting latency SLOs (C4). Specifically, static partition (*e.g.*, MIG [38] or fractional GPU [1, 7, 46]) prevents slices from sharing unused capacity, forcing systems to provision for peak workloads, leading to low utilization during off-peak periods. MuxServe [16] derives model colocation strategies (which models to share a GPU) based on offline profiling, lacking the ability to adjust colocation strategies dynamically based on workload changes. For example, even if a model becomes idle, it continues occupying GPU memory. QLM [40] enforces time sharing across all LLMs through model swapping. However, its swapping incurs significant latency overhead, making frequent swaps of latency-sensitive models prone to SLO violations.

Insight. To unleash the full potential of GPU sharing for multi-LLM serving, we must flexibly combine space and time sharing and dynamically adjust resource allocation in response to runtime workload variations. For example, multiple low-demand models can share a GPU during steady periods (space sharing), but when one becomes idle, its memory can be reallocated to a surging high-demand model (time sharing). The fundamental challenge is enabling *flexible* and *demand-aware cross-model memory coordination*,[1] which is missing in existing systems. *Flexibility* ensures models can promptly acquire the resources needed to meet their latency SLOs, while *demand-awareness* ensures that available memory is shared wisely across models to maximize overall SLO attainment. Together, these two properties enable cost efficiency while maintaining performance objectives in multi-LLM serving.

In this work, we design Prism, a system that fully unleashes the power of GPU sharing for cost-efficient multi-LLM serving. Prism tackles two key challenges in its design, achieving flexibility and demand-awareness in cross-model memory coordination.

Challenge 1: How to enable flexible cross-model memory coordination? Today's LLM serving engines (*e.g.*, SGLang [64] and vLLM [29]) are designed primarily for single-model serving. They rely on *per-model static* memory allocation, where both virtual and physical GPU memory are pre-allocated and remain fixed throughout the serving lifecycle. While PagedAttention [29] is commonly adopted, it operates at the *application level*, managing only the KV cache within each model's pre-allocated memory, without support for memory coordination across models. This design introduces three key limitations: (1) colocated models cannot utilize each other's unused memory; (2) memory released by evicted models cannot be quickly reclaimed by others; (3) evicted models cannot be promptly reactivated upon receiving new requests due to slow memory reallocation.

Flexible memory coordination requires dynamic memory redistribution across multiple models, which in turn demands memory management at the *system level*. Prism enables this through its `kvcached` component, positioned beneath PagedAttention. At its core is an on-demand allocation mechanism that decouples virtual and physical GPU memory. Instead of allocating physical memory during initialization, serving engines reserve only virtual address space, with physical memory allocated and mapped dynamically at runtime. This design allows flexible redistribution of memory between models for both KV caches and model weights. Building on this foundation, Prism also enables fast model activation by employing pre-launched engines with reusable virtual address space and loading model weights in parallel.

Challenge 2: How to coordinate memory allocation to maximize SLO attainment? Models served by shared GPUs inevitably contend for memory. They differ in request rates, reflecting varying KV cache demands, and in SLO requirements, which indicate how urgently each model must acquire the resources it needs. Without proper coordination, memory contention can severely impact performance and lead to SLO violations, as inference is highly sensitive to available memory. For example, enlarging the KV cache from 5 GB to 15 GB can boost the throughput of a Llama3-8B model by more than 2× on an H100 GPU.

Prism solves this problem through *two-level scheduling*. It first employs a *global scheduler* to place models across GPUs using a heuristic called *KV pressure ratio*, which captures the balance between models' KV cache demands (based on request rates and SLOs) and GPUs' available memory. The scheduler aims to maximize the minimum KV pressure ratio across GPUs, resulting in the most balanced placement. For colocated models on a GPU, Prism uses a *local scheduler* to coordinate memory allocation. It introduces a GPU-level request queue and priority-based admission control to dispatch requests to serving engines. Priorities are derived from each request's SLO, preventing arbitrary memory contention.

[1] This work focuses on memory sharing, as GPU compute can already be flexibly shared in time and space using MPS [35].

Results. We implemented Prism on top of SGLang [64], a widely used LLM serving engine. We evaluated Prism comprehensively with two production traces, and with 58 representative LLMs of varying sizes, on GPU clusters with up to 32 H100 GPUs. Our results show that Prism achieves up to 3.3× higher TTFT SLO attainment and 2× higher TPOT SLO attainment given the same amount of GPUs. When targeting the same level of SLO attainment, Prism achieves up to over 2× cost reduction or 3.5× more requests compared to the state-of-the-art. We will release the prototype of Prism to the community.

2 Background

LLM inference. LLM inference consists of two stages: *prefill* and *decode*. The prefill stage processes the request input (*i.e.*, prompt) and generates the first output token. This is followed by the decode stage, where the LLM generates output tokens auto-regressively—producing one token per iteration using all previously generated tokens as input. Thus, the output length is usually unknown. To improve throughput, LLM serving systems commonly batch multiple requests to process together [58]. To balance the throughput-latency trade-off of batching, chunked-prefill [2] divides prompts into smaller chunks and feeds them to the inference batch incrementally, reducing latency by overlapping prefill with ongoing decode.

KV cache. LLM inference generates a large amount of intermediate data, known as the KV cache. KV cache must remain in GPU memory for the entire inference duration of a request. Its size grows linearly with the request's input and output lengths, and can easily reach tens of gigabytes for long sequences. Thus, inference batch size is directly constrained by the available GPU memory for KV cache.

PagedAttention. To use GPU memory more efficiently, PagedAttention [29] is introduced to manage KV cache. PagedAttention is inspired by the classical virtual memory and paging techniques in operating systems—it breaks the GPU memory into small, fixed-size pages and manages them with a page table. This design allows requests to allocate GPU memory at page granularity, significantly improving memory utilization compared to prior systems [58], which reserve a contiguous memory block for each request's maximum decoding length. PagedAttention has been widely adopted by mainstream serving engines, such as SGLang [64] and vLLM [29].

Inference latency metrics. There are various latency metrics in LLM serving [54]. In this work, we use two common metrics: time-to-first-token (TTFT), which measures the time to generate the first output token, and time-per-output-token (TPOT), which captures the average time of generating a token. TTFT reflects the user-perceived latency in receiving initial responses, while TPOT captures the user experience during token-by-token output generation.

Trace name	Service provider	# models	Time span
Hyperbolic	Hyperbolic [26]	24	4 months
Novita	Novita AI [4]	16	1 month
Arena-Battle	Chatbot Arena [14]	129	16 months
Arena-Chat	Chatbot Arena [14]	84	11 days

Table 2: Production trace summary.

3 Motivation

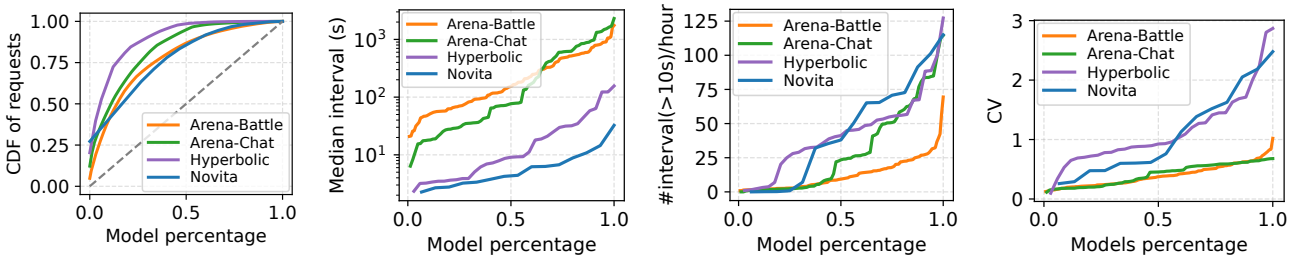
3.1 Workload Implications for GPU Sharing

To understand the workload of multi-LLM serving, we analyze four production traces in detail. These traces are collected from representative service providers as summarized in Table 2. The first two traces are from Hyperbolic [26] and Novita AI [4], two popular LLM inference service providers. They offer inference APIs for a variety of foundation models and also support user-deployed, fine-tuned models. The last two traces are from Chatbot Arena [14], a widely used open-source platform for LLM evaluation. It compares model responses via human preference voting (Arena-Battle) and also provides interfaces for real-time conversations with various models (Arena-Chat). Our analysis reveals four unique workload characteristics, creating both opportunities and challenges for GPU sharing in multi-LLM serving.

(1) Long-tail model popularity. Figure 2a presents the distribution of requests received by each model. We find that all traces exhibit long-tail model popularity—approximately 60% of the models contribute to just 25% of all requests. This trend is due to the wide variety of LLMs available today. Most users choose recently released foundation models, while other models are used only in domain-specific or less common scenarios. As a result, models in the popularity tail are often unable to fully utilize their dedicated GPUs due to their low request rates. For example, the Llama-3.2-3B model in the Hyperbolic [26] trace utilizes only 2.3% of compute and 10.2% of memory on average on an H100 GPU.

Implication #1: Multiple low-demand models can space share GPUs to improve resource utilization.

(2) Frequent idle periods. Models often experience idle periods during which no requests arrive. Figure 2b shows the median idle duration across all models in each trace. In all traces, many models exhibit a median idle duration exceeding 30 seconds. The Chatbot Arena traces show even longer idle durations, with over 50% of models having a median idle time greater than 80 seconds. Figure 2c further reports the average number of idle intervals (>10 seconds) that occur per hour. As shown, idle periods occur frequently—around 50% of models experience more than 40 such intervals per hour in the Hyperbolic and Novita traces. Some models even exhibit over 100 intervals, implying they are idle for at least 27% of the time (assuming each interval lasts 10 seconds).



(a) Long-tail model popularity. (b) Median request interval. (c) Number of intervals per hour. (d) CV of request rate per minute.

Figure 2: A detailed analysis of four production multi-LLM serving traces. These figures show that long-tail model popularity exists in all traces (a), many models have frequent idle period (b)+(c), and the request rate can fluctuate rapidly (d).

Implication #2: The frequent long idle periods allow time sharing by evicting idle models from GPUs and reloading upon new request arrivals.

(3) Rapid workload fluctuations. Figure 1b illustrates the workload fluctuations of three models from the Hyperbolic trace. Here, we present a more detailed quantitative analysis of all traces. Figure 2d shows the distribution of the coefficient of variation (CV) of requests per minute for each model. The CV, calculated as the standard deviation divided by the mean (σ/μ), quantifies the relative variability in request rates—higher values indicate more bursty workloads. Both the Hyperbolic and Novita traces contain many models with a CV greater than 1.0, indicating significant workload fluctuations. The Chatbot Arena traces have lower request rates, resulting in smaller CVs overall, but many models still exhibit CVs greater than 0.5.

Implication #3: No fixed sharing policies work well across all scenarios caused by workload changes. The policy, *e.g.*, which models to colocate and how to allocate resources for them, must be adjusted based on real-time workload.

(4) Diverse latency SLOs. Models have varying latency SLOs (∼1–10+ seconds), determined by their specific use cases and user requirements [40, 43]. For example, coding assistants [53] require low response latency to keep up with a programmer's coding flow, while document summarization [10] can accommodate relatively higher delays. The Novita trace exhibits end-to-end latency SLOs from 5s to 25s.

Implication #4: Latency SLOs reflect how quickly models need to access their required resources. GPU sharing should explicitly consider the SLO diversity to prioritize resource allocation and maximize overall SLO attainment.

Summary. The low resource demands of tail models and the frequent long idle time provide opportunities for GPU space and time sharing to improve utilization. The challenge lies in handling rapid workload fluctuations, which requires dynamically adjusting sharing policies and resource allocation based on real-time workloads and SLO requirements.

3.2 Limitations of Existing Approaches

Existing systems allocate memory statically or use fixed sharing policies, lacking the flexibility to adapt to workload fluctuations at runtime. Thus, they fail to maximize the efficiency of memory usage to meet latency objectives. To illustrate, we evaluated their performance using a workload (Figure 3) that mirrors the request rate patterns of Model1 and Model2 in Figure 1b during 10:00–10:30 (T1) and 13:30–14:00 (T2). In T1, both models have fluctuating loads, with Model1 handling a higher volume of requests. In T2, Model2 experiences a spike in demand, while Model1 becomes idle.

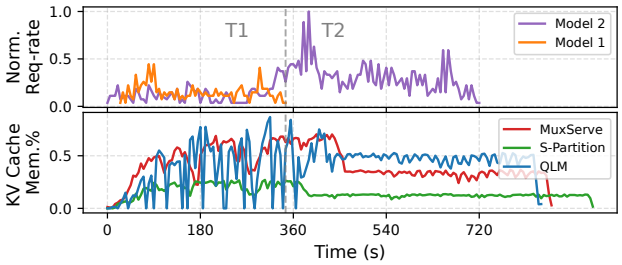


Figure 3: The size of KV cache memory different GPU sharing systems can use on an example workload.

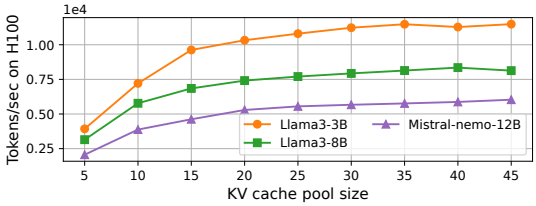


Figure 4: Memory sizes directly impact inference performance.

Figure 3 (bottom) shows the total KV cache size over time. Although both MuxServe [16] and static partition allow GPU space sharing, MuxServe achieves higher KV cache usage during T1 by flexibly sharing memory between models. In contrast, static partition prevents Model1 from leveraging Model2's unused memory. QLM [40], which time shares the GPU through model swapping, shows periodic drops to zero, reflecting the substantial overhead caused by its swapping. During T2, as Model1 becomes idle, QLM dedicates the full GPU to Model2, producing the highest KV cache usage. Meanwhile, MuxServe exhibits lower usage than QLM since it lacks the ability to evict idle models, leaving Model1's weights in GPU memory unnecessarily. KV cache size directly determines the batch size and impacts inference performance. As Figure 4 shows, enlarging the KV cache pool from 5 GB to 15 GB yields over a 2× throughput improvement.

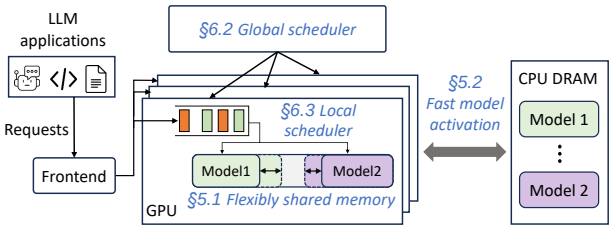


Figure 5: The system architecture and design overview of Prism.

This example highlights the importance of cross-model memory coordination in unlocking the full potential of GPU sharing for cost-efficient multi-LLM serving. Memory must be allocated flexibly to adapt to workload shifts and used wisely to meet performance goals. In this example, during T1, the two models should be colocated and share GPU memory flexibly; during T2, the GPU should be dedicated to Model3 by evicting the idle Model1.

4 Overview

Prism is a multi-LLM serving system that unleashes the full potential of GPU sharing to improve cost efficiency while meeting latency SLOs. Figure 5 shows its system architecture and design overview. Prism receives inference requests at its frontend, which routes them to the corresponding LLMs for processing. To improve cost efficiency, Prism serves LLMs via GPU sharing with flexible combinations of space and time sharing. For instance, high-rate models may occupy GPUs exclusively, while multiple low-rate models can be colocated on a single GPU or GPU group. Idle models are temporarily evicted to CPU DRAM and reactivated when new requests arrive. Prism continuously adjusts its sharing strategies based on runtime workload to maximize overall SLO attainment.

Prism achieves this with two key designs that enable flexible and demand-aware cross-model memory coordination. First, introduces an on-demand memory allocation mechanism (§5.1), enabling models to acquire memory based on actual demand. This flexible mechanism allows Prism to adapt rapidly to workload variations and policy changes, while also facilitating fast model activation (§5.2). Second, Prism optimizes the overall resource allocation through demand-aware cross-model memory coordination. It employs a two-level scheduling strategy to coordinate model placement and memory allocation. At the global level, Prism places models across GPUs based on their memory demands (§6.2). At the GPU level, it uses a shared request queue combined with priority-based admission control to manage memory sharing among colocated models (§6.3).

5 Enabling Flexible GPU Sharing

In this section, we describe how Prism enables flexible memory coordination across models (§5.1) and leverages it for fast model activation (§5.2), forming the foundation for adapting sharing policies to runtime workload changes.

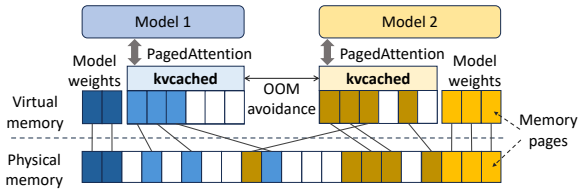


Figure 6: Flexible memory sharing in Prism.

5.1 Cross-Model Memory Coordination

Memory management in today's inference engines is primarily designed for single-model serving. It pre-allocates a large chunk of GPU memory during initialization, including both virtual and physical memory in a fixed 1:1 mapping. This memory is statically reserved for the model and remains unchanged throughout its lifecycle. Although PagedAttention [29] is widely used to dynamically allocate KV cache memory based on request demand, it manages memory at the *application level*, still operating within the statically reserved memory region of a single model. This *per-model static allocation* prevents flexible memory redistribution across models, limiting the ability to adapt GPU sharing policies at runtime. As shown in §3.2, this inflexibility will lead to limited inference performance.

To solve this problem, we need engines to dynamically acquire memory on demand rather than reserve statically. However, implementing this in current engines would require intrusive changes, as their system stacks are designed with static memory allocation in mind. For example, most CUDA kernels are developed with the assumption of contiguous virtual addresses, whereas dynamic memory allocation can lead to non-contiguous address space, necessitating modifications to both kernel implementations and their calling stacks.

kvcached: A shim layer for on-demand cross-model memory allocation. Prism achieves on-demand memory allocation while maintaining compatibility with current engines by designing `kvcached` as a shim layer. It decouples virtual and physical memory allocation using recently introduced CUDA virtual memory management APIs [36]. As shown in Figure 6, in Prism, engines only need to reserve virtual memory during initialization, which is a large chunk of contiguous address space consisting of fixed-size memory pages (*e.g.*, 2 MB). This creates an illusion to engines that all memory is readily available, while in reality, physical memory is allocated and mapped to a virtual page only on demand.

`kvcached` is implemented as a runtime library that can be integrated with current inference engines by changing only *∼20 lines* of code. Table 3 summarizes the provided APIs. Engines can use `alloc_kvcache` or `free_kvcache` APIs to create or destroy a KV cache memory pool. Internally, `kvcached` allocates or frees the corresponding virtual space. Engine's

APIs for inference engines	Corresponding `kvcached` functions
alloc_kvcache (size, shape)	alloc_virtual_tensor (size, shape)
free_kvcache (size, shape)	free_virtual_tensor (size, shape)
alloc_kv (num_tokens)	map (tensor, offset)
free_kv ([ids_to_free])	unmap (tensor, offset)

Table 3: APIs and functions provided by Prism's `kvcached`.

KV manager can dynamically allocate or free the KV cache of a request via `alloc_kv` and `free_kv`. `kvcached` maintains the usage status of KV cache pages. When `alloc_kv` is called, it checks whether the corresponding virtual page has been mapped; if not, it allocates a physical page and maps it accordingly. Similarly, if a page becomes empty on `free_kv`, its physical memory will be unmapped and released.

Prism enables model-independent memory allocation by allowing each engine to link to its own instance of `kvcached`. This is important because CUDA manages GPU memory in fixed-size pages (*i.e.*, 2 MB), while models may use different token sizes (*e.g.*, 16 KB vs. 48 KB). Storing tokens of varying sizes on the same page breaks PagedAttention, as it relies on index-based lookups for efficient token access, which requires uniform token sizes. Moreover, Prism introduces shared system variables across `kvcached` instances to track global memory usage. Combined with proper locking mechanisms, it coordinates memory allocation across engines and help prevent out-of-memory errors caused by race conditions.

Optimizations. Allocating and mapping memory pages on the fly incurs extra latency. Prism reduces this overhead by two optimizations. First, it prioritizes using partially filled pages and placing new tokens in the most utilized page that fits the memory demand. This improves page utilization and reduces the frequency of new allocations. Second, Prism maintains a buffer of pre-allocated and mapped memory pages. When a engine requires new pages, it retrieves them directly from this buffer. The buffer is managed asynchronously, with its overhead fully overlapped with inference computation. These optimizations ensure that `kvcached` has negligible overhead on inference computation.

5.2 Fast Model Activation

Model swapping speed directly impacts the flexibility of GPU sharing. High latency can hinder the timely swapping of models with strict SLOs, limiting policy adaptability. While deactivation is straightforward, *i.e.*, terminating the engine and releasing all memory, reactivation is more complicated, which involves: (1) initializing a new serving engine and reserving a new virtual address space for KV cache pool; and (2) loading the model weights from CPU DRAM. If done naïvely, this process can take tens of seconds—far exceeding the TTFT SLOs of online LLM inference, which are often within just a few seconds or less.

Reusable engine pools. The root cause of (1) lies in the tight coupling between the engine and the model it serves. In current systems, an engine shares the same lifecycle as its model—when a model is evicted, its engine is also terminated, along with its virtual address space. As a result, every model activation incurs the full cost of engine initialization.

Prism eliminates this overhead by decoupling the lifecycles of the engines and models. Specifically, it maintains an engine pool on each GPU, where engines are pre-initialized with virtual address space and distributed contexts. Upon model activation, Prism selects an available engine from the pool and starts model loading directly. When a model is evicted, its physical memory is released, but its engine with virtual address space is returned to the engine pool for future reuse.

However, an engine cannot directly reuse previously reserved virtual memory space to serve a new model. This is because current inference engines perform index-based token access, which depends on a model-specific memory layout that is incompatible with models of different architectures, *e.g.*, different numbers of layers or token sizes. To address this, Prism introduces a KV cache virtual memory manager to manage the pre-reserved virtual memory spaces in engine pool. When a new model is loaded, the manager dynamically aligns the reserved virtual space to match memory layout required by the new model (one-time effort), and then create a new `kvcached` based on the aligned memory spaces. The `kvcached` can then correctly and efficiently locate the virtual memory page where each token resides during inference.

Parallel model weight loading. The time spent on (2) model weight loading is heavily influenced by the utilization of the CPU–GPU interconnect bandwidth. We found that loading models naïvely via the `cudaMemcpyAsync` API to a GPU fails to saturate the interconnect bandwidth, even when invoked from multiple threads. This may be due to all `cudaMemcpyAsync` operations targeting the same GPU executing serially, limited by the CUDA driver and hardware.

Prism overcomes this bottleneck by chunking model weights into smaller segments, loading them in parallel across multiple GPUs on the same node, and then aggregating them to the target GPU via high-speed NVLink interconnects. This parallelized strategy significantly accelerates model loading. To minimize interference with running workloads on GPUs, Prism partitions model weights at the granularity of individual weight tensors and loads them in a streaming fashion. As a result, each GPU only needs to maintain a small buffer (*e.g.*, 30MB), minimizing possible memory contention.

With these optimizations, as we show in our evaluation (Figure 9), Prism can activate an 8B model in 0.7s and a 70B model in 1.5s—4.8× to 7.1× faster than the naïve approach.

6 Two-Level Demand-Aware Scheduling

The techniques proposed in §5 lay the foundation for flexible GPU sharing in multi-LLM serving. However, to fully unleash its benefits, we need to derive effective sharing policies

from real-time workload to maximize the sharing benefits. In this section, we first describe the scheduling problem and then present our two-level demand-aware scheduling algorithm.

6.1 The Scheduling Problem

Models sharing GPUs inevitably contend for memory when resources are constrained, so GPU memory must be used efficiently to ensure performance. This raises a key question: given a fixed number of GPUs, a set of models, and their real-time workloads, how should we coordinate memory allocation across models to satisfy their demands as much as possible?

We prioritize optimizing time-to-first-token (TTFT), as it depends on the prompt length, which is known at the time of request submission. In contrast, time-per-output-token (TPOT) is influenced by the output length, which is unknown in advance due to the auto-regressive nature of LLM decoding. Nonetheless, our system can also benefit TPOT because it satisfies each model's resource demand as much as possible. This reduces the likelihood of request preemption due to insufficient memory, which would otherwise degrade TPOT.

Prism coordinates memory allocation across models using two-level scheduling: (1) global scheduling (§6.2) determines the placement of models across GPUs based on their resource demands, aiming to balance the load and avoid resource bottlenecks; and (2) GPU-local scheduling (§6.3) manages requests from models colocated on the same GPU based on priority, achieving efficient KV cache memory sharing.

6.2 Global Model Placement Scheduling

Prism's global scheduler intelligently places models on available GPUs according to their resource demands, with the goal of balancing the load to minimize resource contention. It performs three key operations: (1) determining model-to-GPU placement, (2) evicting idle models, and (3) activating inactive models upon request arrival.

Model-to-GPU placement. To maximize memory usage efficiency, the scheduler must consider both the request load and SLOs across models. The request load determines the total memory a model requires for serving, while the SLO reflects how urgently a model must acquire these resources to meet its latency goals. This task faces two difficulties in practice. First, the search space is huge (*i.e.*, N^M, assuming N GPUs and M models). Second, accurately estimating the memory demand of a model is difficult, as it depends on output lengths that are not known in advance.

Prism solves this problem using a heuristic we call *KV pressure ratio* (KVPR), which quantifies the degree of KV cache pressure on a GPU. It is calculated as $\frac{w\_req\_rate}{shared\_kv}$, where $w\_req\_rate = \frac{req\_rate}{SLO}$ represents the SLO-weighted request rate of a model, indicating its memory demand per unit time, and *shared_kv* is the memory available for KV cache on a GPU. A higher KVPR indicates higher memory pressure and a more congested GPU.

Algorithm 1 Global Model Placement Scheduling

Require: Number of GPUs N, GPU memory capacity C, migration threshold τ, and M models. Each model m_j has: request rate r_j, weight w_j, current device index g_j, and latency SLO s_j.

Ensure: Assign each model to a GPU to balance the resource demand and remaining memory capacity.

1: Sort models by $\frac{r_j}{s_j}$ in descending order. Denote the sorted sequence as $m_1, m_2, \ldots, m_M$.
2: **for** $i = 1$ to N **do**
3: $\quad shared\_kv_i \leftarrow C$; $w\_req\_rate_i \leftarrow 0$
4: **for** $k = 1$ to M **do**
5: $\quad$ /* find the GPU *best_idx* that minimizes KVPR */
6: $\quad best\_r,\ best\_idx \leftarrow$ (min, argmin) $\frac{w\_req\_rate_i}{shared\_kv_i}$
7: $\quad current\_r \leftarrow \frac{w\_req\_rate_{g_k}}{shared\_kv_{g_k}}$
8: $\quad gpu\_best \leftarrow \begin{cases} best\_idx, & \text{if } current\_r - best\_r > \tau \\ g_k, & \text{otherwise} \end{cases}$
9: $\quad$ Assign model m_k to *best_gpu*
10: $\quad$ /* update states */
11: $\quad w\_req\_rate_{best\_gpu} \leftarrow w\_req\_rate_{best\_gpu} + \frac{r_k}{s_k}$
12: $\quad shared\_kv_{best\_gpu} \leftarrow shared\_kv_{best\_gpu} - w_k$
13: **return** Model-to-GPU placement

Using this heuristic, Prism determines the best model-to-GPU placement through a customized multi-machine scheduling algorithm, as shown in Algorithm 1. It first sorts models by their weighted request rates in descending order and initializes the GPU states (Lines 1–3). Next, for each model, Prism selects the GPU that minimizes KVPR (Lines 5–8). If the selected GPU differs from the model's current GPU, Prism migrates the model to the new GPU. However, this migration incurs overhead from engine switching and model weight loading. To avoid unnecessary migrations with marginal benefit, Prism compares the KVPR of the best and current GPUs, and proceeds only if the improvement exceeds a threshold τ (Line 8). Finally, Prism assigns the model to its selected GPU and updates the corresponding GPU states (Lines 9–12).

Analysis. The global model placement algorithm ensures that the maximum KVPR across all GPUs is bounded by the maximum KVPR in the optimal placement. This algorithm also supports models utilizing tensor parallelism (TP) by treating each TP partition as a separate model and enforcing their placement on distinct GPUs. A more detailed analysis is provided in Appendix A.1.

Model eviction and activation. The global scheduler evicts a model if it remains idle longer than an empirical threshold, which can be determined by analyzing the idle interval distribution. Prism evicts models only when GPU resources are constrained and prioritizes evicting models with larger SLOs. When the evicted model receives new requests, Prism immediately reactivates it by selecting the GPU with the lowest KV pressure ratio to serve it.

6.3 GPU-Local Request Scheduling

Models assigned to the same GPU may compete for KV cache memory when the total memory is limited. Without proper coordination, this contention can lead to inefficient memory usage and degraded SLO attainment. For example, a model with a high request rate but a relaxed SLO may consume a large portion of memory, preventing models with stricter SLOs from obtaining enough memory to use.

A naïve solution is to limit the memory that each model can use. However, determining appropriate limits is challenging due to the dynamic request patterns and different SLOs. Conservative limits may unnecessarily throttle a model's throughput, while overly generous ones can reduce the available memory for other models, as total allocation must remain within the GPU's capacity. An improvement is to adjust these limits dynamically based on runtime workloads. However, such adjustments cannot take effect immediately, as memory must first be released by one model before it can be reallocated to others. This requires waiting for the model to complete some of its ongoing requests—a process that can take seconds to minutes, depending on request length and load.

This memory coordination challenge stems from each engine maintaining its own request queue and greedily scheduling requests as memory becomes available. To address this, Prism introduces a shared GPU-level request queue, coupled with a priority-based admission control mechanism to coordinate memory usage across models. Incoming requests from all models sharing a GPU are first placed into the GPU-level request queue. Prism then dequeues requests based on their priorities and dispatches them to the corresponding engines. Prism greedily dispatches requests up to the point where they can execute immediately without causing queuing within the engines' local waiting queues. This approach balances high memory utilization with strict admission control.

Prism determines the request dequeue order based on the Moore-Hodgson algorithm [32], which minimizes the number of deadline misses. As shown in Algorithm 2, given a set of requests R, Prism first sorts them in ascending order of their prefill completion deadlines, *i.e.*, $a_i + s_i$ for each request r_i, where a_i is its arrival time and s_i is the TTFT SLO. Then, for each request in the sorted list, Prism appends it to the schedule list S and checks whether it can finish within its TTFT SLO. Specifically, it checks whether $current\_time + e_r \leq a_r + s_r$, where e_r denotes the prefill time of request r. e_r can be calculated as $e_r = \frac{p_r}{c_r}$, with p_r being the prompt length and c_r the chunked prefill speed determined by the model's chunk size configuration. If the most recently added request cannot meet its deadline, Prism evicts the request in S with the longest execution time (Lines 9–11). It then moves to the next request and continues this process until evaluating all requests. Finally, Prism dispatches the accepted requests in S following their order in the schedule.

Algorithm 2 GPU-Local Request Scheduling

Require: A set of n requests $R = \{r_1, r_2, \ldots, r_n\}$. Each request r_i has: a prompt length p_i, a chunked-prefill speed c_i determined by the model serves it, a TTFT SLO s_i, and an arrival time a_i.
Ensure: A subset of requests $S \subseteq R$ that can be executed in order to maximize TTFT SLO attainment.
1: Sort R in ascending order of deadlines $d_i = a_i + s_i$: $r_1, r_2, \ldots, r_n$ such that $d_1 \leq d_2 \leq \cdots \leq d_n$.
2: Initialize $S \leftarrow \emptyset$, $current\_time \leftarrow$ Timer.time()
3: **for** $k = 1$ to n **do**
4: Let $r \leftarrow r_k$, $e_r \leftarrow \frac{p_r}{c_r}$
5: Append r to S
6: Update $current\_time \leftarrow current\_time + e_r$.
7: **if** $current\_time > a_r + s_r$ **then**
8: /* pop the request with longest execution time */
9: Let $r_{\max} \leftarrow \arg\max_{r' \in S} \frac{p_{r'}}{c_{r'}}$
10: Remove $r_{\max}$ from S
11: Update $current\_time \leftarrow current\_time - \frac{p_{r_{\max}}}{c_{r_{\max}}}$
12: **return** S

Analysis. When chunked-prefill has prefill running at each inference step, the request scheduling ensures optimal TTFT attainment. This is because, in this case, the prefill time e_r of a request r can be estimated as $e_r = \frac{p_r}{c_r}$, allowing us to computes the prefill completion time of any request r_i in a sequence using $d_{r_i} = a_{r_i} + \sum_{i=1}^{n} \frac{p_{r_i}}{c_{r_i}}$, where n is the number of requests (including r_i) waiting for processing. With this information, we can follow the proof of the original Moore-Hodgson algorithm [13] to prove the optimality of our scheduling algorithm. The assumption that prefill runs at each inference step holds in most cases. However, in rare situations where the KV cache is insufficient to admit new requests, prefill may be temporarily paused and some running requests will be preempted. Our admission control alleviates this by admitting a proper number of requests and reserving a memory buffer per engine to ensure enough space to complete part of the request batch.

7 Implementation and Evaluation

7.1 Implementation

We implemented a prototype of Prism with ∼10,400 lines of Python and 774 lines of C++ code. As the serving backend, we used SGLang [64], a widely adopted open-source inference engine, and extended it with our `kvcached` library to support on-demand memory allocation. `kvcached` is implemented using CUDA VMM APIs [36] and provides standard KV cache allocation APIs (Table 3) accessible through Python bindings. These APIs are designed to be agnostic to attention mechanisms and architectural differences across inference engines, enabling seamless integration—we modified only 22 lines of code in SGLang. For compute sharing, we configured the MPS percentage to 100% per model, allowing models to time and space share the GPU compute resources.

Model size	1B-3B	4B-8B	9B-30B	31B-70B
#LLMs	43	8	3	4

Table 4: Models used in our evaluation.

On the frontend, we used a Redis queue [45] to cache incoming requests from all clients. Prism's local scheduler dispatches these requests to the serving engines of corresponding models based on Algorithm 2. For tensor-parallel models, the GPU-local scheduler runs only on the first rank, and the resulting scheduling decisions are broadcast to all other ranks to ensure consistency. Prism's global scheduler operates as a separate Python process, collecting execution metrics from each engine—such as request rates and queue status. It makes scheduling decisions (*e.g.*, model evictions and activations) and communicates them to the engines using ZeroMQ [59].

7.2 Experimental Setup

Testbed. We conducted our experiments on a cluster of four nodes, each equipped with eight NVIDIA H100-80G GPUs interconnected via 600GB/s NVLink. These nodes communicate through an 100Gbps Ethernet network. Each node features two 52-core Intel Xeon Platinum 8480+ CPUs, 1.7 TB of DRAM, and a PCIe Gen5 x16 interface, which provides up to 64 GB/s of unidirectional bandwidth per GPU connection. All nodes run Ubuntu 22.04 with CUDA Toolkit 12.4.

Baselines. We compared Prism against three baselines: (1) *Static partition (S-Partition)*, (2) *MuxServe++*, and (3) *QLM [40]*. Note that the original MuxServe [16] is built on vLLM and supports only Llama-2 models. We ported it to SGLang and extended it with our on-demand memory allocation mechanism to support a wider range of models, referred as MuxServe++. We carefully tuned MuxServe++ to ensure it achieves performance comparable to or better than the original version (Calibration data in Table 5).

Traces and models. We used two real-world traces, Hyperbolic [26] and Arena-Chat [48], that are previously analyzed in §3.1. For each trace, we sampled a representative set of models, including both popular and long-tail models, ensuring their workload characteristics (*e.g.*, popularity distribution and idle patterns) align with the observations in §3.1. To simulate a variety of scenarios, we scaled the traces by multiplying the number of requests by a factor of N, increasing the load while preserving the original traffic patterns—a method commonly used in prior work [16, 30]. In total, we evaluated 58 LLMs as detailed in Table 4. The large-scale experiments (§7.5) use the full model set, while other evaluations (§7.3—§7.4) select subsets tailored to specific goals.

Metrics. Our primary performance metrics are TTFT and TPOT attainment. To establish SLOs for each model, we first ran its workload on dedicated GPUs to measure its P95 TTFT and TPOT latencies. This process produced TTFT SLOs ranging from 0.04s to 0.13s and TPOT SLOs from 5.2ms to 50.9ms. We then scaled these baseline values by a factor to evaluate system performance under varying latency requirements, following an approach consistent with prior work [16, 30, 43]. We also reported aggregated throughput. To account for model idle periods, throughput considers the actual time (excluding idle time) spent serving them.

7.3 End-to-End Performance

We first evaluate the end-to-end performance of our system under varying request rates, SLO requirements, and numbers of GPUs, on both Hyperbolic and Arena-Chat traces.

SLO attainment vs. request rate. We first evaluated Prism's performance on various inference load by serving eight model on two shared GPUs. As shown in Figure 7 (first row), Prism consistently outperforms the baselines by maintaining significantly higher TTFT SLO attainment. On the Hyperbolic trace, Prism supports up to 2.3× and 3.5× more requests than MuxServe++ and static partitioning, respectively, while still achieving 99% SLO attainment. On the Arena-Chat trace, it handles over 3× more requests than all baselines. This improvement stems from Prism's ability of flexibly adjusting sharing policies according to the real-time workload, ensuring that more models can acquire their needed resources to achieve their SLO requirements.

MuxServe++'s TTFT SLO attainment drops quickly as the request rate increases because it cannot evict idle models or relocate models across GPUs to balance resource demands. As the load grows, MuxServe++ experiences increasing memory contention, leading to degraded performance. QLM consistently shows low TTFT SLO attainment, even lower than static partition, because it forces all models to time-share GPUs through swapping, which incurs significant overhead and causes severe SLO violations.

Prism also maintains high TPOT attainment, thanks to its demand-aware scheduler that can dynamically balance workloads to minimize memory contention. In contrast, QLM exhibits lower TPOT attainment, as it tends to over-batch requests under high load, leading to extended execution times per iteration and increased TPOT values. Both MuxServe++ and static partition suffer from significant memory contention at high request rates, leading to frequent request preemptions that significantly degrade TPOT attainment.

SLO attainment vs. SLO requirements. The middle row of Figure 7 illustrates the attainment when we set different SLO targets. As SLO scales up, Prism quickly achieves 99% TTFT and TPOT attainment. Specifically, it reaches 99% TTFT and TPOT attainment at SLO scales of 20 and 22 on the Hyperbolic trace, and 10 and 3 on the Arena-Chat trace. In contrast, no baseline achieves 99% TTFT attainment even at the largest SLO scale in this experiment, and their attainment rates do not significantly improve as the scale increases. Among the baselines, MuxServe++ achieves the best TTFT performance, reaching 84.79% and 67.22% attainment at the

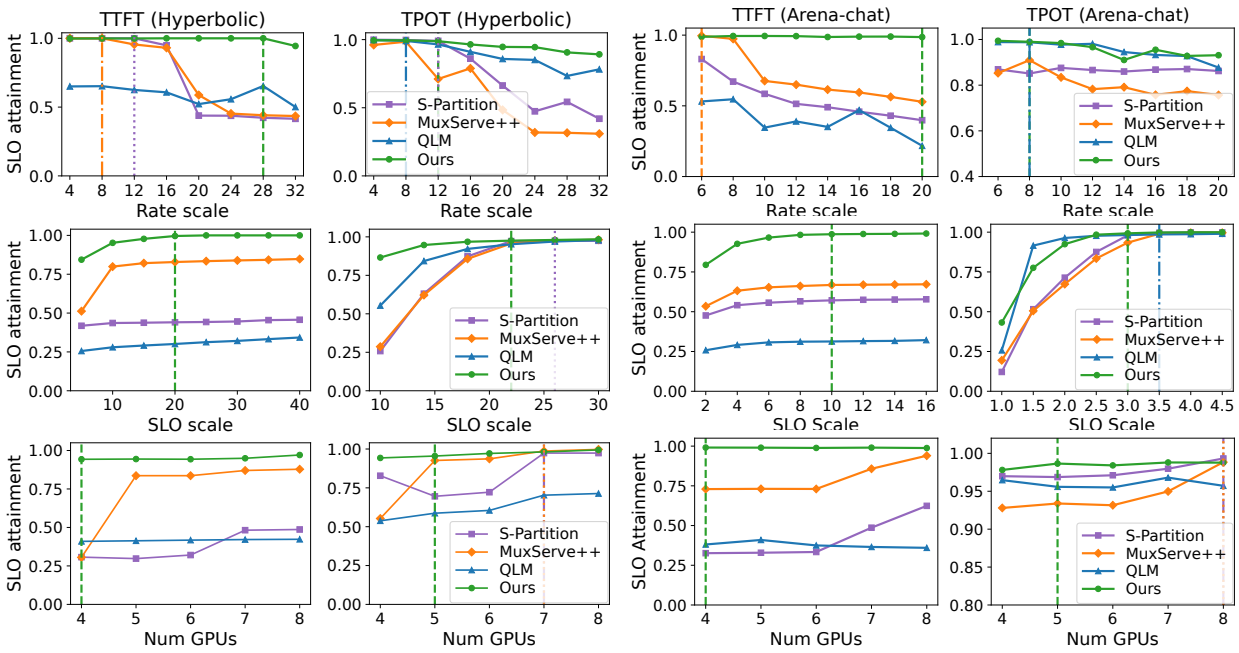


Figure 7: End-to-end performance comparison on SLO attainments under varing scales of rates, SLOs, and number of available GPUs. The dotted vertical lines mark where the system reaches 99% TTFT or TPOT attainment.

largest SLO scale. This gap is primarily from their inflexibility of adapting sharing policies to dynamic workloads. The TPOT attainment of all systems increases rapidly as the SLO scale grows. Static partitioning achieves 99% attainment on Hyperbolic trace (scale=26), while QLM reaches 99% on Arena-Chat trace (scale=3.5). This is because TPOT is less sensitive to memory contention compared to TTFT. We also observe that the Hyperbolic trace requires higher SLO scales due to its more bursty and heavier request patterns.

SLO attainment vs. available GPUs. Finally, we evaluate performance when provisioning more GPUs. We selected 18 models from Table 4, representing a mix of popular and tail models with diverse load variability. To fully test the flexibility of our scheduling strategy, we included models of varying sizes from 1B to 8B, all of which fit within a single 80GB GPU. This setup enables a wide range of model-sharing combinations across GPUs. As shown in Figure 7 (last row), Prism achieves 99% TTFT and TPOT attainment using only four and five GPUs on the two traces, respectively, demonstrating its effectiveness in improving cost-efficiency while maintaining performance. In comparison, all baselines fail to reach 99% TTFT attainment even with eight GPUs, and only a few baselines achieve 99% TPOT attainment when seven or eight GPUs are provisioned.

7.4 Performance Analysis

Next, we provide a detailed performance breakdown to analyze the effectiveness of each design in Prism and how they contribute to its strong end-to-end performance.

Flexible cross-model memory coordination. We first evaluated the benefits of our flexible cross-model memory coordination by comparing with static partition using a simplified two-model trace extracted from Arena-Chat, shown in Figure 8 (first row). We present the normalized total KV cache usage and aggregated throughput for both methods in the last two rows in Figure 8. As we can see, Prism's on-demand memory allocation allows it to use more memory for KV cache, particularly after the 20th second, when Model1 experiences low demand while Model2 faces a surge in request rates. The larger KV cache memory enables Prism to achieve higher throughput, as shown in the last row. In contrast, under static partitioning, even when Model1 underutilizes its memory, Model2 cannot leverage the unused memory due to the static allocation boundary. This shows that Prism's on-demand memory allocation significantly improves memory efficiency and, in turn, enhances performance in GPU sharing.

Model activation speed. Then, we measured how fast can Prism activate models. We measured the activation latency from pageable CPU memory for models ranging from 1B to 70B parameters and summarized the results in Figure 9. As shown, each optimization significantly reduces activation latency. With all optimizations enabled, Prism loads small models from 1B to 8B within 0.7s, and a medium-size 14B model in just 1.3s, which is 5.5× faster than the baseline that naïvely uses `cudaMemcpyAsync` and takes 7.1s. Larger models are usually served using TP; for a 70B model that is served using TP=8, Prism can activate it in 1.5s. Since a TP model is already sharded and loaded in parallel among different TP ranks, it can achieve relatively good loading

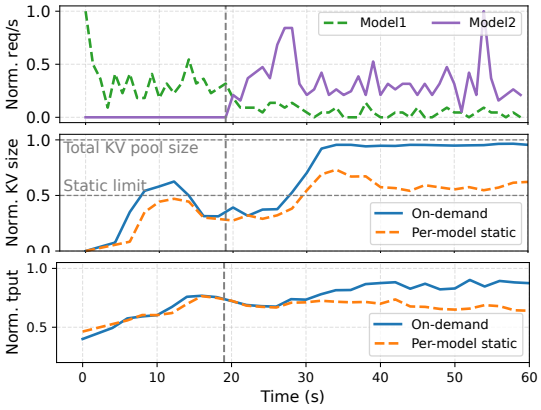


Figure 8: The benefits of cross-model memory coordination. The first figure shows the request rates. The last two figures shows the total KV memory size and the throughout of the two models.

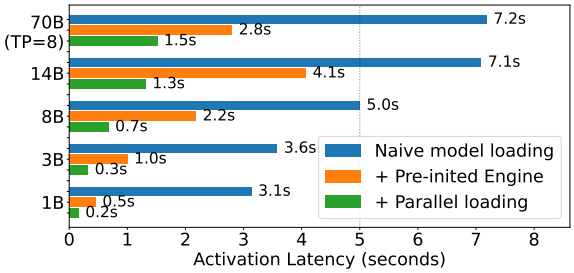
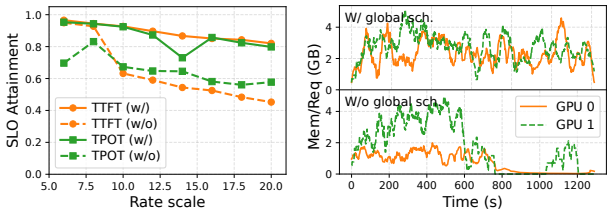


Figure 9: The activation time for models with different sizes. Data is measured on H100 GPUs.

performance even without our parallel loading optimization. These results show that Prism can prompt activate evicted models upon receiving new requests, helping reduce SLO violations.

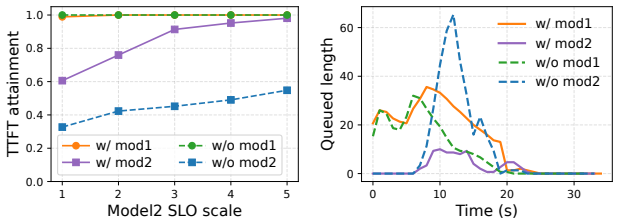
Global scheduling. Next, we evaluated the benefits of our global scheduler. In this experiment, we used two GPUs to serve eight models. Figure 10a presents the TTFT and TPOT attainment with the global scheduler enabled or disabled. The results show that enabling the global scheduler significantly improves both TTFT and TPOT attainment. To provide further insights, we plot the average KV cache memory available per request as it arrives on each GPU. With the global scheduler enabled, the load is more evenly balanced across the two GPUs, allowing each request to access more KV cache memory on average. In contrast, without the global scheduler, the load is imbalanced: GPU1 shows more available memory during the first 600 seconds, while the near-zero availability between the 800th and 1000th seconds indicates that GPU1 is idle while GPU0 is overloaded. These results demonstrate the global scheduler's ability to coordinate resource demands across GPUs, avoid bottlenecks, and ultimately improve performance in GPU sharing.

GPU local scheduling. We also evaluated the benefits of GPU-local scheduling in coordinating memory between models sharing the same GPU. Here, we use two models: we fix the SLO scale of Model1 to eight and vary the SLO scales of



(a) Attainment with rate scales (b) GPU load status

Figure 10: The effectiveness of global model placement scheduling.



(a) Attainment with SLO scales (b) Queue length with time

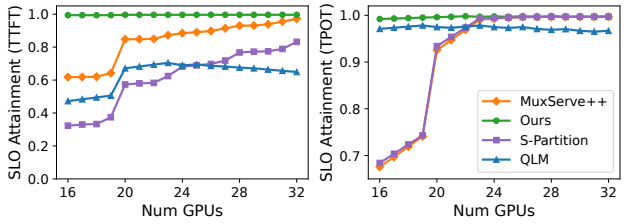
Figure 11: The effectiveness of GPU local request scheduling.

Model2 to evaluate the priority-based admission control in the GPU-local scheduling. Figure 11a shows the TTFT attainment as we vary the SLO scale of Model2. Model1 consistently maintains high attainment, and enabling our GPU-local scheduling improves the SLO attainment of Model2 by more than 40%. To dive deeper, we plot the queue length of each model in Figure 11b of one experiment run. From the queue lengths, we clearly observe that when the local scheduler is enabled, the system prioritizes Model2's requests, which are shorter but have stricter SLO requirements. Specifically, between the 10th and 20th seconds, Model2's queue length is noticeably lower when local scheduler is enabled. This shows the effectiveness of GPU-local scheduling in coordinating memory allocation across models based on their SLOs.

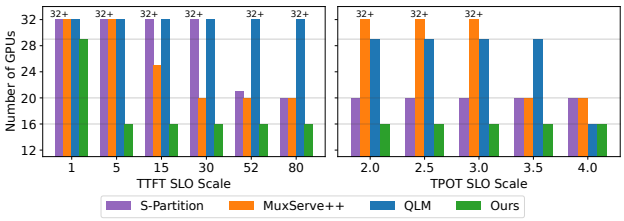
7.5 Large-Scale Deployment

Finally, we evaluated how effectively Prism in reducing the cost of multi-LLM serving at scale. We served all 58 models listed in Table 4, following common TP practices for large models: TP=4 for 32B models [51, 52] and TP=4 or 8 for 70B models [3, 31], utilizing 32 GPU in total. We sampled 58 models from the Arena-Chat trace. The Hyperbolic trace includes only 24 models, so we generated additional traces by sampling different time periods from the same models, creating a larger and more representative workload.

SLO attainments vs. number of GPUs. Figure 12a shows TTFT and TPOT attainment with increased number of GPUs. Prism consistently outperforms all baselines, achieving nearly 99% TTFT attainment with just 16 GPUs, while MuxServe++ requires 32 GPUs to reach similar performance, and other methods require even more. As the number of GPUs increases, both static partition and MuxServe++ improve in TTFT and TPOT attainment, as fewer models need to share a GPU on



(a) SLO attainment with cluster sizes scales



(b) Number of GPUs needed for 99% SLO attainments

Figure 12: SLO attainment and cost saving at large scales.

average. However, QLM fails to achieve better performance. We find this to be related to its suboptimal scheduling algorithm. QLM assigns incoming request groups to GPUs without considering which models are already on GPUs. If it detects the queue is empty, the scheduler simply selects the first available GPU, often triggering unnecessary model swaps. With more GPUs, requests are drained faster, resulting in more idle GPUs and higher likelihood of swapping. Its unoptimized swapping mechanism, which forces the inference engine to stop and restart with a different model [44], incurs significant latency overhead, causing subsequent queued requests to miss their SLOs.

Cost saving. Figure 12b shows the number of GPUs required by each system to achieve 99% SLO attainments at different SLO scales. If a system fails to achieve 99% attainment with all 32 GPUs, we denote its GPU requirement as "32+". Prism achieves 99% TTFT SLO attainments with only 16 GPUs when SLO scale is 5 and TPOT SLO scale is 2.0. MuxServe++ needs 20 GPUs to get 99% TTFT SLO attainments with SLO scale ≥ 30, while static partition needs even more GPUs or higher SLO scale. For TPOT, static partition is the best across all baseline, requiring 20 GPUs, while QLM and MuxServe++ need at least 29 GPUs when TPOT SLO scale ≤ 3.0.

8 Discussion

Model activation time. With the proposed optimizations (§5.2), Prism can activate an 8B model in 0.7s and a 70B model in 1.5s, which is acceptable for many serving scenarios. For scenarios with very strict TTFT requirements, Prism can optionally disable model eviction by trading off some cost efficiency. Notably, next-generation GPU hardware, such as the NVIDIA GH200 [37], provides a CPU-GPU interconnect bandwidth of 900 GB/s, which will further reduce the model activation time. Currently, we store all model weights in CPU DRAM. In scenarios where models need to be loaded from SSD disks, Prism can apply optimizations such as pipelined and parallel weight loading, as explored in recent work [19, 60]; we leave it to future work.

GPU compute sharing. We use MPS with a 100% SM utilization limit per model to enable GPU compute sharing. Under this configuration, MPS allows models to share GPU SMs in time and opportunistically in space. This approach already works well in practice, given the low compute utilization relative to memory contention. In the future, we plan to incorporate better compute resource coordination into our design by dynamically adjusting SM limits, leveraging recently introduced techniques such as green context [34].

Reducing serving costs using heterogeneous GPUs. Some work proposes reducing LLM serving costs by leveraging heterogeneous GPUs, *e.g.*, using low-end GPUs to serve smaller models [21, 28, 63]. We argue that this is an orthogonal direction that does not fully address the core problem. First, serving LLMs on heterogeneous GPUs has not seen wide adoption in practice, as it introduces new challenges, such as managing heterogeneous clusters. Second, due to workload variability, systems must still provision for peak demand, resulting in the same resource underutilization problem as in homogeneous clusters. Moreover, Prism is compatible with heterogeneous GPU clusters and can be deployed to improve resource utilization in such settings as well.

9 Related Work

Systems serving multiple LLMs. Beyond MuxServe [16] and QLM [40], recent systems including ServerlessLLM [18], DEEPFLOW [24], ENOVA [25] and BLITZSCALE [60] aim to serve LLMs in a serverless style, with techniques such as optimized checkpoint formats [18], DRAM preloading [24], and fine-grained autoscaling [60] to reduce cold start latency. In contrast, Prism focuses on efficient GPU sharing during runtime, enabling dynamic memory coordination across models. In addition, SamuLLM [17] enhances end-to-end efficiency for multi-LLM applications in offline settings, whereas Prism is designed for online inference. AlpaServe [30] also explores GPU multiplexing for serving multiple models, but it does not consider auto-regressive models, which introduce new memory coordination challenges that Prism is specifically designed to handle.

SLO-aware LLM scheduling. Recent work has explored SLO-aware scheduling to improve LLM inference performance, including Llumnix [50], SLOs-Serve [11], ExeGPT [39], SAGESERVE [27], and MELL [42]. These systems primarily focus on request scheduling or resource allocation for single-model serving. In contrast, Prism addresses dynamic memory allocation and coordination across multiple models to support efficient multi-LLM serving.

Memory management in LLM serving. vAttention [41] and vTensor [57] also leverage CUDA virtual memory APIs [36]

to decouple physical and virtual memory allocation. However, their purpose is to reduce programming complexity and improve kernel efficiency for serving a single model. In addition, their approaches need to reimplement the entire stack of current inference engines, while our method preserves compatibility with the widely used PagedAttention [29] mechanism.

GPU sharing techniques. In addition to native GPU sharing mechanisms (*i.e.*, MPS [38] and MIG [38]) and widely adopted fractional GPU techniques [1, 7, 46], recent research has proposed more advanced GPU sharing strategies [22, 23, 33, 49, 61, 62]. However, these techniques primarily focus on compute resource sharing and lack support for memory coordination, which is a central challenge in multi-LLM serving due to the memory-intensive, auto-regressive nature of generation. Another line of work, such as S-LoRA [47] and dLoRA [56], adderess scenarios involving a single large base model with many lightweight adapters. In contrast, Prism targets a broader setting by enabling serving of multiple large base models cost efficiently.

10 Conclusion

Serving LLMs at scale is expensive, particularly for providers hosting many models with dynamic and bursty workloads. This paper introduces Prism, a multi-LLM serving system that improves cost efficiency while maintaining SLO attainment by fully unleashing the power of GPU sharing. Prism achieves this by first enabling flexible cross-model memory coordination, which allows it to dynamically adjust GPU sharing policies based on runtime workloads. It then employs a two-level scheduling algorithm to make efficient use of GPU memory to avoid resource bottlenecks and maximize latency performance. Evaluations on real-world traces show that Prism delivers more than 2× cost savings and 3.3× improvement in SLO attainment over state-of-the-art systems.

Acknowledgement

We thank Professor Matei Zaharia for insightful discussions. We are grateful to Hyperbolic, Novita AI, and Chatbot Arena for providing production traces that informed our key findings, and Hyperbolic, Novita AI, and NVIDIA for providing hardware support. Special thanks to the SGLang open-source community for discussions and hardware resources. We thank Jiangfei Duan for insights on MuxServe. We also thank Chenxi Wang, Shi Liu, Zhenting Zhu, Yicheng Liu for their valueable feedback. This work is supported by NSF grants CNS-1763172, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2147909, CNS-2128653, CNS-2301343, CNS-2330831, CNS-2403254. Sky Computing Lab is supported by gifts from Accenture, AMD, Anyscale, Cisco, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, Microsoft, NVIDIA, Samsung SDS, and SAP.

References

[1] N. Agarwal. Implementing Fractional GPUs in Kubernetes with Aliyun Scheduler. https://huggingface.co/blog/NileshInfer/implementing-fractional-gpus-in-kubernetes, 2024. Accessed: May, 2025.

[2] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[3] M. AI. Llama 3.3-70B-Instruct. https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct, 2024. Accessed: May, 2025.

[4] N. AI. Novita AI homepage. https://novita.ai/, 2025. Accessed: May, 2025.

[5] T. AI. Together AI Fine-tuning Guide. https://docs.together.ai/docs/fine-tuning-quickstart, 2025. Accessed: May, 2025.

[6] T. AI. Together AI homepage. https://www.together.ai/, 2025. Accessed: May, 2025.

[7] Ailiyun. GPU Sharing Scheduler for Kubernetes Cluster. https://github.com/AliyunContainerService/gpushare-scheduler-extender, 2023. Accessed: May, 2025.

[8] Amazon. Amazon Bedrock User Guide. https://docs.aws.amazon.com/bedrock/latest/userguide/what-is-bedrock.html, 2025. Accessed: May, 2025.

[9] Amazon. AWS AI Pricing. https://aws.amazon.com/sagemaker-ai/pricing/, 2025. Accessed: May, 2025.

[10] Anthropic. Anthropic Legal Summarization Guide. https://docs.anthropic.com/en/docs/about-claude/use-case-guides/legal-summarization, 2025. Accessed: May, 2025.

[11] S. Chen, Z. Jia, S. Khan, A. Krishnamurthy, and P. B. Gibbons. SLOs-Serve: Optimized Serving of Multi-SLO LLMs. *arXiv preprint arXiv:2504.08784*, 2025.

[12] K. Cheng, Z. Wang, W. Hu, T. Yang, J. Li, and S. Zhang. Towards SLO-Optimized LLM Serving via Automatic Inference Engine Tuning. *arXiv preprint arXiv:2408.04323*, 2024.

[13] J. Cheriyan, R. Ravi, and M. Skutella. A Simple Proof of the Moore-Hodgson Algorithm for Minimizing the Number of Late Jobs. *Operations Research Letters*, 49(6):842–843, 2021.

[14] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference, 2024.

[15] G. Cloud. Model Garden on Vertex AI. https://cloud.google.com/model-garden?hl=en, 2025. Accessed: May, 2025.

[16] J. Duan, R. Lu, H. Duanmu, X. Li, X. Zhang, D. Lin, I. Stoica, and H. Zhang. MuxServe: Flexible Multiplexing for Efficient Multiple LLM Serving. *arXiv preprint arXiv:2404.02015*, 2024.

[17] J. Fang, Y. Shen, Y. Wang, and L. Chen. Improving the End-to-End Efficiency of Offline Inference for Multi-LLM Applications Based on Sampling and Simulation. *arXiv preprint arXiv:2503.16893*, 2025.

[18] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai. ServerlessLLM:Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[19] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo. Cost-Efficient Large Language Model Serving for Multi-Turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, 2024.

[20] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[21] T. Griggs, X. Liu, J. Yu, D. Kim, W.-L. Chiang, A. Cheung, and I. Stoica. Mélange: Cost Efficient Large Language Model Serving by Exploiting GPU Heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[22] L. Han, Z. Zhou, and Z. Li. Pantheon: Preemptible Multi-DNN Inference on Mobile Edge GPUs. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, pages 465–478, 2024.

[23] M. Han, H. Zhang, R. Chen, and H. Chen. Microsecond-Scale Preemption for Concurrent GPU-Accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[24] J. Hu, J. Xu, Z. Liu, Y. He, Y. Chen, H. Xu, J. Liu, B. Zhang, S. Wan, G. Dan, Z. Dong, Z. Ren, J. Meng, C. He, C. Liu, T. Xie, D. Lin, Q. Zhang, Y. Yu, H. Feng, X. Chen, and Y. Shan. DeepFlow: Serverless Large Language Model Serving at Scale, 2025.

[25] T. Huang, P. Chen, K. Gong, J. Hawk, Z. Bright, W. Xie, K. Huang, and Z. Ji. Enova: Autoscaling towards cost-effective and stable serverless llm serving, 2024.

[26] Hyperbolic. Hyperbolic: The Open Access AI Cloud. Hyperbolic Provides Affordable GPU Access and Inference Services for Those at the Edges of AI. https://hyperbolic.xyz/, 2024.

[27] S. Jaiswal, K. Jain, Y. Simmhan, A. Parayil, A. Mallick, R. Wang, R. S. Amant, C. Bansal, V. Rühle, A. Kulkarni, et al. Serving Models, Fast and Slow: Optimizing Heterogeneous LLM Inferencing Workloads at Scale. *arXiv preprint arXiv:2502.14617*, 2025.

[28] Y. Jiang, F. Fu, X. Yao, G. He, X. Miao, A. Klimovic, B. Cui, B. Yuan, and E. Yoneki. Demystifying Cost-Efficiency in LLM Serving over Heterogeneous GPUs. *arXiv preprint arXiv:2502.00722*, 2025.

[29] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023*, pages 611–626. ACM, 2023.

[30] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, and I. Stoica. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[31] Meta. Introducing Llama 3.1: Our Most Capable Models to Date. https://ai.meta.com/blog/meta-llama-3-1/, 2024.

[32] J. M. Moore. An N Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, 1968.

[33] K. K. Ng, H. M. Demoulin, and V. Liu. Paella: Low-Latency Model Serving with Software-Defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 595–610, 2023.

[34] NVIDIA. CUDA Toolkit Documentation—Green Contexts. https://docs.nvidia.com/cuda/cuda-driver-api/

group__CUDA__GREEN__CONTEXTS.html. Accessed: May, 2025.

[35] NVIDIA. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2024. Accessed: May, 2025.

[36] NVIDIA. CUDA Toolkit Documentation: Virtual Memory Management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__VA.html, 2025. Accessed: May, 2025.

[37] NVIDIA. NVIDIA Grace Hopper Superchip, 2025. Accessed: May, 2025.

[38] NVIDIA. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2025. Accessed: May, 2025.

[39] H. Oh, K. Kim, J. Kim, S. Kim, J. Lee, D.-s. Chang, and J. Seo. Exegpt: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 369–384, 2024.

[40] A. Patke, D. Reddy, S. Jha, H. Qiu, C. Pinto, C. Narayanaswami, Z. Kalbarczyk, and R. Iyer. Queue Management for SLO-Oriented Large Language Model Serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 18–35, New York, NY, USA, 2024. Association for Computing Machinery.

[41] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention, 2024.

[42] L. Qianli, H. Zicong, C. Fahao, L. Peng, and G. Song. Mell: Memory-Efficient Large Language Model Serving via Multi-GPU KV Cache Management. *arXiv preprint arXiv:2501.06709*, 2025.

[43] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu. Mooncake: Trading More Storage for Less Computation—A KVCache-Centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, 2025.

[44] QLM Project. QLM: Quantum Language Model Project. https://github.com/QLM-project/QLM/blob/eea5b622e2c4c6abd705876880f50014c4d9d0d1/qlm/endpoints/endpoint.py#L57, 2024. Accessed: May, 2025.

[45] Redis. Redis List documentation. https://redis.io/docs/latest/develop/data-types/lists/, 2025. Accessed: May, 2025.

[46] Run:ai. Quickstart: Launch Workloads with GPU Fractions. https://docs.run.ai/v2.17/Researcher/Walkthroughs/walkthrough-fractions/, 2023. Accessed: May, 2025.

[47] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, et al. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. *Proceedings of Machine Learning and Systems*, 2024.

[48] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA, July 2024. USENIX Association.

[49] F. Strati, X. Ma, and A. Klimovic. Orion: Interference-Aware, Fine-Grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.

[50] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, 2024.

[51] Q. Team. Qwen2.5-32b. https://huggingface.co/Qwen/Qwen2.5-32B, 2024. Accessed: May, 2025.

[52] Q. Team. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/, 2024. Accessed: May, 2025.

[53] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741*, 2024.

[54] Z. Wang, S. Li, Y. Zhou, X. Li, R. Gu, N. Cam-Tu, C. Tian, and S. Zhong. Revisiting SLO and Goodput Metrics in LLM Serving. *arXiv preprint arXiv:2410.14257*, 2024.

[55] K. Wiggers. OpenAI Launches Flex Processing for Cheaper, Slower AI Tasks. https://techcrunch.com/2025/04/17/openai-launches-flex-processing-for-cheaper-slower-ai-tasks/, 2025. Accessed: May, 2025.

[56] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin. dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, 2024.

[57] J. Xu, R. Zhang, C. Guo, W. Hu, Z. Liu, F. Wu, Y. Feng, S. Sun, C. Shao, Y. Guo, et al. vTensor: Flexible Virtual Tensor Management for Efficient LLM Serving. *arXiv preprint arXiv:2407.15309*, 2024.

[58] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. 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.

[59] ZeroMQ. ZeroMQ Website. https://zeromq.org/, 2025. Accessed: May, 2025.

[60] D. Zhang, H. Wang, Y. Liu, X. Wei, Y. Shan, R. Chen, and H. Chen. Fast and Live Model Auto Scaling with O (1) Host Caching. *arXiv preprint arXiv:2412.17246*, 2024.

[61] S. Zhang, Q. Chen, W. Cui, H. Zhao, C. Xue, Z. Zheng, W. Lin, and M. Guo. Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 573–588, 2025.

[62] Y. Zhang, H. Yu, C. Han, C. Wang, B. Lu, Y. Li, Z. Jiang, Y. Li, X. Chu, and H. Li. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 267–281, 2025.

[63] J. Zhao, B. Wan, Y. Peng, H. Lin, and C. Wu. LLM-PQ: Serving LLM on Heterogeneous Clusters with Phase-Aware Partition and Adaptive Quantization. *arXiv preprint arXiv:2403.01136*, 2024.

[64] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng. Efficiently Programming Large Language Models using SGLang, 2023.

A Appendix

A.1 Analysis of Algorithm 1

A.1.1 KVPR Bound Analysis

The global model placement algorithm ensures that the maximum KV pressure ratio (KVPR) across all GPUs is bounded by the maximum KVPR in the optimal placement. We give the following analysis.

Let $KVPR_{OPT}$ be the minimum possible maximum KVPR achievable by any optimal placement. Let $KVPR_{max}$ be the maximum KVPR produced by Algorithm 1. We want to show $KVPR_{max}$ is bounded by $KVPR_{OPT}$.

Bottleneck Analysis: Focus on the GPU, denoted as g_{max}, that achieves the highest KVPR ($KVPR_{max}$) given by Algorithm 1's placement. Let m_k be the *last* model assigned to this GPU g_{max}. Let W_{before} and S_{before} represent the total SLO-weighted request rate and shared KV memory on g_{max} *just before* model m_k was assigned. The final state on this GPU is $KVPR_{max} = (W_{before} + d_k)/(S_{before} - w_k)$, where d_k is the SLO-weighted request rate (r_k/s_k) and w_k is the memory weight of model m_k. Similar to Graham [20], this proof aims to demonstrate that both the state *before* m_k was added and the contribution *of* m_k are bounded relative to $KVPR_{OPT}$. Specifically, it seeks to establish two conceptual bounds:

- **Bound 1 (Related to state before m_k):** The KVPR on g_{max} just before m_k's assignment, $\frac{W_{before}}{S_{before}}$, was the minimum among all GPUs at that moment due to the algorithm's greedy choice. This minimum KVPR is typically related to the average "pressure" across the system, which, in turn, is argued to be no larger than the optimal maximum pressure. This suggests the inequality: $W_{before}/S_{before} \leq KVPR_{OPT}$
- **Bound 2 (Related to model m_k):** The "pressure" exerted by the critical model m_k must be handled by the optimal solution. A fundamental lower bound on the optimal solution is the maximum pressure any single model would exert if placed alone on an otherwise empty GPU, *i.e.*, $KVPR_{OPT} \geq d_k/(C - w_k)$.

The final step involves integrating these insights to bound $KVPR_{max} = \frac{W_{before}+d_k}{S_{before}-w_k}$. Following Graham's proof [20], we substitute these into the numerator and get $KVPR_{max} \leq KVPR_{OPT} \cdot (1 + \frac{C}{S_{g_{max}}-w_k})$.

A.1.2 TP Support

The model placement algorithm in Algorithm 1 seamlessly integrates models utilizing Tensor Parallelism (TP). We conceptualize a TP model requiring tp_size GPUs as being composed of tp_size distinct parts. For scheduling purposes, we create tp_size entries in the sorted model list for such a model, assigning each entry $\frac{1}{tp\_size}$ of the original weight and request rate. A beneficial property emerges from this decomposition: since these entries have identical $\frac{r_k}{s_k}$ values, they remain adjacent after sorting. This adjacency increases the likelihood that, as the algorithm iterates, these parts are initially assigned to different GPUs due to rising KVPRs. To ensure the distribution, if assigning a TP part to the GPU with the minimum KVPR would result in collocating it with another part of the same original model, we instead assign it to the GPU exhibiting the second-lowest KVPR. Through this decomposition strategy and modified assignment rule, our algorithm effectively considers and manages the placement of TP models alongside single-GPU models.

	Mean E2E (s)	P95 E2E (s)	Req Tput (r/s)	Token Tput (t/s)
MuxServe	7.40	18.85	7.97	3363.53
MuxServe++	5.25	12.09	7.98	3353.09
	Mean TTFT (s)	P95 TTFT (s)	Mean TPOT (ms)	P95 TPOT (ms)
MuxServe	1.47	11.04	21.21	31.95
MuxServe++	0.089	0.320	18.82	33.97

Table 5: Performance comparison of MuxServe and MuxServe++

A.2 MuxServe Calibration

We evaluated the performance of MuxServe++ and MuxServe using three Llama-3.1-8B models under different request rates over a 10-minute period: 199 requests/min, 262 requests/min, and 22 requests/min. All experiments were conducted under the same and consistent conditions. The results are shown in Table 5. As we can see, MuxServe++ achieves comparable or even better performance.