WarmServe: Enabling One-for-Many GPU Prewarming for Multi-LLM Serving

Chiheng Lou[1] Sheng Qi[1] Rui Kang[2] Yong Zhang[2] Chen Sun[2]
Pengcheng Wang[2] Bingyang Liu[2] Xuanzhe Liu[1] Xin Jin[1]
[1]*School of Computer Science, Peking University* [2]*Huawei Technologies Co., Ltd*

Abstract

Deploying multiple models within shared GPU clusters is promising for improving resource efficiency in large language model (LLM) serving. Existing multi-LLM serving systems optimize GPU utilization at the cost of worse inference performance, especially time-to-first-token (TTFT). We identify the root cause of such compromise as their unawareness of future workload characteristics. In contrast, recent analysis on real-world traces has shown the high periodicity and long-term predictability of LLM serving workloads.

We propose *universal GPU workers* to enable one-for-many GPU prewarming that loads models with knowledge of future workloads. Based on universal GPU workers, we design and build WarmServe, a multi-LLM serving system that (1) mitigates cluster-wide prewarming interference by adopting an evict-aware model placement strategy, (2) prepares universal GPU workers in advance by proactive prewarming, and (3) manages GPU memory with a zero-overhead memory switching mechanism. Evaluation under real-world datasets shows that WarmServe improves TTFT by up to 50.8× compared to the state-of-the-art autoscaling-based system, while being capable of serving up to 2.5× more requests compared to the GPU-sharing system.

1 Introduction

The rapid evolution of large language models (LLMs) has produced various models specialized for tasks including chat [1–4], coding [5, 6], and reasoning [7–10]. This proliferation introduces significant challenges for model serving, as platforms must now concurrently host multiple models to meet varied user demands. For example, OpenAI offers more than 20 model variants through its API [11].

Serving multiple LLMs is challenging due to the highly dynamic workloads. As LLM loads frequently change [12–14], a dedicated allocation strategy (i.e., a set of GPUs is dedicated to serve a particular model) leads to low GPU utilization. To solve this problem, existing solutions can be divided into two categories: (1) autoscaling approaches that automatically scale the number of instances for each model according to current loads [12, 15–19], and (2) GPU sharing approaches that colocate models on the same GPUs and perform spatial and temporal sharing [13, 14, 20, 21].

Unfortunately, although these approaches can improve cluster-wide GPU utilization, they have non-negligible impact on inference performance. Autoscaling approaches incur waiting periods for a burst of requests, as model instances should be initialized on-demand. GPU sharing approaches have no initialization overhead, but limit the KV cache space of colocated models, which is essential for LLM serving.

We attribute the inefficiency of existing approaches to the unawareness of future workload characteristics. Due to this lack of foresight, autoscaling can only be triggered after bursty requests arrive, and the model colocation strategy must be stable over time. However, real-world production traces have shown that while the short-term request arrival pattern is *unpredictable*, the long-term statistical characteristics of LLM requests tend to be more *predictable* [22–24]. Our analysis (§2.2) also confirms that LLM request demands in 5-minute windows can be effectively predicted with an accuracy of approximately 93%.

Therefore, one natural solution for efficiently serving multiple LLMs is to provision sufficient instances for models to meet the predicted maximum request loads. However, constantly preparing for peak demands leads to significant GPU consumption and low GPU utilization, as it reserves far more instances than the current load requires.

Our key insight is that we can satisfy future demand without launching the entire instance. Instead, GPUs that already contain parameters for a given model are able to provide potential serving capacity. Upon load spikes, as model weights are already resident on these GPUs, the cluster can quickly create new instances on top of them and begin inference.

This approach creates opportunities to pack backup instances for different models onto the same GPUs. A GPU can store (i.e., prewarm) parameters for multiple models because, in modern LLM deployments, model weights occupy only a small fraction of the total GPU memory. This is due to two main factors: (1) the substantial memory required by the

KV cache to serve long contexts [25–28] and facilitate prefix caching [29–31], and (2) the use of larger-scale parallelism to reduce latency. For example, in the official DeepSeek-V3 deployment, model weights account for 30% and 7% of GPU memory in prefill and decode instances, respectively [3]. Furthermore, prewarming the first several layers is sufficient for rapid startup, as the loading of subsequent layers can be overlapped with the forward computation of the loaded layers.

We propose *universal GPU workers*—GPU workers capable of seamlessly transforming into serving instances for any model. Specifically, we create universal GPU workers on top of idle GPUs by proactively loading weights from different models. During load spikes, these workers can be quickly reconfigured into *dedicated GPU workers* for specific models by evicting the weights of other models, enabling rapid startup. This approach allows a single GPU to serve as a prewarmed backup for multiple models, significantly reducing the total number of GPUs that need to be reserved.

The goal of universal GPU workers is to realize one-for-many prewarming, which has been extensively adopted in reducing cold start latency for serverless functions [32–35]. For example, one can prepare multiple universal containers so that when a request arrives, it can directly use a prewarmed one after loading a few function-specific packages [32,33,36]. However, the unique characteristics of LLMs introduce two challenges to achieving one-for-many prewarming.

First, as LLMs span across multiple GPUs, a prewarmed model requires the availability of all GPUs that host its weights. Therefore, the allocation of a GPU to one model invalidates other models residing on it. Although only a single weight partition of these invalidated models becomes unavailable, the entire prewarming effort for these models is wasted because a rapid startup requires all partitions simultaneously. Given the unpredictability of short-term requests, such interference among colocated models significantly complicates the prewarming decisions for universal GPU workers.

Second, as LLM workloads are dynamic, a GPU that has just been released may be quickly reallocated to other models. In production, LLM requests can increase 5× within 2 seconds [12], which leaves a short time window for prewarming new models. Due to large LLM checkpoints, most prewarming cannot complete before the GPU is allocated again, leading to limited successful prewarming.

To address these challenges, we propose WarmServe, a multi-LLM serving system designed to unleash the potential of one-for-many GPU prewarming. To reduce the prewarming interference among LLMs, WarmServe adopts an evict-aware model placement strategy that prohibits partially overlapping GPU sets during model placement. The strategy minimizes interference by tightly coupling every two colocated models, and maximizes the prewarming performance under potential evictions by isolating high-priority prewarming models.

To ensure a successful prewarming before the next GPU allocation, WarmServe proactively prewarms models before a GPU is released. Specifically, we observe that GPUs that are about to be released usually have idle memory due to fewer ongoing requests than maximum capacity. Thus, WarmServe prewarms models into those GPUs in advance, and stores the new parameters in their unused KV cache space. These GPUs can quickly turn into universal GPU workers after they are released. Since the GPU memory may simultaneously contain tensors from the served model, KV cache, and prewarming models, we propose a zero-overhead memory switching mechanism to efficiently manage these tensors, enabling GPUs to be seamlessly transformed among different types.

Experiments on real-world datasets show that WarmServe reduces the tail TTFT by up to 50.8× compared to the state-of-the-art autoscaling-based system, and is able to handle up to 2.5× more requests compared to the GPU-sharing system. Furthermore, experiments show that WarmServe effectively maintains the decoding performance under various scenarios.

In summary, we make the following contributions.

- We identify the potential of model prewarming in multi-LLM serving and propose universal GPU workers to enable one-for-many GPU prewarming.
- We propose an evict-aware model placement strategy, a proactive prewarming policy, and a zero-overhead memory switching mechanism to unleash the potential of universal GPU workers.
- We evaluate WarmServe comprehensively to show its effectiveness compared to state-of-the-art solutions.

2 Background and Motivation

In this section, we introduce LLM serving, analyze the long-term predictability of real-world LLM serving traces, and summarize challenges.

2.1 LLM Serving

LLM serving is the end-to-end process wherein a client sends a request, known as a *prompt*, to a serving engine, which in turn performs inference and streams the response back to the client. User experience is primarily measured by two key metrics: time-to-first-token (TTFT) and time-per-output-token (TPOT). TTFT is the latency to generate the first token, while TPOT represents the average time interval for generating each subsequent token.

LLM inference. LLM inference consists of two stages. The *prefill* stage generates the first token from the initial prompt. During this stage, the key and value vectors for each token in the prompt are computed and stored in GPUs, known as the *KV cache*. The *decoding* stage then generates the remaining outputs autoregressively. In each decoding step, the model computes and stores the key and value vectors of the last token in the sequence, and generates a new token. As LLM checkpoints are large, they are typically deployed across multiple GPUs using model parallelism, which partitions the weights and synchronizes results at specific stages during inference.

To improve GPU utilization, modern serving systems often process multiple requests in a batch. A maximum batch size is typically configured to prevent performance degradation when too many requests are processed.

Multi-LLM serving. Large-scale LLM service providers, such as OpenAI [11], Google [9], and Anthropic [37], offer a catalog of multiple models tailored to different use cases. Consequently, the serving cluster must be capable of concurrently hosting these models and efficiently managing dynamic workloads for each. In this case, a static GPU allocation strategy leads to resource wastage. To address this inefficiency, a variety of multi-LLM serving systems have emerged, which fall into two categories.

• *Autoscaling solutions* dynamically control the number of instances for each model based on current loads. Upon a load spike, the system creates new instances to serve arriving requests. As instance creation is on the critical path of request handling, these systems reduce this latency by caching models locally [15, 18, 19], fetching models from peers [12, 16], and distributing models across servers [17]. Despite these efforts, bursty requests still suffer from long waiting latency in these systems because instances are created on demand.

• *GPU sharing solutions* share GPUs among models [13, 14, 20, 21]. Models are colocated on the same GPUs, and are allocated specific ratios of GPU computational power based on the traffic. In these systems, GPU sharing limits the KV cache space of each model. Additionally, these systems usually enlarge the parallelism strategy of LLMs to place more models together, degrading inference performance.

WarmServe uses the autoscaling approach, i.e., it assigns instances with dedicated GPUs and scales the number of instances for each model. However, unlike existing autoscaling systems that passively create instances after bursty requests arrive, WarmServe *prewarms* models in advance by loading their parameters into shared idle GPUs. This synthesis of on-demand autoscaling and proactive GPU sharing allows WarmServe to harness the key benefits of each design.

Autoscaler. An autoscaler is responsible for dynamically adjusting the number of instances assigned to each model in multi-LLM serving systems [12, 15–19]. Typically, it periodically checks the current load and number of active instances for each model. If the resource utilization of a model falls below a predefined threshold, the autoscaler attempts to shut down some of its instances. However, since requests are often distributed in a load-balanced manner (e.g., using round-robin), most instances still have active requests and cannot be terminated immediately. To address this, the autoscaler first stops scheduling new requests to the targeted instances marked for termination, and then waits for the completion of ongoing requests. This waiting period is referred to as the *grace period* for an instance.

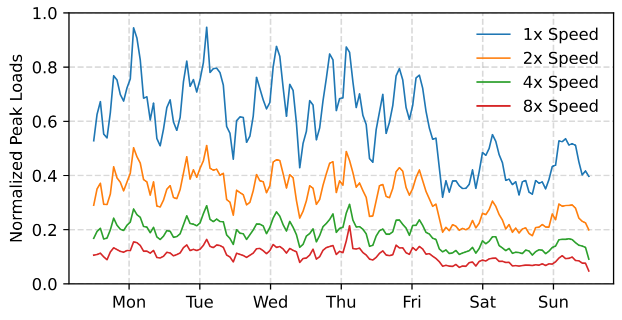


Figure 1: Peak loads under 5-minute windows of the AzureConv [23] trace. Data smoothed using cubic spline interpolation.

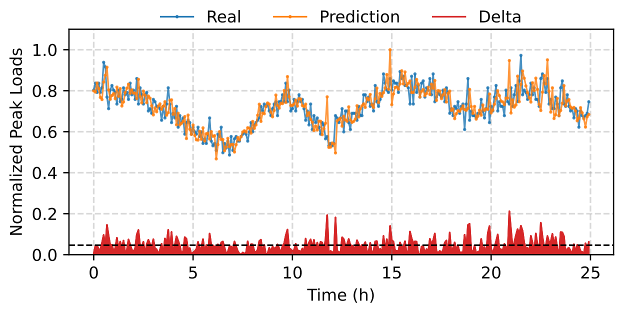


Figure 2: Real and predicted peak loads under 5-minute windows and 8× Speed of the AzureConv [23] trace. The delta is shown in absolute values and the black dotted line represents its average value.

2.2 Workload Predictability

While the short-term burstiness of LLM requests is often regarded as unpredictable [20, 38], recent analysis of real-world LLM services has revealed that the long-term statistical characteristics of requests are relatively periodic and predictable [22–24]. In this paper, we focus on the characteristics of the average and peak load, which refers to the average and maximum number of concurrent requests within a given time window, respectively.

Take peak load as an example. Figure 1 shows the values of load peaks in 5-minute windows under varying inference speeds, using data collected from the AzureConv [23] trace. The running time of a request is computed based on its input and output lengths. The results demonstrate that the load peaks follow a periodic pattern across all speeds, indicating that peak values within a given time window can be effectively predicted using historical data.

To validate this assumption, we develop a simple corrective seasonal predictor (§5.1) to predict the peak load for each time window. Figure 2 shows the prediction performance of our predictor under the most dynamic load (8× Speed). By using the recorded peak loads of both previous days and recent times, the predictor achieves an average accuracy of 92.7%, which is sufficient for prewarming models. Notably, predicting the average load is even easier, with an average accuracy of 94.7%.

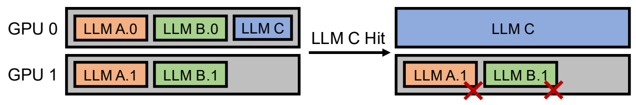


Figure 3: Example of cluster-wide prewarming interference.

This paper leverages the workload predictability to prewarm appropriate instances for different LLMs. By ensuring that sufficient instances are prewarmed for each LLM, most incoming requests can be served without delay. Additionally, we propose universal GPU workers to prewarm multiple LLMs simultaneously, significantly reducing the GPU resources consumed by prewarming.

2.3 Challenges

While prewarming multiple LLMs on several universal GPU workers is promising, unleashing its full potential presents two challenges listed below.

• *Cluster-wide prewarming interference.* The large checkpoints of LLMs typically necessitate their deployment across multiple GPUs. Since a universal GPU worker is responsible for concurrently prewarming multiple models, these colocated models can experience prewarming interference. For example, Figure 3 illustrates a potential weight placement strategy when prewarming three LLMs using two GPUs. As the deployment of LLMs A and B requires two GPUs, their weights are partitioned and placed on both GPUs during prewarming.

When LLM C encounters a burst of requests, the system spawns a new instance on GPU 0 for LLM C, and evicts the parameters of LLMs A and B from that GPU. This eviction renders the first weight partition of LLMs A and B unavailable, which also invalidates their second partitions since prewarming requires all parameters to reside on GPUs. Consequently, prewarming contention arises between every two interleaved models: a prewarming hit for one model invalidates all partitions of another, even if they only share a single GPU. Such cluster-wide prewarming interference significantly complicates the design of effective model placement strategies.

• *Frequent GPU reallocation.* LLM workloads are highly dynamic, with request rates capable of increasing 5× within just 2 seconds [12]. This leads to a high frequency of GPU allocation and deallocation, where idle GPUs are quickly assigned to new models. Such rapid reallocation cycles leave a narrow time window for prewarming new models. As loading models into GPUs is time-consuming due to limited PCIe bandwidth and the large size of LLM checkpoints, most prewarming attempts fail to complete before GPUs are reallocated, leading to low prewarming hit probability.

3 WarmServe Overview

To address these challenges, we design and build WarmServe, a multi-LLM serving system that strategically prewarms models to enable one-for-many prewarming. Figure 4 shows the architecture of WarmServe. WarmServe employs a global manager to control a pool of GPUs (i.e., GPU workers), categorized as idle, universal, or dedicated. The global manager dynamically loads or evicts model weights on these GPUs, adapting their roles in different scenarios. Specifically, idle GPU workers transition into universal workers after prewarming several LLMs. Universal workers, in turn, become dedicated GPU workers when a new instance of a prewarmed model is launched. Once the instance is released, these workers revert to idle status and begin prewarming new models.

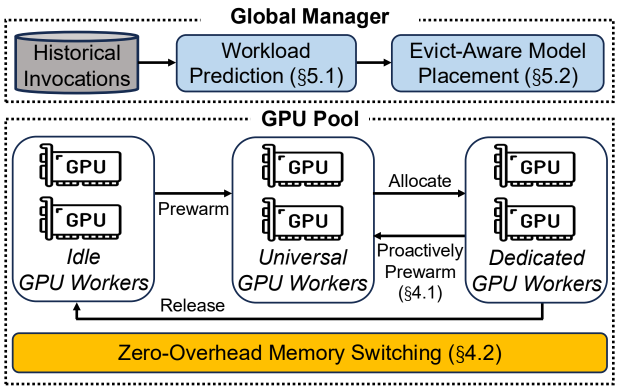
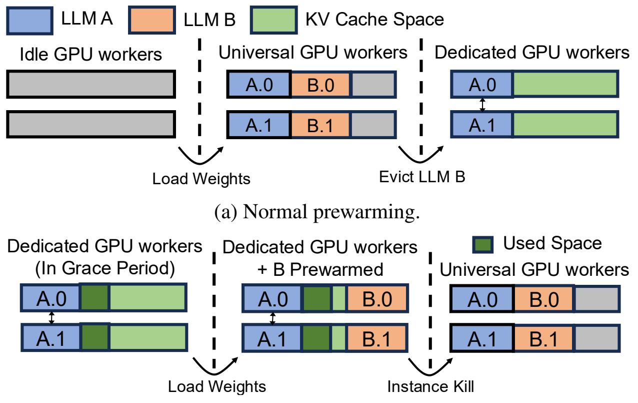


Figure 4: WarmServe system overview.

WarmServe adopts an evict-aware model placement strategy (§5.2) to mitigate prewarming interference. This strategy restricts GPU sharing to occur only when the GPU set of one prewarmed LLM fully contains that of another, ensuring the models are tightly coupled. Additionally, WarmServe computes the scores for the prewarming replicas of each model based on load predictions (§5.1). It then isolates high-score replicas to enhance the overall prewarming effectiveness.

To address the challenge of frequent GPU reallocation, WarmServe proactively prewarms models on dedicated GPU workers (§4.1). This strategy is applied to grace-period instances, and the weights of new models are stored in unused KV cache space. Once the instance is terminated, its dedicated GPU workers can seamlessly transform into universal workers without additional weight loading. To ensure the flexible coexistence of serving model parameters, prewarmed model weights, and KV cache, WarmServe dynamically switches the virtual addresses of GPU memory and adjusts address mappings during prewarming and allocation (§4.2). This memory management technique hides all time-consuming operations behind other actions to achieve zero overhead.

WarmServe has no impact on inference performance since it allows serving instances to have exclusive access to GPUs. Meanwhile, it adopts the idea from GPU sharing systems [13, 14, 20, 21] to colocate prewarmed models on universal GPU workers. This approach maximizes prewarming efficiency and significantly reduces the TTFT for cold-start models since the model preparation process is launched in advance.



(a) Normal prewarming.

(b) Proactive prewarming.

Figure 5: Overview of GPU worker lifecycle in WarmServe.

4 WarmServe Universal GPU Workers

WarmServe classifies GPU workers in the cluster into three categories: idle, universal, and dedicated. While dedicated GPU workers run inference tasks for specific models, universal workers store weights from multiple models simultaneously. This allows workers to quickly transition into serving instances for any of these models. To prewarm an LLM, its weights are loaded into several selected idle GPU workers within the same server. These GPUs are then able to immediately begin inference upon scaling up.

Figure 5(a) illustrates the lifecycle of a GPU worker. WarmServe loads weights from different LLMs into idle GPU workers to transform them into universal workers. When a model experiences a load spike, these workers can be exclusively assigned to that model after evicting all other models. At this point, the workers become dedicated GPU workers, with the remaining GPU memory allocated for the KV cache.

To reduce the size of loaded weights, we avoid prewarming the entire model across universal GPU workers. Instead, we prewarm only the first several layers and perform parallel model loading and inference when spawning the instance, similar to prior works [12, 16, 17, 39]. Specifically, we conduct an offline profiling pass to obtain the number of layers that must be preloaded to ensure that inference is not blocked. These numbers are then directly applied during online prewarming.

4.1 Proactive Prewarming

WarmServe leverages proactive prewarming to quickly prewarm models under dynamic LLM workloads. As described in §2.1, instances enter a grace period before final termination. During this period, we proactively load weights from different models into the GPUs of these instances. While they continue processing ongoing requests, we utilize their unused KV cache space to store new parameters. This approach is viable because these instances are typically underutilized and no longer receive additional requests.

Figure 5(b) illustrates the proactive prewarming process. For dedicated GPU workers in grace periods, we monitor the amount of free KV cache space. If sufficient memory is available, we load the weights of other models into unused KV cache to prewarm new models. Once the instance is terminated, these workers transition directly into universal GPU workers that contain both the previously running model and the proactively loaded models. This quick conversion allows us to prepare universal GPU workers in advance, ensuring prewarming completion before the next allocation.

However, determining the appropriate amount of free KV cache space to allocate for prewarming new models is challenging. Since these dedicated GPU workers are still processing ongoing requests, their KV cache requirements can increase over time. For example, a single request with a sequence length of 128K can consume the majority of the KV cache space when serving a Llama3.1-8B model with a single A100-40G GPU [4]. Consequently, an aggressive prewarming strategy that provides too much memory for prewarming may lead to insufficient KV cache space, forcing the eviction of the recently loaded weights. Conversely, a conservative prewarming strategy can cause the underutilization of GPU memory, slowing down the conversion into universal workers.

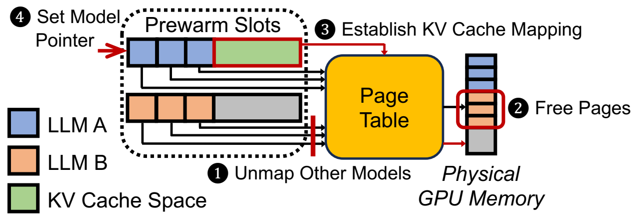
Given that the total generation length of an LLM request is commonly regarded as unpredictable [40, 41], WarmServe strikes a balance between aggressive and conservative prewarming strategies. Formally, let R represent the number of ongoing requests and C be the maximum batch size of the instance. The GPU utilization at the request granularity can be computed as R/C. Furthermore, let M denote the total KV cache capacity and K be the currently consumed portion. The amount of GPU memory to reserve for prewarming can be determined using the following formula.

$$\text{Reservation Target} = \max\left(M \cdot \frac{R}{C},\, K + \frac{M}{C}\right). \tag{1}$$

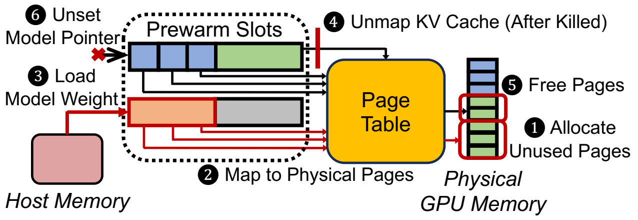
Eq. 1 consists of two parts: (1) the expected KV cache usage based on the number of ongoing requests and (2) the actual KV cache usage plus a reserved buffer for potential KV space expansion. Since the total generation length is unpredictable, we allocate additional KV cache space for ongoing requests by estimating the average KV cache usage per request, calculated as M/C.

Upon the completion of each request for a grace-period instance, WarmServe verifies whether the size of the current reserved KV cache exceeds the reservation target. If it does, the surplus KV blocks are released to prewarm new models. The instance then reports the amount of newly freed memory to the global manager via a shared memory region.

In addition to proactive prewarming, WarmServe ensures that even if prewarming has not completed upon instance startup, the partially prewarmed weights are immediately operational. In this case, the instance can directly begin inference on the available layers while asynchronously loading the remaining parameters in the background.



(a) Transition a universal GPU worker to a dedicated one.



(b) Proactively prewarm models on a dedicated GPU worker (1-3) and then seamlessly transition the GPU to a universal GPU worker (4-6).

Figure 6: Overview of zero-overhead memory switching in WarmServe.

4.2 Zero-Overhead Memory Switching

WarmServe uses a zero-overhead memory switching strategy to efficiently switch GPU memory among different LLMs. The strategy leverages the CUDA Virtual Memory Management (VMM) API [42] to directly manipulate the GPU page table, similar to previous works [14, 27, 43]. This API allows us to pre-allocate all physical pages in the GPU and map virtual memory addresses to them on demand.

Our method begins by pre-allocating multiple virtual memory regions on each GPU, where the size of each region is equivalent to the total available GPU memory. Each of these regions serves as a *prewarm slot* that is used to store an individual prewarmed model. During the prewarming phase, we assign models different slots and allocate the necessary physical pages for each model according to model sizes. Subsequently, the corresponding virtual addresses within the slot are mapped to these physical pages, with model weights always placed at the start of its respective slot.

Figure 6(a) illustrates the process of converting a universal GPU worker into a dedicated one for a specific model. Upon a successful prewarm hit, we first release the resources of other prewarmed models by unmapping their virtual pages and freeing the underlying physical pages. Next, we create additional mappings for the prewarm slot containing the target model. The unmapped virtual pages within this slot are mapped to all remaining physical pages on the GPU, which will serve as the KV cache. At this point, a complete one-to-one mapping is established between the virtual address space of the prewarm slot and the entire physical GPU memory. Finally, we direct the inference framework to this active slot by configuring the model pointer. This ensures that the framework accesses the model weights and KV cache through a single, contiguous virtual address space, effectively concealing the non-contiguous nature of the underlying physical pages and isolating it from other inactive prewarm slots.

This mechanism is also applicable when launching an instance with a model that has not been prewarmed. In this case, a series of initialization steps is performed. First, all existing prewarmed slots are reclaimed and the loaded model weights are invalidated. Next, an empty slot is allocated to the target model, and all physical pages are mapped to it. Finally, the model weights are loaded into the newly allocated slot, making the GPU ready for inference.

Figure 6(b) further illustrates how WarmServe proactively prewarms models on dedicated GPU workers. During a GPU's grace period, unused KV cache blocks are leveraged to prewarm new models. We compute the corresponding physical pages of these blocks, and map them to the prewarm slot of the new model. The model's weights are then loaded into these pages by accessing the slot's virtual address.

When the original instance terminates, its associated KV cache is reclaimed by unmapping the virtual addresses and freeing the physical pages. The model pointer is also cleared, signaling the absence of an active model. The GPU then transitions back to a universal state, now holding the prewarm slots of both the newly prewarmed models and the previously active one, ensuring readiness for immediate deployment.

Achieving zero-overhead switching. The primary performance bottleneck in our GPU management lifecycle is the latency of modifying page tables through the CUDA VMM API. For instance, mapping a 10GB virtual address space can take up to 0.2 seconds [43]. To eliminate this overhead, WarmServe decouples page table manipulations from the critical execution path by overlapping them with other operations. These modifications are categorized into two types: mappings for model loading and mappings for the KV cache.

For model loading, WarmServe pipelines the mapping and data transfer operations. As soon as a virtual page is mapped, a data copy for the corresponding weights is immediately triggered. Since the time to map a single page is significantly shorter than the data transfer time, this fine-grained pipelining strategy effectively hides the mapping latency.

For the KV cache, mappings are performed in the background. Since the inference framework consumes cache space at a slower rate than the mapping process produces it, the mapping overhead is fully overlapped. Unmapping operations are also executed asynchronously, as they do not block any subsequent actions.

Consequently, by strategically overlapping all page table manipulations with data transfers and other non-blocking operations, WarmServe ensures the memory switching process incurs negligible overhead.

5 WarmServe Global Manager

The global manager is responsible for managing GPU workers of different types to efficiently prewarm LLMs. To accurately forecast workloads, the manager first splits the time into several windows based on a fixed window size W. At the beginning of each time window, the manager predicts the average and peak loads for each model and uses these predictions to devise an optimal placement strategy. The global manager also receives the scale-down signals from the autoscaler to identify stopping instances. For these instances, it initiates a memory reclamation process by querying them to report their unused KV cache blocks. This reclaimed memory is leveraged to prewarm new models, enabling a quick transition into universal GPU workers.

5.1 Workload Prediction

WarmServe predicts the model workload for the next time window using a simple corrective seasonal predictor (CSP) [44–46]. Take peak load as an example, let $L_{k,i}$ represent the peak load of the i-th time window on day k. The prediction is based on two components: a seasonal pattern and a corrective pattern. For the seasonal pattern, we calculate the historical average peak load for that specific time window. Formally, to predict $L_{k,i}$, we use the average peak load of the i-th window in the past several days, calculated as follows.

$$P_{k,i} = \frac{1}{D}\sum_{j=1}^{D} L_{k-j,i}, \tag{2}$$

where D is the number of historical days to sample, typically set based on the stability of daily traffic patterns.

For the corrective pattern, we incorporate recently observed peak loads to quantify the delta between the current trend and the historical seasonal pattern. We denote this delta as $\Delta_{k,i}$. It is calculated as an exponentially weighted average of the prediction errors from the previous N windows.

$$\Delta_{k,i} = \frac{\sum_{j=1}^{N}(L_{k,i-j} - P_{k,i-j})\cdot 2^{j-1}}{2^N - 1}, \tag{3}$$

where N is the size of the lookback window, typically set to 10. This weighting scheme gives more importance to more recent errors.

The final prediction, $\hat{L}_{k,i}$, is the sum of the seasonal component and this corrective term.

$$\hat{L}_{k,i} = P_{k,i} + \Delta_{k,i}. \tag{4}$$

The prediction of average load follows the same procedure, where $L_{k,i}$ is simply redefined as the average load within the window. Due to the highly periodic nature of LLM workloads, our predictor demonstrates strong performance on real-world datasets. With a 5-minute window size, it achieves an average relative error of 5.3% for average loads and 7.3% for peak loads on the AzureConv [23] trace. While more sophisticated prediction algorithms, such as ARIMA [47] and deep learning models [48], could potentially yield higher accuracy, we found that CSP is sufficiently effective for guiding model prewarming while imposing negligible overhead.

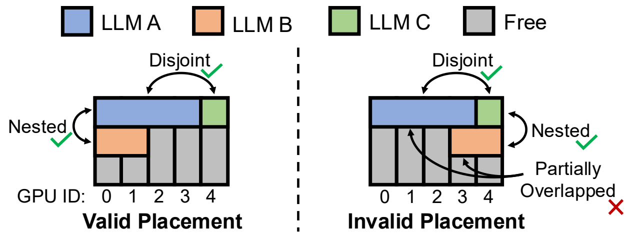


Figure 7: Placement guideline of WarmServe.

5.2 Evict-Aware Model Placement

As shown in §2.3, prewarming LLMs across multiple GPUs can lead to interference between colocated models. To mitigate this, WarmServe employs a placement strategy governed by two primary guidelines. The first guideline strictly prohibits partial GPU sharing. Specifically, for any two models, the set of GPUs allocated to them must either be entirely disjoint or one set must be a complete subset of the other. This constraint ensures that contention for GPU resources is limited to models in a nested arrangement, thereby reducing the interference patterns that arise from more complex overlaps.

Figure 7 provides a visual representation of this placement restriction. In a valid placement, models are either disjoint (using separate GPUs) or nested (where one model's GPUs are a subset of another's). Placements with partial overlap, where models share a subset of GPUs while also holding exclusive ones, are explicitly forbidden. The rationale for this prohibition is that partial overlaps can escalate resource contention. For example, as shown in the figure, moving LLM B from GPUs (0,1) to GPUs (3,4) would cause it to partially overlap with LLM A, and this would simultaneously create a new contention point with LLM C.

The second placement guideline focuses on prioritizing models based on their anticipated demand. WarmServe aims to isolate models that have high prewarming hit probabilities by allocating them to disjoint GPU sets. This strategy prevents these critical, high-priority models from interfering with one another's performance. Subsequently, models with a lower prewarming hit probability can be colocated on these same GPUs, leveraging unused resources with minimal performance impact on the primary models.

At runtime, WarmServe continuously monitors the status of all GPUs in the cluster. When available GPU memory is detected, it initiates the prewarming process. This process consists of two stages. First, the system determines the number of replicas to prewarm for each model and calculates a *prewarming score* for each replica. Second, it places these replicas onto the available GPUs according to their scores, strictly adhering to the placement guidelines outlined above.

Obtaining prewarming scores. We classify replicas for each model into two categories.

• *Basic replicas.* These replicas are prewarmed to ensure sufficient model instances under average loads. For models whose active instances cannot meet the predicted average load, we create basic replicas to fill these gaps.

• *Burst replicas.* These replicas are used to tackle load spikes. After creating sufficient basic replicas, we continue to prewarm additional replicas until the total number of instances (both active and prewarmed) can serve the predicted peak load. These additional replicas are designated as burst replicas, ensuring warm starts under the maximum loads.

Formally, given a model with K active instances and a batch size of B, and with predicted average and peak loads denoted as L_A and L_P respectively. The number of basic (N_{basic}) and burst (N_{burst}) replicas to be prewarmed is calculated as follows.

$$N_{basic} = \max\left(\lceil L_A/B \rceil - K,\ 0\right); \quad (5)$$

$$N_{burst} = \max\left(\lceil L_P/B \rceil - N_{basic} - K,\ 0\right). \quad (6)$$

Within each category, replica priority is further refined using a prewarming score, S. The score is calculated differently for each replica type.

$$S_{basic} = \exp\left(-\frac{i}{N_{basic}+N_{burst}}\right)\cdot T_c; \quad (7)$$

$$S_{burst} = \exp\left(-\frac{N_{basic}+i}{N_{basic}+N_{burst}}\right)\cdot T_c \cdot \frac{L_P - L_A}{L_A}, \quad (8)$$

where i is the zero-indexed rank of the replica within its category, and T_c is the latency of loading the whole model weights into GPUs, which is obtained through offline profiling.

The score for basic replicas (Eq. 7) is a product of two factors. The first, an exponential decay term, models the diminishing returns of prewarming replicas. As more replicas are prewarmed (increasing i), the incremental utility of the next one decreases. The second factor, T_c, prioritizes models with longer loading times, as they incur a higher penalty if a prewarmed instance is unavailable.

For burst replicas, the score (Eq. 8) contains an additional burstiness factor, $(L_P - L_A)/L_A$. This term represents the burstiness of the load in the upcoming time window, ensuring that models expecting a larger spike in traffic are prioritized accordingly. Note that basic replicas always have higher priorities than burst replicas, regardless of their scores.

Placement algorithm. Our placement algorithm strategically allocates prewarming replicas to GPU workers. It begins by calculating the prewarming score for each replica and then proceeds to place them according to specific placement guidelines. Replicas are processed in descending order of their prewarming scores, while basic replicas are prioritized over burst replicas. For each replica r, the following procedure is executed.

• *Candidate worker identification*. First, the algorithm identifies a set of candidate GPU workers. This set includes all idle and universal workers in the cluster with sufficient available memory. To facilitate proactive prewarming, dedicated workers in a grace period are also considered candidates. A replica of a model with size S and parallelism degree D requires S/D memory per GPU worker.

Algorithm 1 Evict-aware Model Placement Algorithm

Input: #models N, #GPUs P; model size S_i, parallelism degree D_i, batch size B_i, current number of instances K_i, predicted average load L_{A_i}, peak load L_{P_i}, cold start latency T_{c_i} for each model i; available memory M_j for each GPU j.

Output: to-prewarm models and placement.

```
$Replicas \leftarrow \emptyset$                     ▷ Replicas to be prewarmed
$Selected \leftarrow \emptyset$                     ▷ Selected replicas to prewarm
for $i \in \{1, 2, \cdots, N\}$ do
    $N_{basic} \leftarrow \max\left(\lceil L_{A_i}/B_i \rceil - K_i,\ 0\right)$
    $N_{burst} \leftarrow \max\left(\lceil L_{P_i}/B_i \rceil - N_{basic} - K_i,\ 0\right)$
    for $r \in \{1, 2, \cdots, N_{basic}\}$ do
        $score_{f,r} = \exp\left(-\frac{r-1}{N_{basic}+N_{burst}}\right)\cdot T_{c_i}$
        $Replicas \leftarrow Replicas \cup (i, score_{f,r})$
    for $r \in \{1, 2, \cdots, N_{burst}\}$ do
        $score_{b,r} = \exp\left(-\frac{N_{basic}+r-1}{N_{basic}+N_{burst}}\right)\cdot T_{c_i}\cdot\frac{L_{P_i}-L_{A_i}}{L_{A_i}}$
        $Replicas \leftarrow Replicas \cup (i, score_{b,r})$
$Replicas' \leftarrow$ sort($Replicas$, $key = (type, score)$)
for $(i, score) \in Replicas'$ do
    $Demand_{GPU} \leftarrow (D_i, S_i/D_i)$      ▷ Prewarming model $i$ requires $D_i$ GPUs, each has at least $S_i/D_i$ free memory.
    $G \leftarrow$ GPU groups that can accommodate $Demand_{GPU}$
    $G' \leftarrow \{g \in G \mid g$ satisfies placement requirements$\}$
    if $G'$ is $\emptyset$ then
        continue
    else
        $H_g, S_g \leftarrow$ Highest and sum of scores of overlapped replicas for every $g \in G'$
        if $\min\{H_g\} < score$ then
            $g' \leftarrow g \in G'$ that has least $S_g$ while $H_g < score$
        else
            $g' \leftarrow g \in G'$ that has least $S_g$
    $Selected \leftarrow Selected \cup (i, g')$
    Update $M_1, \cdots, M_P$ after placing model $i$ to $g'$
return $Selected$
```

• *Placement group formation*. Next, from the pool of candidate GPU workers, the algorithm attempts to form valid placement groups. A valid group must meet two requirements: (1) all workers in the group must be located on the same server for inference performance guarantee, and (2) the group's workers must not partially overlap with any other existing prewarming replica. If no valid group can be formed, we continue to consider the next replica.

• *Optimal group selection*. If one or more valid groups exist, the algorithm greedily selects the optimal one. The selection process prioritizes groups where the new replica's score is higher than any other existing replica that is nested with the group. If multiple such groups are available, the one with the

Model	Size (GB)	# Required GPUs	Sampling Time
Llama2-7B-0	12.55	1	Thursday
Llama2-7B-1	12.55	1	Friday
Llama2-13B	24.24	2	Saturday
Llama2-70B	128.49	4	Sunday

Table 1: Specification of models in our experiments.

minimum sum of scores from its nested replicas is chosen. Otherwise, the algorithm defaults to selecting the group with the minimum sum of scores.

This placement strategy is twofold: it isolates high-priority replicas to prevent mutual interference, while placing lower-priority replicas in a way that minimizes their impact on existing replicas. Algorithm 1 outlines our evict-aware model placement algorithm. Additionally, when creating new instances on top of prewarmed models, WarmServe reduces the impact on existing prewarmed models by minimizing the sum of prewarming scores for evicted replicas.

6 Implementation

WarmServe is implemented based on vLLM [49], extended with approximately 1.8K lines of C++ and Python code to support universal GPU workers. The global manager is implemented in about 4K lines of Python code.

GPU worker management. Each GPU worker in the cluster is managed by a Ray Actor [50] that performs model inference. To transition a universal GPU worker into a dedicated one, we instantiate a vLLM serving engine and connect it to actors of its allocated workers. We intercept the engine's remote function calls to these actors to leverage prewarmed model parameters and enable proactive prewarming.

Prewarming other instance startup stages. Creating an instance involves multiple stages apart from loading model parameters [17]. WarmServe prewarms time-consuming stages to achieve sub-second instance startup, targeting the loading of libraries and establishment of communication groups.

• *Pre-loading library*. We maintain a pool of vLLM processes with all necessary libraries loaded. A new serving engine starts by taking over a prepared process and receiving model-specific arguments. The idle processes are blocked and consume no CPU resources.

• *Pre-establishing communication group*. When a model is prewarmed on multiple GPUs, a communication group is pre-established among them. Upon prewarming hit, the GPU workers leverage this existing group to synchronize messages during inference. We utilize PyTorch MultiWorld [51] to enable a GPU runtime to simultaneously host multiple communication groups with different peers.

7 Evaluation

In this section, we present experimental results to validate the efficiency and effectiveness of WarmServe. Our evaluation shows that a successful prewarming achieves sub-second

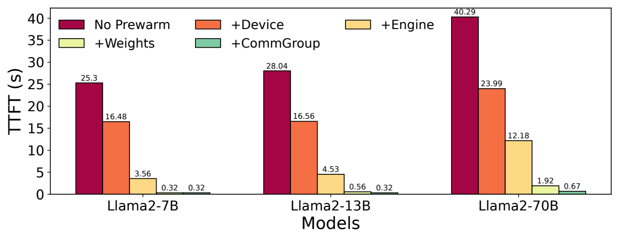


Figure 8: Prewarming performance breakdown.

latency for generating the first token (§7.2). In end-to-end experiments, WarmServe significantly reduces the P95 and P99 TTFT by up to 50.79× compared to the autoscaling-based system, while being capable of serving up to 2.5× more requests than the GPU-sharing system (§7.3). Our analysis of the workload predictor reveals an average relative error ranging from 5.25%–11.16% across different targets and traces, which has potential to be further improved given more historical data (§7.4). Lastly, the ablation study confirms the substantial performance gains attributable to the evict-aware model placement strategy and proactive prewarming (§7.5).

7.1 Experiments Setup

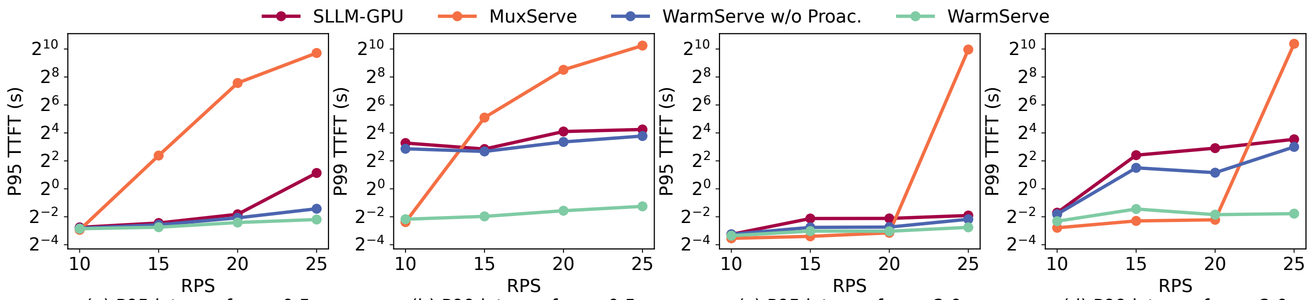
Testbed. We evaluate WarmServe on two servers, each having eight GPUs, 64 CPUs, and 2TB of host memory. A single GPU has around 2K TFLOPS computational power for FP16 precision and intra-server GPUs are connected via NVLink 4.0 with a bandwidth of 400GB/s. Servers are connected with eight 200Gbps RDMA NICs per server. Model weights are stored in host memory and are loaded into GPUs on demand via PCIe5.0x16 channels with a bandwidth of 128GB/s.

Workloads. We use the Llama2 model series [52] with FP16 precision and the details of models are shown in Table 1. The workloads consist of four models, with Llama2-7B duplicated to increase the scaling frequency, following a similar approach as prior works [13, 15, 17, 20]. This setting ensures that all models are able to concurrently scale to two instances in our testbed, thereby creating more opportunities to evaluate scaling behavior. With respect to memory capacity, a single GPU can store the weights of six whole Llama2-7B models.

Following the prior work [12], we use the AzureConv and AzureCode trace [23] to generate workloads. The requests for models are sampled from different days in the trace and we directly use the number of input and output tokens provided by the trace. The request rate for each model is generated using power-law distribution with an exponent α, similar to the prior work [13]. We use a variable *request per second* (RPS) to control the number of requests generated in total.

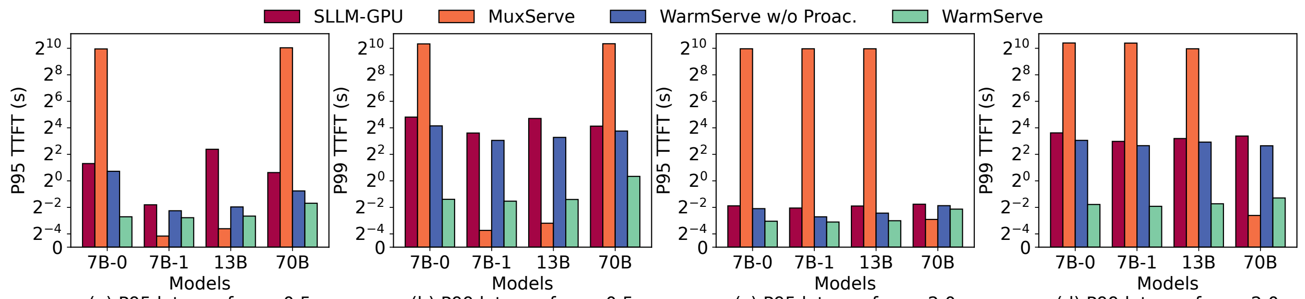
Baselines. We compare WarmServe with the following baselines.

• **ServerlessLLM-GPU (SLLM-GPU)** [15]: Since model weights already exist in host memory, we extend the caching mechanism of ServerlessLLM to GPU. When an instance stops, its model parameters remain in GPU memory for future



(a) P95 latency for α=0.5 (b) P99 latency for α=0.5 (c) P95 latency for α=2.0 (d) P99 latency for α=2.0

Figure 9: TTFT of systems in different settings. A logarithmic scale is used for the y-axis.



(a) P95 latency for α=0.5 (b) P99 latency for α=0.5 (c) P95 latency for α=2.0 (d) P99 latency for α=2.0

Figure 10: TTFT for models under RPS=25. A logarithmic scale is used for the y-axis.

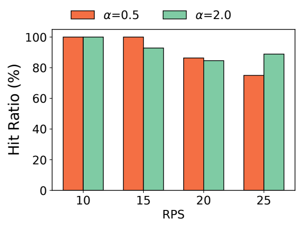


Figure 11: Prewarming hit ratios in end-to-end experiments.

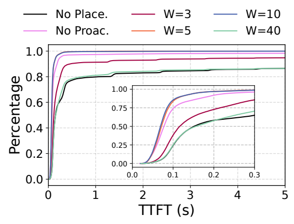


Figure 12: TTFT CDF when disabling evict-aware model placement, proactive prewarming, or using other window sizes.

invocation. We implement ServerlessLLM-GPU based on WarmServe's zero-overhead memory switching strategy.

• **MuxServe** [13]: It colocates models on GPUs and uses CUDA MPS [53] to isolate inference tasks. LLM serving instances are created in prior. As the original MuxServe implementation is based on a low vLLM version, we reimplement it using the same vLLM version as WarmServe for fairness.

All systems use vLLM [49] to run inference. SLLM-GPU and WarmServe adopt a batch size of 32 per instance, while MuxServe uses the configuration generated by itself. We do not compare with BlitzScale [12] since its documentation is incomplete and we are unable to install it in our environment.

Metrics. We focus on the time-to-first-token (TTFT) and time-per-output-token (TPOT) in our experiments, where TTFT is defined as the end-to-end duration from request submission to receipt of the first token, and TPOT is the average time interval for generating each subsequent token.

7.2 Prewarming Performance

We first evaluate the effectiveness of prewarming by measuring the instance startup latency after the model has been prewarmed. Figure 8 shows the incremental improvement achieved by prewarming each stage in instance creation. Starting with the original vLLM system (No Prewarm), we apply the following prewarming methods step-by-step: pre-initialize GPU workers (+Device), pre-create serving endpoints (+Engine), pre-load model weights to GPU (+Weights), and pre-create communication groups (+CommGroup).

The results demonstrate that a successful prewarming can significantly reduce TTFT. First, an already initialized GPU worker can reduce the TTFT by 1.6×–1.7×, owing to pre-created Ray actors. Secondly, prewarming serving endpoints prevents loading complex packages during instance startup and achieves a 2.3×–4.7× TTFT reduction further. As both prewarming methods are unrelated to specific models, all newly started instances can enjoy these TTFT reductions in WarmServe. Finally, by loading model weights to GPUs and creating communication groups in advance, WarmServe can further improve the TTFT by 7.0×–7.8×, achieving a 29.8×–54.8× TTFT reduction in total. Under a successful prewarming, WarmServe can generate the first token in ∼665ms for a Llama2-70B model when the instance is created on demand. We believe that this speed is able to satisfy most service level objectives for LLM serving in production.

7.3 End-to-End Experiments

We further evaluate the effectiveness of WarmServe through comprehensive end-to-end experiments. Figure 9 presents the

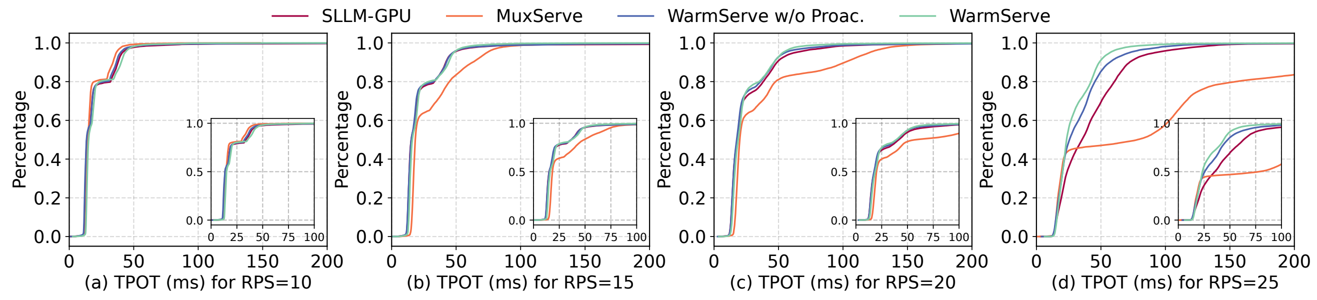


Figure 13: TPOT CDF of systems for α=0.5.

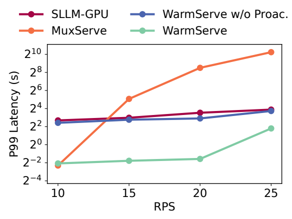


Figure 14: Tail TTFT latency on AzureCode for α=0.5.

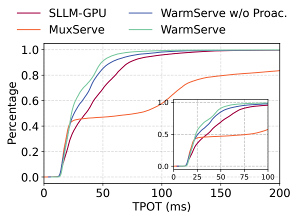


Figure 15: TPOT CDF on AzureCode for α=0.5 and RPS=25.

tail TTFT of systems under various scenarios. To validate the effectiveness of proactive prewarming, we also conduct experiments on WarmServe with proactive prewarming disabled.

WarmServe consistently delivers low TTFT across all settings, significantly outperforming other systems that exhibit high latency, particularly under heavy loads. Compared to SLLM-GPU, WarmServe achieves a 1.07×–10.06× reduction in P95 TTFT and a 1.53×–50.79× reduction in P99 TTFT by rapidly launching new instances from prewarmed models. Moreover, due to the dynamic nature of workloads, proactive prewarming largely improves prewarming efficiency, reducing tail TTFT by 1.03×–32.87×. The tail TTFT in autoscaling-based systems remains stable, as higher traffic merely increases the frequency of scaling events rather than the percentage of requests that experience delays.

The GPU-sharing system, MuxServe, exhibits performance comparable to WarmServe under light loads due to its pre-created model instances. However, its static model placement strategy limits serving capacity and degrades performance for colocated models, leading to severe queuing under heavy loads. For example, at a RPS of 15 and α=0.5, MuxServe's P95 and P99 TTFT are 34.5× and 134.03× higher than those of WarmServe, respectively.

TTFT for different models. Figure 10 details the TTFT for different models under RPS=25. For α=0.5, MuxServe colocates the 7B-0 and 70B models on 8 GPUs, whereas for $\alpha = 2.0$, the 7B-0, 7B-1, and 13B models all share the same 8 GPUs. Although MuxServe can achieve low TTFT for models that do not share GPUs, colocated models are constrained by limited GPU resources, leading to severe request queuing. GPU sharing also introduces performance interference. For example, under the α=2.0 setting, the 13B model receives only 8.2% of total requests but still experiences significant queuing. This is due to resource contention from the high-demand 7B-0 model, which processes 70.2% of the requests.

In contrast, WarmServe maintains stable TTFT performance across all models and settings. It achieves a 1.29×–73.60× reduction in tail TTFT compared to SLLM-GPU, and a 1.20×–46.52× reduction compared to itself with proactive prewarming disabled.

Prewarming hit ratio. Figure 11 demonstrates the effectiveness of one-for-many prewarming by presenting the prewarming hit ratios of WarmServe. Under light loads, the cluster has sufficient idle GPUs, allowing WarmServe to prewarm all required model replicas and ensure a 100% prewarming hit ratio. As the load increases, the resources available for prewarming are reduced. Despite this, WarmServe successfully prewarms the majority of required instances, achieving an average hit ratio of 82% under RPS=25. This high hit ratio confirms the viability of one-for-many prewarming under pressure.

Inference performance. We evaluate the inference performance for all systems by reporting TPOT. Figure 13 shows the cumulative distribution function of TPOT for all systems under α=0.5. We leave the results under α=2.0 to the appendix.

MuxServe's strategy of increasing model parallelism and colocating multiple models on the same GPUs leads to severe inference performance degradation. This is evident under heavy load (RPS=25): while over 60% of requests in autoscaling-based systems achieve a TPOT under 50ms, the overhead from GPU sharing causes over 40% of MuxServe's requests to exceed a TPOT of 100ms. This analysis highlights the fundamental advantage of WarmServe: its prewarming techniques drastically reduce TTFT, and its exclusive allocation of GPU workers preserves inference performance.

Evaluation on AzureCode. Figure 14 and Figure 15 depict the TTFT and TPOT of systems on the AzureCode trace, with α fixed at 0.5. This trace contains fewer requests than AzureConv, making it less predictable. Consequently, WarmServe exhibits a slight performance degradation on this trace.

Figure 14 shows that, on the AzureCode trace, WarmServe achieves a 4.23×–34.52× reduction in P99 TTFT over SLLM-GPU, and a 3.81×–23.34× reduction compared to itself with proactive prewarming disabled. Under a light load (RPS=10), MuxServe's P99 TTFT is slightly lower (0.85× that of WarmServe). However, this marginal benefit disappears under heav-

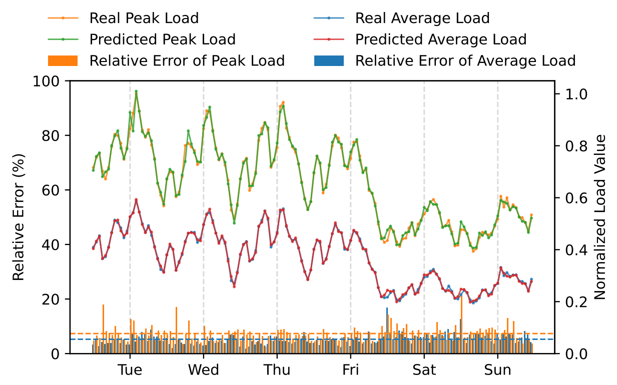


Figure 16: Average relative error of CSP along with the average predicted and actual loads in each hour. The dotted horizontal line represents the mean of relative error.

ier loads, where MuxServe's latency becomes significantly higher. Furthermore, MuxServe fails to provide consistent performance guarantees, incurring an average TPOT that is 3.26× higher than that of WarmServe, as shown in Figure 15.

7.4 Workload Prediction

We evaluate our workload predictor, CSP, by predicting average and peak loads within 5-minute windows on the AzureConv trace. For each window from Tuesday to Sunday, we predict its workload using the historical data of all previous windows. Figure 16 shows the prediction performance along with the actual loads. Across the entire period, CSP demonstrates high accuracy, achieving an average relative error of 5.25% for average loads and 7.34% for peak loads.

Since the AzureConv trace only contains seven days of requests, CSP is limited in capturing daily periodic patterns. However, LLM workloads exhibit significant differences between workdays and weekends [22], leading to slight performance degradation of CSP on weekends. Specifically, the prediction errors for average and peak loads on workdays are 4.60% and 6.99%, whereas they increase on Saturday (6.89% and 8.12%) and Sunday (6.25% and 8.02%). Therefore, we believe that the prediction performance could be further improved by leveraging weekly periodicity from longer traces.

We also evaluate CSP with the AzureCode [23] trace, which is inherently more difficult to predict due to a smaller volume of historical data. On this more challenging workload, CSP achieves an average relative error at 9.58% and 11.16% for average and peak loads, respectively.

7.5 Ablation Study

We conduct an ablation study to validate the effectiveness of evict-aware model placement strategy and proactive prewarming, and analyze the impact of prediction window size. Figure 12 presents the TTFT CDF when disabling either evict-aware model placement or proactive prewarming, or using different window sizes (unit: min). By default, a 5-minute window size is used. The results demonstrate that disabling either evict-aware model placement or proactive prewarming significantly impacts performance, with 0.29× and 0.88× fewer requests receiving the first token in 100ms, respectively.

The choice of window size presents a trade-off: windows that are too small impact prediction accuracy, while those that are too large may fail to capture recent load dynamics. Compared to the default 5-minute window, system performance degrades significantly with 3-minute and 40-minute windows, which serve only 0.46× and 0.30× as many requests under the 100ms TTFT threshold, respectively. A 10-minute window yields comparable performance (1.02×), indicating a robust operational range around 5-10 minutes. In practice, one can select an appropriate window size based on the resulting prediction accuracy.

8 Related Work

Multi-LLM serving. To concurrently serve multiple models in a cluster, existing approaches can be classified into autoscaling systems [12, 15–19] and GPU sharing systems [13, 14, 20, 21]. Autoscaling systems allocate exclusive GPUs for serving instances. Among these systems, caching is widely used to reduce model loading latency [15, 18, 19]. One can also leverage the high-bandwidth network between GPU servers to quickly fetch weights [12, 16]. In network-limited scenarios, distributed fetching is proposed to improve aggregated network bandwidth [17]. GPU sharing systems colocate models on GPUs and allocate computational resources among models according to traffic. The parallelism degree of models is usually increased to make sharing happen among more models. WarmServe adopts the autoscaling approach to serve multiple LLMs, and leverages the long-term predictability of LLM requests to prewarm models in advance.

KV cache management in LLM serving. Many LLM serving systems optimizes the management of KV cache to improve serving performance [25, 29, 49, 54, 55]. For example, vLLM [49] efficiently manages KV cache with the concept of virtual memory in operating systems, allocating and evicting KV blocks on demand. InfiniGen [55] further leverages the Top-K attention to reduce KV calculation. As long sequences require massive KV cache that may exceed the capacity of single GPU, Infinite-LLM [54] and LoongServe [25] propose to distribute the placement of KV cache when serving long requests. Prefix caching also helps improve serving performance, especially when applying workload-specific strategies [29]. WarmServe leverages unused KV cache to store model weights of prewarming models, so that a dedicated GPU worker can quickly transform into a universal one after the GPU has been released.

Serverless prewarming strategies. Prewarming has been widely used to reduce cold start latency in serverless computing [32–35, 56–60]. As serverless applications are usually represented as DAGs, many systems leverage this feature to proactively create containers [56,57] or even execute the function [60]. When functions are not included in a DAG, existing works use a histogram to predict the invocation pattern of functions, and prepare containers in advance [58,59]. Since containers for different functions share similar packages, researchers also enforce one-for-all prewarming that leverages a shared prewarming container for multiple functions [32–34]. Prewarming LLMs are different from serverless functions since LLMs span across multiple GPUs and take longer time to prepare. Moreover, fork-based lazy loading approaches in serverless computing [34, 35] are impractical for LLMs because all weights are required during the first iteration.

9 Conclusion

This paper presents WarmServe, a multi-LLM serving system that adopts one-for-many GPU prewarming to improve inference performance. WarmServe incorporates universal GPU workers to simultaneously prewarm multiple models on the same GPUs, and employs an evict-aware model placement strategy to mitigate prewarming interference. Furthermore, WarmServe proactively prewarms models on dedicated GPU workers to enable seamless transition into universal workers. This prewarming is conducted on grace-period instances that have sufficient free KV cache space. Evaluation results show that WarmServe reduces the tail TTFT by up to 50.8× compared to the state-of-the-art autoscaling-based system, and it successfully operates under up to 2.5× more requests than the GPU-sharing system.

References

[1] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] "GPT-4o System Card," 2025. https://openai.com/index/gpt-4o-system-card/.

[3] DeepSeek-AI, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2025.

[4] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[5] "Anthropic Claude," 2025. https://www.anthropic.com/claude.

[6] "Qwen2.5-Coder Series: Powerful, Diverse, Practical," 2025. https://qwenlm.github.io/blog/qwen2.5-coder-family/.

[7] DeepSeek-AI, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[8] "Introducing OpenAI o3 and o4-mini," 2023. https://openai.com/index/introducing-o3-and-o4-mini/.

[9] "Gemini," 2025. https://deepmind.google/technologies/gemini/.

[10] "Qwen3: Think Deeper, Act Faster," 2025. https://qwenlm.github.io/blog/qwen3/.

[11] "Models - Openai Platform," 2025. https://platform.openai.com/docs/models.

[12] 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," in *USENIX OSDI*, 2025.

[13] J. Duan, R. Lu, H. Duanmu, X. Li, X. Zhang, D. Lin, I. Stoica, and H. Zhang, "Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving," in *ICML*, 2024.

[14] S. Yu, J. Xing, Y. Qiao, M. Ma, Y. Li, Y. Wang, S. Yang, Z. Xie, S. Cao, K. Bao, I. Stoica, H. Xu, and Y. Sheng, "Prism: Unleashing gpu sharing for cost-efficient multi-llm serving," *arXiv preprint arXiv:2505.04021*, 2025.

[15] 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 *USENIX OSDI*, 2024.

[16] M. Yu, R. Yang, C. Jia, Z. Su, S. Yao, T. Lan, Y. Yang, Y. Cheng, W. Wang, A. Wang, and R. Chen, "Lambdascale: Enabling fast scaling for serverless large language model inference," *arXiv preprint arXiv:2502.09922*, 2025.

[17] C. Lou, S. Qi, C. Jin, D. Nie, H. Yang, X. Liu, and X. Jin, "Towards swift serverless llm cold starts with paraserve," *arXiv preprint arXiv:2502.15524*, 2025.

[18] M. Yu, A. Wang, D. Chen, H. Yu, X. Luo, Z. Li, W. Wang, R. Chen, D. Nie, H. Yang, and Y. Ding, "Torpor: Gpu-enabled serverless computing for low-latency, resource-efficient inference," in *USENIX ATC*, 2025.

[19] J. Hu, J. Xu, Z. Liu, Y. He, Y. Chen, H. Xu, J. Liu, J. Meng, B. Zhang, S. Wan, G. Dan, Z. Dong, Z. Ren, C. Liu, T. Xie, D. Lin, Q. Zhang, Y. Yu, H. Feng, X. Chen, and Y. Shan, "Deepserve: Serverless large language model serving at scale," in *USENIX ATC*, 2025.

[20] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *USENIX OSDI*, 2023.

[21] 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 *ACM Symposium on Cloud Computing*, 2024.

[22] Y. Wang, Y. Chen, Z. Li, X. Kang, Z. Tang, X. He, R. Guo, X. Wang, Q. Wang, A. C. Zhou, and X. Chu, "Burstgpt: A real-world workload dataset to optimize llm serving systems," *arXiv preprint arXiv:2401.17644*, 2024.

[23] J. Stojkovic, C. Zhang, I. Goiri, J. Torrellas, and E. Choukse, "Dynamollm: Designing llm inference clusters for performance and energy efficiency," in *IEEE HPCA*, March 2025.

[24] Y. Xiang, X. Li, K. Qian, W. Yu, E. Zhai, and X. Jin, "Servegen: Workload characterization and generation of large language model serving in production," *arXiv preprint arXiv:2505.09999*, 2025.

[25] B. Wu, S. Liu, Y. Zhong, P. Sun, X. Liu, and X. Jin, "Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism," in *ACM SOSP*, 2024.

[26] C. Hu, H. Huang, L. Xu, X. Chen, C. Wang, J. Xu, S. Chen, H. Feng, S. Wang, Y. Bao, N. Sun, and Y. Shan, "Shuffleinfer: Disaggregate llm inference for mixed downstream workloads," *ACM Transactions on Architecture and Code Optimization*, July 2025.

[27] R. Cheng, Y. Lai, X. Wei, R. Chen, and H. Chen, "Kunserve: Efficient parameter-centric memory management for llm serving," *arXiv preprint arXiv:2412.18169*, 2025.

[28] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko, and Y. He, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," *arXiv preprint arXiv:2401.08671*, 2025.

[29] J. Wang, J. Han, X. Wei, S. Shen, D. Zhang, C. Fang, R. Chen, W. Yu, and H. Chen, "Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider," in *USENIX ATC*, July 2025.

[30] 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 *USENIX Conference on File and Storage Technologies*, 2025.

[31] B. Wu, Z. Zhang, Y. Zhong, G. Huang, Y. Zhu, X. Liu, and X. Jin, "Tokenlake: A unified segment-level prefix cache pool for fine-grained elastic long-context llm serving," *arXiv preprint arXiv:2508.17219*, 2025.

[32] A. Sahraei, S. Demetriou, A. Sobhgol, H. Zhang, A. Nagaraja, N. Pathak, G. Joshi, C. Souza, B. Huang, W. Cook, A. Golovei, P. Venkat, A. Mcfague, D. Skarlatos, V. Patel, R. Thind, E. Gonzalez, Y. Jin, and C. Tang, "Xfaas: Hyperscale and low cost serverless functions at meta," in *ACM SOSP*, 2023.

[33] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *ACM ASPLOS*, 2024.

[34] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ACM ASPLOS*, 2020.

[35] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in *USENIX OSDI*, 2023.

[36] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ACM ASPLOS*, 2021.

[37] "Anthropic," 2025. https://www.anthropic.com/.

[38] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters," in *USENIX NSDI*, 2022.

[39] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX OSDI*, 2020.

[40] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," *arXiv preprint arXiv:2305.05920*, 2023.

[41] A. Khare, D. Garg, S. Kalra, S. Grandhi, I. Stoica, and A. Tumanov, "SuperServe: Fine-Grained inference serving for unpredictable workloads," in *USENIX NSDI*, 2025.

[42] "Introducing low-level gpu virtual memory management," 2025. https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/.

[43] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar, "vattention: Dynamic memory management for serving llms without pagedattention," in *ACM ASPLOS*, 2025.

[44] C. C. Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," *International Journal of Forecasting*, 2004.

[45] P. R. Winters, *Forecasting Sales by Exponentially Weighted Moving Averages*. Springer Berlin Heidelberg, 1976.

[46] B. Ghosh, B. Basu, and M. O'Mahony, "Multivariate short-term traffic flow forecasting using time-series analysis," *Intelligent Transportation Systems, IEEE Transactions on*, 2009.

[47] G. E. P. Box and G. M. Jenkins, *Time Series Analysis: Forecasting and Control*. Prentice Hall PTR, 5th ed., 2015.

[48] C. Liu, Z. Jin, J. Gu, and C. Qiu, "Short-term load forecasting using a long short-term memory network," in *ISGT-Europe*, 2017.

[49] 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 *ACM SOSP*, 2023.

[50] "Ray: The AI Compute Engine," 2025. `https://www.ray.io`.

[51] M. Lee, A. Jajoo, and R. R. Kompella, "Enabling elastic model serving with multiworld," *arXiv preprint arXiv:2407.08980*, 2024.

[52] "Llama 2: Open Foundation and Fine-Tuned Chat Models | Research - AI at Meta," 2023. `https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models`.

[53] "CUDA Multi-Process Service." `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[54] B. Lin, C. Zhang, T. Peng, H. Zhao, W. Xiao, M. Sun, A. Liu, Z. Zhang, L. Li, X. Qiu, S. Li, Z. Ji, T. Xie, Y. Li, and W. Lin, "Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache," *arXiv preprint arXiv:2401.02669*, 2024.

[55] W. Lee, J. Lee, J. Seo, and J. Sim, "Infinigen: Efficient generative inference of large language models with dynamic KV cache management," in *USENIX OSDI*, 2024.

[56] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *ACM Symposium on Cloud Computing*, 2021.

[57] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Middleware*, 2020.

[58] X. Cai, Q. Sang, C. Hu, Y. Gong, K. Suo, X. Zhou, and D. Cheng, "Incendio: Priority-based scheduling for alleviating cold start in serverless computing," *IEEE Transactions on Computers*, 2024.

[59] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC*, 2020.

[60] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "Specfaas: Accelerating serverless applications with speculative function execution," in *IEEE HPCA*, 2023.

A TPOT CDF

We provide the TPOT CDF of systems under $\alpha = 2.0$ in Figure 17. MuxServe incurs higher TPOT since it enlarges the parallelism degree of models and limits the computational power of an instance during GPU sharing.

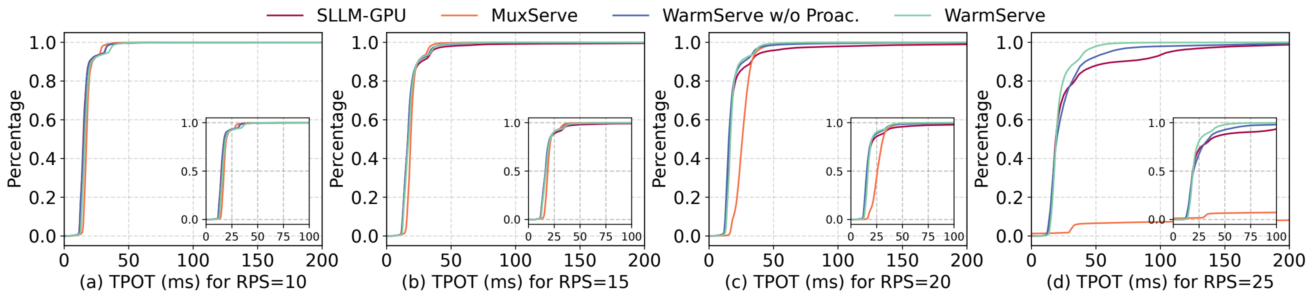


Figure 17: TPOT CDF of systems under α=2.0.