Towards Swift Serverless LLM Cold Starts with ParaServe

Chiheng Lou[†] Sheng Qi[†] Chao Jin[†] Dapeng Nie[‡] Haoran Yang[‡] Xuanzhe Liu[†] Xin Jin[†]
Peking University[†] Alibaba Group[‡]

Abstract

With the surge in number of large language models (LLMs), the industry turns to serverless computing for LLM inference serving. However, serverless LLM serving suffers from significant cold start latency and service level objective (SLO) violations due to the substantial model size, which leads to prolonged model fetching time from remote storage. We present ParaServe, a serverless LLM serving system that minimizes cold start latency through the novel use of pipeline parallelism. Our insight is that by distributing model parameters across multiple GPU servers, we can utilize their aggregated network bandwidth to concurrently fetch different parts of the model. ParaServe adopts a two-level hierarchical design. At the cluster level, ParaServe determines the optimal degree of parallelism based on user SLOs and carefully places GPU workers across servers to reduce network interference. At the worker level, ParaServe overlaps model fetching, loading, and runtime initialization to further accelerate cold starts. Additionally, ParaServe introduces pipeline consolidation, which merges parallel groups back to individual workers to maintain optimal performance for warm requests. Our comprehensive evaluations under diverse settings demonstrate that ParaServe reduces the cold start latency by up to 4.7× and improves SLO attainment by up to 1.74× compared to baselines.

1 Introduction

Large language models (LLMs) have become an essential part of daily work and life, powering a wide range of applications such as chatbots [1–3], programming assistants [4–6], and AI-enhanced search engines [7, 8]. With the widespread adoption of LLMs, the industry has released numerous LLMs with varying structures, sizes, or training corpora for diverse scenarios [1–3, 9–19]. Businesses of different scales are also seeking to leverage custom LLMs tailored to their unique needs [20–23], rather than relying on general-purpose LLM APIs from AI service providers.

Serverless computing [24, 25] has become a compelling choice for users to deploy their LLM serving pipelines [26–29]. In serverless LLM serving, users only need to store the model weights in cloud storage and upload an image that runs LLM services. The cloud platform automatically provisions resources to instantiate the LLM, and dynamically scales the number of workers based on the actual request load. This approach offers significant cost savings through pay-per-use billing, where users are only charged for the resource consumption during active request processing.

However, realizing the serverless paradigm for LLM serving is challenging, primarily due to the long latency of cold starts, i.e., provisioning new workers when there's not enough to meet the current load. Such cold starts are common in serverless workloads due to their bursty traffic patterns, as shown in public traces [30]. As cold starts take place on the critical path of request handling, they directly impact service level objectives (SLOs). The impact is especially pronounced for LLMs that range in size from gigabytes to terabytes. For instance, in our production environment, a cold-start instance produces the first token after up to 40 seconds, whereas subsequent generation takes only ∼30ms per token. This contrast stresses the need for faster cold starts.

In serverless serving, the LLMs are retrieved on demand from remote model registry, which we refer to as model fetching. Serverless functions have limited network bandwidth in nature. As LLMs are large, model fetching costs significant time that contributes most to the total cold start latency. Prior works utilize caching to alleviate the model fetching time [27, 31]. The idea is to cache LLMs in memory or SSDs to accelerate setting up a GPU worker. While caching is effective for the most frequently accessed models, the problem remains unsolved for the long-tail customers, who derive the most benefit from the serverless paradigm [24].

We present ParaServe, a serverless LLM serving system designed to minimize cold start latency. The insight of ParaServe is simple yet effective—whereas a single server has limited bandwidth, we can distribute the workers across different servers, with each fetching a part of the model in parallel. It is viable because LLMs follow a layered structure; the layers are accessed sequentially, obviating the need to load the entire model at once. ParaServe can start the inference once the first few layers are ready on the first server, and forward the intermediate results to subsequent servers that host the rest of the layers. This design, known as *pipeline parallelism*, has been well studied in distributed model training and inference [32–38]. ParaServe applies pipeline parallelism in a novel way to reduce the blocking effect of model fetching.

We take a two-level *hierarchical* approach to realize the parallel model fetching in our production environment. At the *cluster level*, we design an algorithm to compute the appropriate parallelism size (i.e., the number of servers to distribute the model) based on the model size, cluster resource usage, and user SLOs. We then use a network-contention-aware worker placement policy to choose a set of servers to host the pipeline. At the *worker level*, we overlap remote-to-host model fetching, host-to-GPU model loading, and the initial-

ization of container and GPU runtime to further reduce the cold start latency.

Although pipeline parallelism can effectively reduce the cold start latency for the initial inference, it incurs inevitable overheads in subsequent rounds due to the transmissions of intermediate results across workers. To address this problem, we propose *pipeline consolidation*. Specifically, we allow workers to fetch and load the remaining parts of the model in the background, eventually transitioning back to local inference with fully-loaded model. Depending on the load, each participant worker can independently decide whether to release its resources or load the full model after cold starts. Therefore, with a single cold start, ParaServe can scale up as many workers as the pipeline size, allowing for faster elasticity.

In summary, we make the following contributions:

- We design and implement ParaServe, a serverless LLM serving system that leverages pipeline parallelism to reduce cold start latency.
- We adopt a hierarchical approach that combines cluster-level and worker-level optimizations to satisfy user SLOs with pipeline parallelism.
- We propose pipeline consolidation for faster scaling of LLM workers without extra cold starts.
- Experiments on real-world datasets show that ParaServe reduces the cold start latency by up to 4.7× and improves SLO attainment by up to 1.74× compared to baselines.

2 Background and Motivation

In this section, we introduce large language models (LLMs) and serverless LLM serving, and identify opportunities to reduce serverless cold start latency.

2.1 LLM inference

An LLM takes as input a sequence of tokens, called *prompt*, and generates an output sequence in an *autoregressive* fashion, meaning that it generates one token at a time based on both the prompt and its previous outputs. The core component of an LLM is the self-attention layer, which needs to compute the key, value, and query vectors of tokens in the input sequence. Since the key and value vectors of previous tokens remain unchanged during iterations, LLM serving systems usually cache these vectors to avoid redundant computation, known as *KV cache*. The stage that generates the first token is called *prefill*, during which the key and value vectors of all tokens in the prompt are calculated and stored in GPU memory. Subsequent tokens are generated in the *decoding* phase, which reuses the key-value cache and generates one token at a time.

Pipeline parallelism is a model parallelism strategy that is commonly used in LLM training to enable the scaling of model sizes beyond the capacity of a single GPU [32–37]. In this method, different layers of the model are distributed across multiple workers and inter-worker communication is needed to transfer intermediate results. To minimize pipeline bubbles, a mini-batch of training examples is usually split into micro-batches that are executed in pipeline [32, 34]. Pipeline parallelism introduces little communication overhead between workers. For example, running an input with 256 tokens using OPT-175B generates intermediate messages of 6MB, which can be transmitted within single-digit milliseconds over a 10Gbps network.

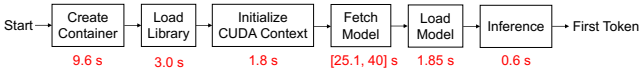


Figure 1: Cold start latency breakdown.

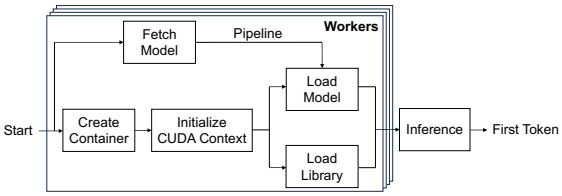


Figure 2: Optimized cold-start workflow.

2.2 Serverless LLM Serving

In serverless LLM serving, users upload (1) LLM models to a model registry, and (2) a function image containing the serving framework and runtime libraries to a container image registry. The serving framework is responsible for fetching the model from the remote registry, loading the model into GPU memory, and running the model to generate tokens upon user requests. The serverless platform automatically adjust the number of function workers to match the current load. One of the most attractive offerings of serverless LLM serving is its pay-per-use billing, where users are charged only for the running period of each worker and not for the idle time. Therefore, serverless is cost-efficient for long-tail customers whose workloads are sporadic and highly unpredictable.

In LLM serving, users often specify service level objectives (SLOs) that represent performance expectations [38–41]. These SLOs are typically made up of two metrics: time to first token (TTFT) and time per output token (TPOT). TTFT represents the time elapsed from user sending the request to the generation of the *first* token, while TPOT is the average time taken to generate each *subsequent* token. Different workloads prioritize these metrics differently. For example, real-time chatbots [1–3] emphasize low TTFT, whereas tasks like article writing prefer lower TPOT [39].

Despite the potential benefits of serverless LLM serving, existing systems suffer from significant cold start latency. A *cold start* occurs when an incoming request finds no available worker hosting the target model, requiring the system to initialize a new worker. Figure 1 provides a breakdown of cold start latency in our production serverless LLM serving platform, using Llama2-7B model on the NVIDIA A10 GPU. In general, a cold start involves the following stages:

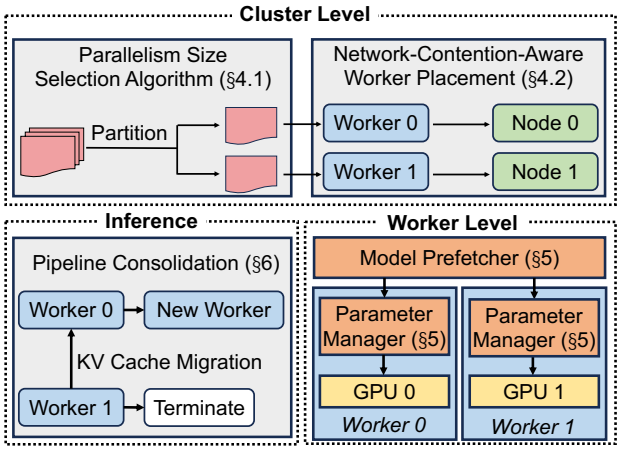
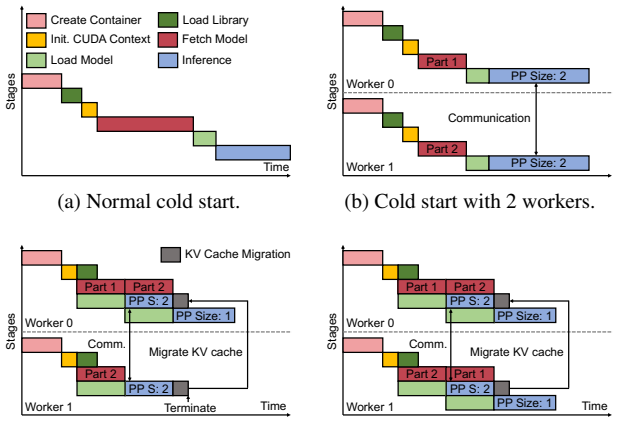


Figure 3: ParaServe system overview.



(a) Normal cold start. (b) Cold start with 2 workers.

(c) Cold start with scaling down. (d) Cold start with scaling up.

Figure 4: Comparison of cold-start workflows for different methods.

- **Container Creation.** The cluster controller allocates resources and creates a container on a GPU server.
- **Library Loading.** The container starts the Python runtime and imports necessary libraries such as PyTorch [42] and TensorFlow [43]. This stage also initializes LLM serving frameworks like vLLM [44], DeepSpeed [45], TensorRT-LLM [46], and Triton [47].
- **CUDA Context Initialization.** The Python runtime set up the CUDA context to prepare for GPU tasks.
- **Model Fetching.** The serving framework retrieves the model from remote storage to local memory.
- **Model Loading.** The fetched model is loaded into GPU memory.
- **Inference.** The serving framework runs the model to generate the first token of the input sequence.

As show in Figure 1, it takes up to 40 seconds to produce the first token for a Llama2-7B model, whereas each subsequent token is generated in ∼30 milliseconds on average. Model fetching is the most time-consuming among all stages since the network bandwidth of inference servers are limited and the model weights are large. Furthermore, network contention can occur when multiple cold start containers are placed on the same server. They compete for the network bandwidth, leading to extremely long fetching latency. Container creation also costs a considerable amount of time, as LLM workloads require large function images.

2.3 Opportunies

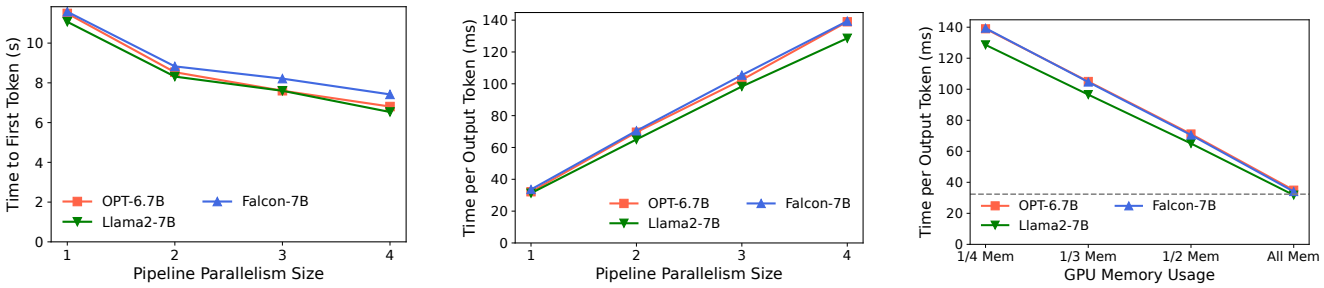
This paper employs a two-level hierarchical approach to achieve maximum overlapping in the cold start critical path for serverless LLM serving. At the cluster level, we leverage pipeline parallelism for coarse-grained overlapping of model fetching across workers, addressing the most time-consuming phase of cold starts. Upon cold start, we create a pipeline parallelism group across GPU servers, with each worker hosting a part of the model. At the worker level, we perform fine-grained overlapping by carefully reorganizing the intra-worker workflow in Figure 1. First, we offload model fetching from user container to a system-level service, allowing the fetching process to begin before container creation. Second, we notice that model and library loading stages are bound by different resources: model loading is GPU-bound, while library loading is CPU-bound. Thus, we run the two stages in parallel. We swap the order of library loading and CUDA context initialization, and trigger model loading once CUDA context has been prepared. Finally, we pipeline the fetching and loading at tensor granularity to hide the loading overhead. After all initialization stages have finished, the worker runs the model and generates the first token, completing the cold start. The optimized cold-start workflow is shown in Figure 2.

3 ParaServe Overview

ParaServe is a serverless LLM serving system designed to minimize the cold start latency. Unlike traditional serverless model serving systems that load the entire model in a single worker, ParaServe leverages parallel model fetching to significantly reduce model fetching time.

As shown in Figure 3, ParaServe consists of two optimization levels. At the cluster level, the central controller employs a parallelism size selection algorithm (§4.1) and a network-contention-aware worker placement strategy (§4.2) to reduce cold start latency through pipeline parallelism. The parallelism size selection algorithm determines the appropriate pipeline parallelism size for each cold-start model based on user SLOs and cluster resource availability. For a given parallelism size, ParaServe partitions the model across multiple workers and assigns these workers to different GPU servers using the network-contention-aware worker placement strategy. This strategy identifies potential network contentions on GPU servers and ensures that such contention does not cause SLO violations.

At the worker level, ParaServe applies fine-grained overlapping across cold start stages to reduce worker initialization time (§5). First, a node-level model prefetcher is used to proactively fetches model weights for cold-start workers, over-



(a) TTFT of different pipeline parallelism sizes.

(b) Impact of pipeline parallelism size on TPOT.

(c) Influence of GPU memory usage on TPOT when pipeline parallelism size is 4. The grey dotted line is the TPOT when pipeline parallelism size is 1.

Figure 5: Tradeoff analysis of pipeline parallelism.

lapping model fetching with container creation and runtime initialization time. Second, ParaServe prioritizes the CUDA context initialization and introduces a parameter manager that loads parameters to GPU in parallel with library loading. Additionally, model fetching and loading are pipelined to hide the model loading overhead. These overlapping strategies reduce the overhead of cold start stages other than model fetching, thereby highlighting the value of pipeline parallelism.

Finally, as pipeline parallelism can degrade inference performance, ParaServe performs pipeline consolidation (§6) to transform pipeline workers into standalone workers that host all parameters. ParaServe allows some of cold-start workers to continue loading the remaining model parts while serving requests. Eventually, these workers evolve into individual workers that host the entire model. Each worker independently decides whether to load additional parameters or terminate after completing the cold start process. Ongoing requests, initially processed in the pipeline parallelism group, are migrated to the standalone workers.

4 Cluster-Level Controller

In this section, we present the design of the cluster-level central controller in ParaServe. The controller makes decision on two aspects: (a) pipeline parallelism size and (b) worker placement. To find the optimal pipeline parallelism size, we first analyze the tradeoffs involved in pipeline parallelism, and then develop an algorithm to select the appropriate parallelism size based on the model size and user SLOs (§4.1). For worker placement, we employ a network-contention-aware worker placement strategy to prevent SLO violations (§4.2).

4.1 Parallelism Size Selection

Figure 4(a) illustrates the normal cold-start workflow, which goes through six stages sequentially. As model fetching is the most time-consuming stage, ParaServe leverages pipeline parallelism to optimize cold starts. As shown in Figure 4(b), upon a cold start, we launch multiple workers on different GPU servers, each fetches a part of the model. During inference, workers exchange intermediate results but the size of these messages is relatively small. By reducing the portion of the model each worker needs to fetch, pipeline parallelism significantly reduces cold start latency.

4.1.1 Tradeoff Analysis

Despite the benefits of pipeline parallelism, it introduces resource and performance overheads. To better understand these tradeoffs, we conduct experiments using a testbed with four GPU servers, each equipped with a NVIDIA A10 GPU and 188GB memory. The network bandwidth of each server is 16Gbps.

Figure 5(a) demonstrates the TTFT of models under different pipeline parallelism sizes. Larger parallelism sizes reduce model fetching time, resulting in shorter TTFT. Note that the marginal improvement on TTFT diminishes as other cold-start stages also consume non-negligible time. This limitation is further tackled by worker-level optimizations in §5.

In terms of TPOT, pipeline parallelism has a negative impact as shown in Figure 5(b). This is because individual workers in a pipeline parallelism group allocate GPU memory proportionally to their partitioned model size by default, and the amount of allocated memory also determines the GPU computation quota in our production environment. Thus, reduced GPU memory leads to severe performance degradation.

Fortunately, users can allocate more GPU resources to workers than the default quota to achieve lower TPOT. Figure 5(c) illustrates the inference performance when different amount of GPU memory is allocated to each worker and pipeline parallelism size is four. Note that the performance degradation would be negligible if each worker in parallelism group is given the same amount of GPU memory as in the non-parallelized case (i.e., parallelism size is one).

In conclusion, larger parallelism size reduces the cold start latency, but impacts the inference performance. Allocating more GPU memory to workers reduces the performance loss but at the cost of higher price. ParaServe designs the pipeline parallelism size selection algorithm based on these two tradeoffs.

Algorithm 1 Pipeline Parallelism Size Selection Algorithm

Input: time cost of container creation and runtime initialization t_c, data transmission t_n, prefill t_p, decoding t_d, and waiting t_w; model size M; GPU server network bandwidth b_i and PCIe bandwidth p_i; user specified requirements SLO_{TTFT} and SLO_{TPOT}.

Output: cold start scheme including pipeline parallelism size s, #full-memory workers w, and selected GPU servers g.

for $s \in \{1,2,\cdots,4\}$ **do**
  for $w \in \{0,1,\cdots,s\}$ **do**
    $i_1, i_2, \cdots, i_k \leftarrow$ Servers that fit a model of size M, sorted by $\frac{1}{b_{i_x}} + \frac{1}{p_{i_x}}$ in increasing order
    $j_1, j_2, \cdots, j_l \leftarrow$ Servers that fit a model of size M/s, sorted by $\frac{1}{b_{j_x}} + \frac{1}{p_{j_x}}$ in descending order
    $j'_1, \cdots, j'_{l'} \leftarrow \text{MergeSort}((j_1, \cdots, j_l), (i_{w+1}, \cdots, i_k))$
    $g \leftarrow (i_1, i_2 \cdots, i_w, j'_1, \cdots, j'_{s-w})$
    $\text{max\_ratio} \leftarrow \max_{x \in g} \left(\frac{1}{b_x} + \frac{1}{p_x}\right)$
    $\text{TTFT} \leftarrow t_w + t_c + \frac{M}{s} \times \text{max\_ratio} + t_p \times (s - w + \frac{w}{s}) + t_n \times s$
    $\text{TPOT} \leftarrow t_d \times (s - w + \frac{w}{s}) + t_n \times s$
    if $\text{TTFT} \leq \text{SLO}_{\text{TTFT}}$ **and** $\text{TPOT} \leq \text{SLO}_{\text{TPOT}}$ **then**
      return (s, w, g)
return $(1, 1, (i_1))$ ▷ Use single worker if no solution

4.1.2 Algorithm Design

The parallelism size selection algorithm takes model size, user SLOs, and cluster resource status as inputs. The cluster resource status includes the remained GPU memory of all servers and their network bandwidth, adjusted by network-contention-aware worker placement policy in §4.2. The algorithm output is a cold start scheme containing the pipeline parallelism size, GPU memory usage of workers, and worker placement.

To narrow down the search space, we limit the possible GPU memory usage of a worker to two cases: (a) the same as the non-parallelized setup (b) proportional to the inverse of pipeline parallelism size. We call them full-memory workers and low-memory workers, respectively.

Our algorithm has two stages. First, we enumerate all choices with different pipeline parallelism sizes or GPU memory usage and predict the TTFT and TPOT for each choice. Second, we select the optimal choice that satisfies user SLOs with minimum resource usage. The maximum pipeline parallelism size is limited to 4 as larger parallelism sizes gain little improvement.

The TTFT and TPOT prediction takes historical information as inputs, including the time cost of container creation and runtime initialization (t_c), data transmission (t_n), prefill (t_p), and decoding (t_d). The data transmission time is the latency of TCP network between GPU servers. The prefill and decoding time cost are model-specific metrics obtained from the model's previous executions. The maximum waiting time of requests for the model is also needed, defined as t_w. Assume the pipeline parallelism size is s and there are w full-memory workers ($0 \leq w \leq s$), while the network and PCIe bandwidth of servers that workers reside in are $\{b_{q_1}, b_{q_2}, \cdots, b_{q_s}\}$ and $\{p_{q_1}, p_{q_2}, \cdots, p_{q_s}\}$. Finally, we predict the TTFT using the following equation:

$$\begin{aligned}\text{TTFT} = t_w + t_c + \frac{M}{s} \times \max_i \{\frac{1}{b_{q_i}} + \frac{1}{p_{q_i}}\} \\ + t_p \times (s - w + \frac{w}{s}) + t_n \times s,\end{aligned} \tag{1}$$

where M is the model size. TTFT is made up of five parts: request waiting, runtime initialization, model fetching, model loading, and prefill. The model fetching time is influenced by the partitioned model size and the slowest worker, while the prefill time depends on parallelism size and number of full-memory workers. A full-memory worker costs $t_p/s + t_n$ units of time in prefill, while a low-memory worker costs $t_p + t_n$ units of time.

To satisfy TTFT requirement, we select GPU servers to minimize the model fetching and loading time for a given pipeline parallelism size s and number of full-memory workers w, Assume the GPU servers that can accommodate full-memory workers are $\{i_1, i_2, \cdots, i_k\}$. Apart from them, there are also servers that can accommodate low-memory workers, denoted as $\{j_1, j_2, \cdots, j_l\}$. Our selection strategy is to first allocate the top w servers with minimum model fetching and loading time (i.e., the smallest $\frac{1}{b_{i_x}} + \frac{1}{p_{i_x}}$) from $\{i_x\}$ to full-memory workers, and then merge the remaining servers into the set $\{j_x\}$. After that, we select the top $s - w$ servers from the merged set by same strategy and allocate them to all low-memory workers.

Similarly, we can predict the TPOT as follows:

$$\text{TPOT} = t_d \times (s - w + \frac{w}{s}) + t_n \times s. \tag{2}$$

Among all choices that satisfy both TTFT and TPOT SLOs, ParaServe selects the one with minimal parallelism size and GPU resource usage. Algorithm 1 presents the pipeline parallelism selection algorithm.

4.2 Network-Contention-Aware Worker Placement

There are two types of network contentions in the cluster. The first one is the contention between model fetching and inference, i.e., the sending of intermediate results across workers. Because the intermediate results are small, simply prioritizing inference packets solves the contention.

The second type of contention is the interference between cold-start workers on the same GPU server. Cold-start workers that collocated on a GPU server will compete for network resources during model fetching, leading to unexpected cold start performance. In contrast, simply assigning cold-start workers to servers in a mutually exclusive manner results in low GPU resource utilization. To this end, we propose a

Algorithm 2 Network-Contention-Aware Worker Placement Policy

Input: each worker's deadline given by TTFT $\{D_i\}$ and estimated pending model size $\{S_i\}$

function GETNODEBANDWIDTH($node, T$)
  $B \leftarrow$ bandwidth of $node$
  $w_1, w_2, \cdots, w_N \leftarrow$ existing cold-start workers of $node$
  for $w \in (w_1, w_2, \cdots, w_N)$ **do**
    if $S_w > \frac{B}{N+1} \times (D_w - T)$ **then**
      return 0
  return $\frac{B}{N}$

function HANDLEBANDWIDTHCHANGE($node, T$)
  $B \leftarrow$ bandwidth of $node$
  $T' \leftarrow$ last T that changed bandwidth of $node$
  $w_1, w_2, \cdots, w_N \leftarrow$ existing cold-start workers of $node$
  for $w \in (w_1, w_2, \cdots, w_N)$ **do**
    $S_w \leftarrow S_w - \frac{B}{N} \times (T - T')$
    if $S_w \leq 0$ **then**
      Remove w from cold-start workers of $node$
  $T' \leftarrow T$

network-contention-aware worker placement policy that identifies network contentions and places cold-start workers based on user SLOs.

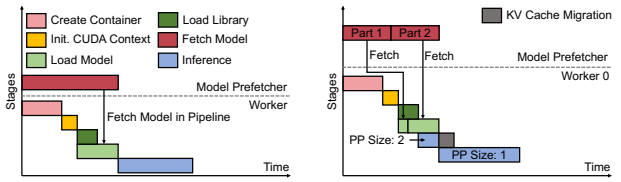
4.2.1 Policy Design

Initially, collocated workers share the network bandwidth with equal credits. Once we assign a worker to a GPU server, we record the model size it needs to fetch and the user-specified maximum TTFT. To place a new worker, we inspect each GPU server to check that whether adding a cold-start worker to this server would lead to SLO violations for existing workers. If check passed, we estimate and record the network bandwidth that the new worker can obtain. Note that PCIe bandwith is much higher than network and PCIe switch is able to further isolate PCIe usage across tasks, so we do not take PCIe contention into account.

Specifically, we denote the bandwidth of a GPU server as B. For each cold-start worker i on the GPU server, we record the fetching deadline D_i and estimate its pending model size S_i. The fetching deadline comes from the prediction of TTFT (Eq. 1). When a new cold-start worker comes, the network bandwidth would change to $B/(N+1)$ if there were N cold-start workers on the server. We check whether all cold-start workers on the server are able to complete fetching before deadline under the new bandwidth. Formally, we check that whether the following condition is true for all workers:

$$S_i \leq \frac{B}{N+1} \times (D_i - T), \tag{3}$$

where T represents current time. This server accepts the new worker if all workers passed the check.



(a) Prefetching whole model. (b) Prefetching in two stages.

Figure 6: Cold-start workflows with worker-level overlapping.

To estimate the pending model size, we record the time of last network bandwidth change T'. The start and completion of a cold start change the network bandwidth, informing controller to adjust the pending model size. Here, we assume the current time is T and the number of cold-start workers before change is N. Then we adjust the pending model size upon each bandwidth change according to the following equation:

$$S_i' = S_i - \frac{B}{N} \times (T - T'). \tag{4}$$

After adjustment, a S_i that is below zero means that the worker has fetched the model ideally, thus we delete it from the cold-start worker list. Algorithm 2 outlines the pseudocode of obtaining server bandwidth and handling bandwidth change.

5 Worker-Level Overlapping

As described in §4.1.1, the TTFT of pipeline parallelism has diminishing marginal returns due to container initialization stages, including container creation, library loading and CUDA context initialization. Container creation is slow because function images of LLM workloads usually contain multiple Python libraries such as PyTorch [42], TensorFlow [48], TensorRT [46] and vLLM [44]. These Python libraries have complex dependencies, leading to large image sizes. For example, the basic image for running vLLM is around 8.31GB. Loading and initializing numerous libraries also cost long libray loading time. Preparing a universal container for all workers [49–51] does not help in this case because users have requirements for different library versions.

In this section, we present the worker-level overlapping in ParaServe that carefully reorganizes the startup workflow to reduce worker initialization time. Basically, ParaServe leverages two optimizations to overlap initialization stages. First, we prefetch model weights on local GPU server as soon as container creation starts so that the container initialization stages are overlapped with model fetching. Second, we prioritize CUDA context initialization and develop a parameter manager to load model while initializing Python libraries so that the library loading stage is overlapped with model loading.

5.1 Model Prefetching

ParaServe launches a model prefetcher on each GPU server, which is responsible for proactively fetching models from

remote storage. When a worker has been allocated to the server, the central controller informs the model prefetcher about model metadata. After that, the prefetcher starts to load the model weights from remote storage to a shared memory region. The cold-start worker will fetch parameters from shared memory in a streaming manner after runtime has initialized.

Model prefetching starts before container creation, so that the container creation and runtime initialization stages are overlapped. The worker performs model prefetching and loading in a pipelined fashion. In the shared memory region of a model, we use the first eight bytes to store the address that represents the end of currently fetched model weights. Model weights are represented using SafeTensors format [52]. SafeTensors format contains the metadata of all parameters at the beginning of the file, so that it is convenient for the worker to check whether a tensor has been fetched.

As allocating shared memory is time-consuming, the model prefetcher allocates a shared memory region for all models in advance. During startup, it accesses each virtual page in the region to allocate corresponding physical pages. When a fetching request arrives, the model prefetcher calculates the size of all model files and allocates space from the shared memory region. A standalone process is then triggered to read the model weights from remote storage and write contents into shared memory.

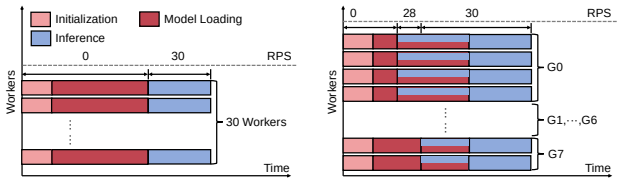
5.2 Parameter Manager

ParaServe leverages a parameter manager to load model parameters in the background. The parameter manager runs in an individual thread and is responsible for resolving tensor metadata, reading weights from the shared memory, and finally loading weights into GPU. The whole procedure is zero-copy and pipelined. The parameter manager also takes advantage of the high parallelism of GPU cores and uses multiple CUDA streams to load models. The priorities of CUDA streams are determined according to whether the loading process is on the critical path or in the background.

During cold starts, the entry program in container will first initialize the parameter manager to start loading model parameters, and then import other AI libraries. The serving framework later queries the parameter manager through a specified API to obtain tensors in a streaming manner with zero copy.

Figure 6(a) illustrates the optimized cold-start workflow after adopting model prefetching and parameter manager. Model fetching and container initialization runs in parallel, while library loading is overlapped with model loading. In this way, the overhead of other cold start stages is concealed behind model fetching, reinforcing the effectiveness of pipeline parallelism.

As ParaServe may launch workers in pipeline parallelism groups and later load the remaining part of model in background, Figure 6(b) also shows the cold-start workflow in this scenario. The model prefetcher downloads two parts of model sequentially. After the first part of model has been loaded, the worker starts to perform inference with other workers in the pipeline parallelism group. Meanwhile, the parameter manager loads the second part of model in background and finally transform itself into a standalone worker.



(a) Normal scaling. (b) Scaling up in ParaServe.

Figure 7: Maximum request per second (RPS) over time for different scaling policies when creating 30 new workers. Assume each worker can handle 1 request per second.

TTFT prediction with worker-level overlapping. After applying worker-level optimizations, the TTFT is reduced by fine-grained overlapping. Based on additional historical information including the time cost of container creation (t_{cc}), CUDA context initialization (t_{cu}) and library loading (t_l), we change the TTFT prediction used in pipeline parallelism size selection (Eq. 1) into the following formula.

$$\begin{aligned}\text{TTFT} = t_w + \max_i \left(t_{cc} + t_{cu} + \max\left(\frac{M/s}{p_{q_i}}, t_l\right), \frac{M/s}{b_{q_i}} \right) \\ + t_p \times (s - w + \frac{w}{s}) + t_n \times s.\end{aligned} \tag{5}$$

6 Pipeline Consolidation

ParaServe proposes pipeline consolidation that merges pipeline parallelism groups into individual workers to obtain optimal performance. Specifically, after the parameter manager has loaded the requested model parameters in pipeline-parallel inference, we inform it to continue loading the remaining part of model in background. Once loading completes, the worker returns to the non-parallelized setup and serves subsequent requests with whole model in local GPU, thereby achieving optimal performance. Note that the parameter manager loads the second part of model in low-priority CUDA streams, so that the performance of inference task will not be affected.

6.1 Worker Scaling

ParaServe provides two scaling choices. First, after workers have loaded corresponding parts of models, we can perform *scaling down* by allowing one of them to fetch the unloaded model parts in background. Once all parameters are loaded, we migrate all existing requests to that worker by migrating their key-value cache. Finally, the worker continues to generate tokens with whole model while other workers are terminated. Figure 4(c) demonstrates the process of scaling down. In this way, we obtain the first token earlier by parallelized model fetching and finally obtain a worker with whole model, similar to normal cold starts.

The second scaling choice is *scaling up*, changing all cold-start workers into individual serving endpoints, as Figure 4(d) has shown. LLM inference workloads usually have bursty traffic [27, 38, 53]. In serverless LLM serving, bursty requests often cause the system to concurrently create multiple workers in the cluster. In this scenario, the scaling up method allows for quickly creating workers in pipeline parallelism groups, so that the system can reach the maximum throughput earlier.

Figure 7 uses an example of creating 30 new workers concurrently to show the benefits of scaling up. When a burst of requests arrive, existing serverless LLM serving systems initialize new workers concurrently, resulting in significant network and memory consumption in the cluster. Additionally, workers are not able to serve requests until the whole model has loaded, as shown in Figure 7(a). In contrast, ParaServe with scaling up is able to prepare pipeline parallelism groups quickly. Figure 7(b) demonstrates the startup strategy of ParaServe, which allocates workers to 8 pipeline parallelism groups, where one group has two workers and others have four. In this way, workers are able to generate tokens upon loaded part of the model, thereby starting to serve requests much earlier. Since pipeline parallelism has comparable throughput with same number of workers that hold the whole model [38], the system can achieve the maximum possible throughput upon the creation of all groups. Furthermore, workers continue to load models in background, and eventually turn into individual workers to reduce inference latency. In other words, the system is able to achieve high throughput earlier and finally obtains same number of workers with normal cold starts.

In default, ParaServe adopts the scaling down mechanism to reduce cold start latency with little overhead. To quickly react to bursty loads and change to the scaling up mechanism, we determine the number of workers to remain with a sliding window strategy. For each model, we record the number of requests that arrived during the past window and use it as the predicted maximum number of requests that may arrive in the next window. The required number of workers to retain is calculated based on the current number of requests in the waiting queue combined with the predicted maximum number of requests expected to arrive in the next window. In cold start, we create a pipeline parallelism group that is no smaller than required number of new workers and later transform the group into desired number of individual workers. We may create multiple pipeline parallelism groups to serve a sudden burst of requests.

6.2 Key-Value Cache Migration

After reducing the pipeline parallelism size, we migrate all uncompleted requests to one worker for better performance, while allowing for earlier release of resources occupied by other workers. Since model layers are assigned to different workers, the KV cache of all requests is distributed across workers, necessitating the KV cache migration.

ParaServe performs KV cache migration inside a pipeline parallelism group, where the KV cache of layers is distributed across workers. Before migration, we first stop scheduling of existing requests and wait for all on-the-fly batches return. Next, we query the cache block manager to obtain the blocks that are used by existing requests, and then collect these blocks from all workers using a *gather* primitive in collective communication. Blocks are gathered to the first worker and placed at different layers, according to which worker it comes from.

The KV cache migration process in ParaServe is highly optimized. First, the whole migration workflow is performed in an individual thread and uses low-priority CUDA streams so that the inference tasks will not be affected. Second, we create multiple CUDA streams when moving data from or to GPUs to utilize the GPUs' high parallelism. Furthermore, the data transmission is performed in a streaming manner. On the target worker, once a chunk of tensors arrives, it is instantly loaded to GPU in a separate CUDA stream. On other workers, once a chunk of tensors is loaded from GPU to host, we immediately send them to the network.

7 Implementation

We implement ParaServe on top of vLLM [44]. We modify vLLM and add around 3200 lines of code in C++ and Python. We also develop a serverless LLM serving framework with around 3000 lines of code in Python. The serving framework performs request routing and automatically scales number of vLLM workers for each model to match current loads. To improve the cold start performance, we run inference in eager mode that does not use CUDA graphs.

Parameter Manager. We develop parameter manager as a TorchScript [54] C++ library. The entry Python program in container first loads the library and triggers the loading thread, which accesses the shared memory created by model prefetcher and loads the model into GPU. The vLLM engine later queries the loading thread and obtains loaded tensors in order.

Engine Initialization Optimizations. We perform several optimizations to the initialization process of vLLM. First, vLLM allocates CPU key-value cache as swapping space during initialization, costing non-negligible time. Thus, we postpone the CPU cache initialization phase. Second, vLLM runs a random input for the model upon loading completes to profile the amount of free memory during inference and calculate the number of key-value cache blocks. We calculate the amount of free memory based on number of model layers and shapes of intermediate results, thereby skipping the profiling phase. Third, vLLM first initializes the dummy model on CPU, and then substitute existing tensors with newly loaded tensors, and finally transmit these tensors to GPU. We allow the engine to directly employ GPU tensors provided by the parameter manager.

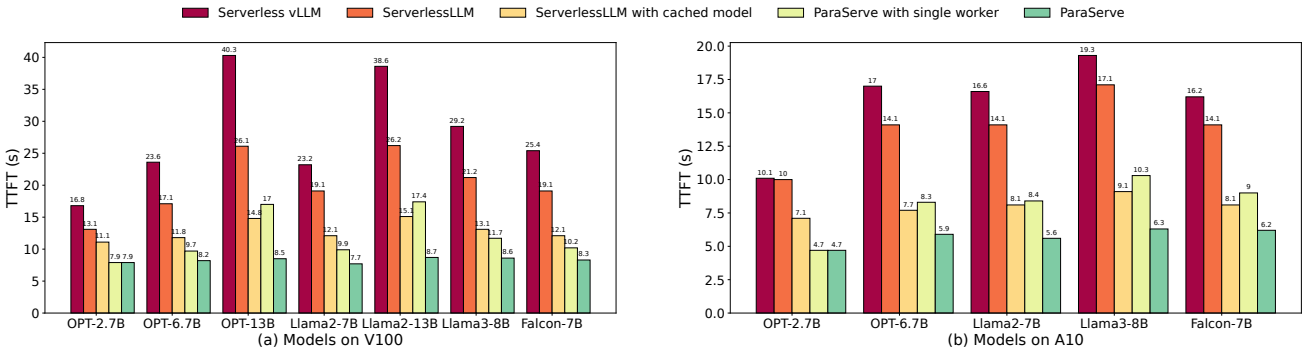


Figure 8: Cold start latency of systems for different models.

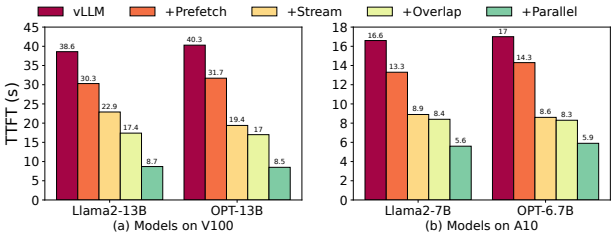


Figure 9: Performance breakdown of techniques in ParaServe.

8 Evaluation

In this section, we present experimental results to validate the efficiency and effectiveness of ParaServe. Our evaluation demonstrates that ParaServe achieves a substantial reduction in TTFT, outperforming baselines by up to 4.7× (§8.2). In end-to-end experiments, ParaServe improves TTFT SLO attainment by up to 1.74× across various loads and constraints while maintaining minimal TPOT SLO violations (§8.3). Furthermore, our analysis of pipeline consolidation reveals that scaling down reduces end-to-end generation time by 2.29×, while scaling up lowers the average TTFT under bursty requests by 1.87× (§8.4). Lastly, we wrap up with brownfield results and show that ParaServe achieves an average TTFT reduction of 2.6× in production environment (§8.5).

8.1 Experiments Setup

Testbed. We have two testbeds: (i) a GPU cluster that contains 4 A10 servers and 4 V100 servers. Each A10 server has a single NVIDIA A10 GPU and 188GB memory, while each V100 server has four NVIDIA V100 GPUs and 368GB memory. The network bandwidth per server is 16Gbps. (ii) a GPU cluster consisting of 2 servers, each equipped with four NVIDIA A10 GPUs, and another 4 servers, each equipped with four NVIDIA V100 GPUs. The A10 servers have 752GB memory and a network bandwidth of 64Gbps per server, while the V100 servers have 368GB memory and a network bandwidth of 16Gbps per server. Both testbeds are connected to a remote storage that hosts the models. The remote storage has sufficient network capacity and the model fetching bandwidth is constrained only by the bandwidth of GPU servers. Since our GPU servers do not have NVLink [55], all evaluated models are able to reside in single card to avoid performance degradation.

Model	Model Size	GPU Card	TTFT	TPOT
Llama2-7B	12.5GB	A10	1.5s	42ms
Llama2-13B	24.2GB	V100	2.4s	58ms

Table 1: Measured TTFT and TPOT of warm requests.

Application	TTFT	TPOT	Dataset
Chatbot Llama2-7B	7.5s	200ms	ShareGPT [57]
Chatbot Llama2-13B	12s	200ms	ShareGPT [57]
Code Completion Llama2-7B	7.5s	84ms	HumanEval [58]
Code Completion Llama2-13B	12s	116ms	HumanEval [58]
Summarization Llama2-7B	15s	84ms	LongBench [59]
Summarization Llama2-13B	24s	116ms	LongBench [59]

Table 2: Summary of applications in end-to-end experiments.

Baselines. We compare ParaServe against two baselines:

- Serverless vLLM. vLLM [44] is a LLM serving engine and we equip it with the same serverless LLM serving framework with ParaServe. During a cold start, the scheduler iterates through all GPU servers and selects the first one with sufficient GPU resources to host the worker. For a fair comparison, we disable CUDA graph in serverless vLLM.
- ServerlessLLM [27]. ServerlessLLM is a state-of-the-art serverless LLM serving system that reduces the cold start latency by loading-optimized checkpoint and caching. Due to the lack of high-speed SSDs in our testbed, we can only allow ServerlessLLM to leverage all free memory on servers to cache models. We use Kubernetes [56] to deploy ServerlessLLM and the containers are created at the beginning, so there will be no container creation overhead during serving. We also configure ServerlessLLM to use vLLM [44] as the serving backend and the CUDA graph is disabled.

8.2 Cold Start Latency

We first evaluate the cold start latency of different systems on testbed (i). We conduct experiments on both V100 and A10 GPUs and configure ParaServe to use a parallelism size of 4. For ServerlessLLM, we measure its performance with and without model caching. Figure 8 illustrates the time to first token for different models. The results demonstrate that

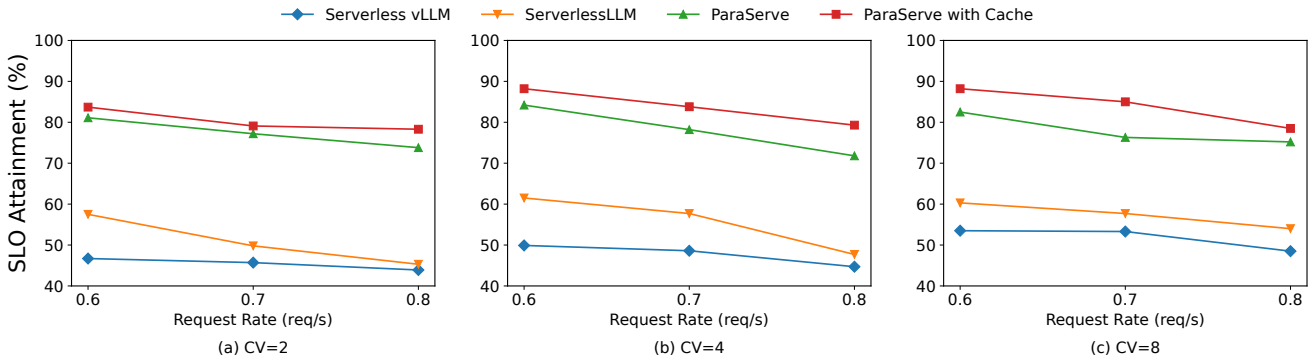


Figure 10: TTFT SLO attainment of systems under different CVs.

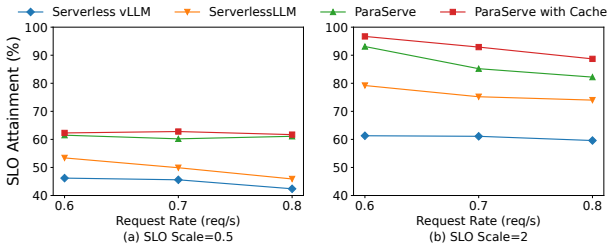


Figure 11: TTFT SLO attainment of systems under different SLOs.

ParaServe achieves the shortest cold start latency for all models. Specifically, ParaServe reduces cold start latency by up to 4.7× compared to serverless vLLM and by up to 3.1× to ServerlessLLM. This performance improvement is attributed to parallelized model fetching and worker-level optimizations.

Furthermore, when ParaServe is constrained to use single worker, it also obtains better performance than ServerlessLLM. For smaller models, such as OPT-6.7B, ParaServe with single worker even achieves a short cold start latency than ServerlessLLM with cached models. This improvement primarily comes from ParaServe's worker-level overlapping strategies and the optimizations applied to vLLM's cold-start process.

Performance Breakdown. To better comprehend the performance of ParaServe, we conduct a detailed breakdown of techniques employed in ParaServe. Figure 9 illustrates the incremental improvement achieved by applying each proposed technique. Starting with the original vLLM system, we apply the following techniques step-by-step: model prefetching (+Prefetch), streaming loading and implementation optimizations (+Stream), overlapped model loading and library initialization (+Overlap), and parallelized model fetching (+Parallel). The results show that each technique contributes to a reduction in cold start latency and the cumulative effect of all techniques results in a substantial overall improvement.

8.3 End-to-End Experiments

We further evaluate the effectiveness of ParaServe through comprehensive end-to-end experiments.

Workloads. We choose the Llama2 model series [16] with FP16 precision in the end-to-end experiments. Following prior work [39], we use three typical LLM applications in experiments: chatbot, code completion, and summarization. Requests for these applications are sampled from ShareGPT [57], HumanEval [58] and Longbench [59], respectively. Since there is no available SLO settings for these applications, we derive SLOs based on the performance of warm requests. Specifically, we measure the TTFT and TPOT for warm requests using Llama2-7B and Llama2-13B, with each request containing 1024 input tokens and a batch size of 8. The results are shown in Table 1. The global TTFT SLO is set to five times the TTFT of warm requests, while the TPOT SLO is defined as a stricter constraint, being twice the TPOT of warm requests.

Given the nature of summarization tasks, which typically allow more relaxed latency requirements, their TTFT SLOs are doubled. Additionally, for chatbot tasks, the TPOT SLO is aligned with standard human reading speeds (i.e., 300 words per minute). Finally, we generate 64 instances for each application to represent various user models, similar to prior work [27, 38]. Table 2 shows the summary of applications.

We leverage Microsoft Azure Function Trace [30] to generate workloads, similar to the approach taken by prior works [27, 38]. Models are mapped to functions in the trace using a round-robin approach, and the requests are sampled from the trace using a Gamma distribution. We manage the sampling process by controlling the coefficient of variance (CV) and request rate per second (RPS).

Effectiveness under different CVs. Figure 10 demonstrates the TTFT SLO attainment of the systems under different CVs. To assess the effectiveness of cache, we also evaluate ParaServe with caching enabled. The results indicate that as RPS increases, TTFT SLO attainment decreases due to resource lacking during bursty requests. Nevertheless, ParaServe consistently satisfies the majority of TTFT SLO requirements under heavy loads, achieving up to 1.74× higher compared to baselines in all scenarios. This is because ParaServe select appropriate pipeline parallelism size based on user SLOs and distribute cold-start workers to miti-

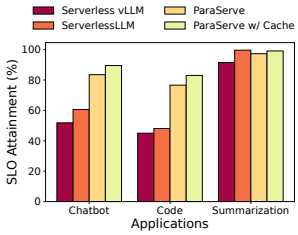


Figure 12: TTFT SLO attainment for different applications.

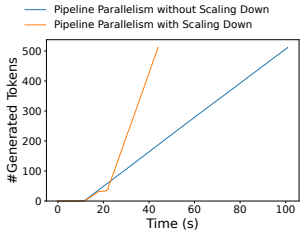
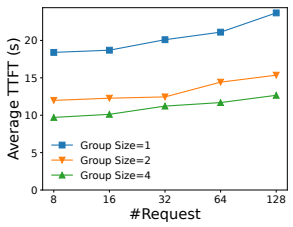
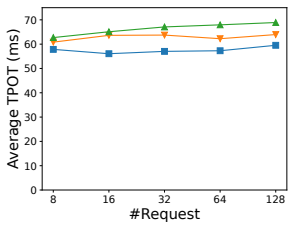


Figure 13: Total tokens generated over time for a single request.



(a) Average TTFT of different loads.



(b) Average TPOT of different loads.

Figure 14: Performance comparison for handling bursty loads with different parallel group sizes.

gate network contention. In contrast, ServerlessLLM exhibits high SLO violations due to limited effectiveness of caching for long-tail models. Enabling cache in ParaServe further improves the TTFT SLO attainment by up to 1.11×, as frequently accessed models benefit from caching.

For TPOT SLO attainment, both ParaServe and baselines achieve over 95% attainment in most scenarios, and more than 90% under all CVs and RPS. This level of performance is sufficient for most use cases.

Effectiveness under different SLO scales. We evaluate the systems under different TTFT and TPOT SLOs by adjusting a global SLO scaling parameter, with CV fixed at 8. As shown in Figure 11(a), under tight SLO conditions, all systems experience significant SLO violations, as the time granted for preparing a cold-start worker is exceedingly limited. In such cases, the TTFT SLO attainment of all systems is capped at 63%. However, ParaServe still outperforms baselines, as its faster worker initialization reduces the waiting time for subsequent requests, even if the first request violates SLO. Figure 11(b) demonstrates the performance under looser SLO conditions. ParaServe achieves up to 1.52× improvement in TTFT SLO attainment compared to baselines, with caching further enhancing this improvement to 1.58×.

Application Analysis. We also analyze the performance of systems for different applications. Figure 12 illustrates the TTFT SLO attainment of chatbot, code completion, and summarization applications under CV=8 and RPS=0.6. First, the results show that ParaServe significantly enhances the TTFT SLO attainment for chatbot and code application, with up to 1.61× and 1.70× improvement, respectively. The code application has lower SLO attainment compared to others

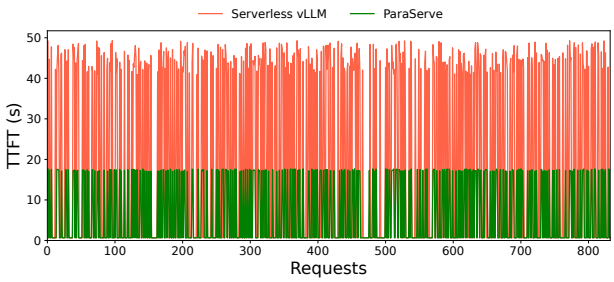


Figure 15: TTFT of requests in brownfield evaluation.

because code completion tasks (i.e., requests in HumanEval dataset [58]) have shorter average output length than chat tasks (i.e., requests in ShareGPT dataset [57]) [39]. Therefore, workers for code completion models keep alive for shorter time, leading to more cold starts. Since the summarization application has loose SLO requirements, it meets few SLO violations for all systems.

8.4 Pipeline Consolidation

To evaluate the effectiveness of pipeline consolidation, we conduct experiments to analyze ParaServe's performance using two scaling methods. We deploy Llama2-13B on V100 servers in testbed (i) and set the input and maximum output length of a request to 512 tokens.

Scaling down. We demonstrate the benefits of scaling down using a single cold start request that generates 512 tokens. We set pipeline parallelism size to 4 and record the generation time of each token, as shown in Figure 13. With scaling down, the system continues loading the remaining parts of the model in parallel with inference, and migrates the key-value cache of the ongoing request once loading is complete. This allows subsequent tokens to be generated at a faster speed. As a result, scaling down reduces the end-to-end generation time by 2.29×, while maintaining almost same inference speeds during the early cold-start period.

Scaling up. We evaluate scaling up by measure ParaServe's performance under bursty workloads. We set the maximum batch size for each worker to 8 and vary the number of incoming requests. Figure 14(a) illustrates the average TTFT across different loads. The results show that larger pipeline parallelism sizes significantly reduce TTFT, enabling the system to increase throughput earlier. For example, when handling 128 concurrent requests (the maximum load for 16 V100 GPUs), using four workers in a pipeline parallelism group reduces the average TTFT by 1.87×. Furthermore, Figure 14(b) indicates that scaling up incurs little inference overhead, with the average TPOT increasing by 1.08-1.19×. This increase is attributed to transmission overhead of intermediate results.

8.5 Brownfield Evaluation

We implement ParaServe in the production environment and evaluate its effectiveness. Specifically, we deploy multiple

serverless functions for cold-start instances while incorporating the parameter manager and engine optimizations. Due to security constraints in the production environment, functions cannot establish direct TCP connections with one another. To address this, we leverage a shared object in remote storage to enable inter-worker communication. For the evaluation, we use vLLM to run Llama2-7B on NVIDIA A10 GPUs with 24GB GPU memory. Requests are generated following Microsoft Azure Function Trace [30]. The bandwidth per function is limited to 5Gbps and each instance is granted a grace period of 5 minutes. Figure 15 shows the cold start latency improvement achieved by ParaServe. The results show that ParaServe significantly reduces cold start latency compared to the original vLLM system, with an average cold-start TTFT reduction of 2.6×. This substantial performance improvement stems from ParaServe's parallelized model fetching and optimized worker initialization processes.

9 Discussion

Deploying larger models. ParaServe is applicable to both models that reside in single GPU server as well as models that are deployed in multiple servers with tensor and pipeline parallelism. For models that are distributed across GPU servers, we can improve their cold-start performance through enlarging the pipeline parallelism size. By controlling the degree of parallelism, we reduce cold start latency to satisfy user SLOs. Moreover, worker-level overlapping and pipeline consolidation can also be utilized in this context.

Maximum request capacity of ParaServe. For each cold-start instance, ParaServe allows it to use more workers or GPU resources than needed to reduce cold start latency, probably decreasing the maximum cold-start requests that the system can serve. However, this issue is effectively mitigated in ParaServe. First, increasing pipeline parallelism size actually improves cluster-level resource utilization by enabling finer-grained GPU memory allocation. Second, for users who allocate extra GPU resources to cold-start workers , these resources are quickly released by adopting pipeline consolidation, so the overhead is minimum. Finally, although pipeline parallelism introduces additional host memory usage due to multiplexed Python runtime, this does not block the launching of workers since host memory is usually adequate in GPU servers. Our end-to-end experiments in §8.3 also validate that ParaServe can achieve high SLO attainment under heavy loads, while baselines cannot.

10 Related Work

Cold-start optimizations in serverless model serving. There have been plenty of works on reducing the cold start latency in serverless model serving [27, 31, 60–62]. For instance, FaaSwapfaaswap and ServerlessLLM [27] utilize locality by caching models in local memory or SSDs, while INFless [61] prewarms instances based on historical request patterns. gCROP [63] introduces a on-demand GPU image restore technique that accelerates GPU runtime initialization. Unlike these works, ParaServe primarily targets reducing model fetching latency by aggregating network bandwidth of GPU servers.

Pipeline parallelism. Pipeline parallelism has been widely used to scale GPU resources for training large models [32–37]. Recently, researchers have also explored its use during inference [38, 64–66]. AlpaServe [38] observes that model parallelism improves GPU utilization under bursty workloads and introduces a model placement policy when deploying models. HPipe [64] and PipeEdge [66] apply pipeline parallelism to improve the inference performance for LLMs on edge devices. PipeInfer [65] explores speculative inference through asynchronous pipeline parallelism to reduce inference latency. ParaServe leverages pipeline parallelism to reduce cold start latency in serverless LLM serving and further introduces pipeline consolidation, which dynamically scales works in a pipeline parallelism group to improve performance.

LLM serving optimizations. Many LLM serving optimizations have been proposed to improve serving performance [27, 38–41, 44, 67–70]. In the area of request scheduling, Orca [68] introduces iteration-level scheduling to run more requests in parallel, while Llumnix [40] employs runtime scheduling to meet user SLOs. FastServe [41] focuses on average job completion time and introduces a preemptive scheduling policy. For key-value cache management, vLLM [44] leverages the concept of virtual memory in operating systems and InfiniGen [69] optimizes KV block placement to improve inference efficiency. Additionally, to mitigate the interference between prefill and decoding phases, DistServe [39] proposes disaggregated inference, whereas SARATHI [70] adopts chunked prefill. ParaServe specifically focuses on reducing cold start latency in serverless LLM serving. Our techniques are orthogonal to existing inference optimizations and we can seamlessly integrate new optimizations into ParaServe.

11 Conclusion

This paper presents ParaServe, a serverless LLM serving system designed to reduce cold start latency through parallelized model fetching. ParaServe adopts a two-level hierarchical approach to accelerate cold starts, and consolidates pipeline workers into individual workers to ensure high performance for warm requests. Evaluation results show that ParaServe reduces cold start latency by up to 4.7× compared to baselines and improves TTFT SLO attainment by up to 1.74× under various constraints. By integrating cluster-level and worker-level optimizations, ParaServe offers a scalable and efficient solution to meet user-defined SLOs in serverless LLM serving, particularly for latency-sensitive applications.

References

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

[2] "Introducing Claude," 2024. https://www.anthropic.com/index/introducing-claude.

[3] "Introducing claude 2," 2024. https://www.anthropic.com/index/claude-2.

[4] "GitHub Copilot," 2024. https://github.com/features/copilot.

[5] "Cursor - The AI Code Editor," 2024. https://www.cursor.com.

[6] "Amazon Q Developer," 2024. https://aws.amazon.com/q/developer/.

[7] "Introducing microsoft new bing," 2024. https://news.microsoft.com/the-new-Bing/.

[8] "Chatgpt plugin: Browsing," 2024. https://openai.com/blog/chatgpt-plugins#browsing.

[9] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Neural Information Processing Systems*, 2020.

[11] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Alhammadi, M. Daniele, D. Heslow, J. Launay, Q. Malartic, B. Noune, B. Pannier, and G. Penedo, "The falcon series of language models: Towards open frontier models," 2023.

[12] "Gemma," 2024. https://blog.google/technology/developers/gemma-open-models/.

[13] "Gemma 2," 2024. https://blog.google/technology/developers/google-gemma-2/.

[14] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[15] "Introducing Llama," 2023. https://research.facebook.com/publications/llama-open-and-efficient-foundation-language-models.

[16] "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.

[17] "Gemini," 2024. https://deepmind.google/technologies/gemini/.

[18] "Mistral," 2024. https://mistral.ai/.

[19] "Introducing Phi-3: Redefining what's possible with SLMs," 2024. https://azure.microsoft.com/en-us/blog/introducing-phi-3-redefining-whats-possible-with-slms/.

[20] J. E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations (ICLR)*, 2022.

[21] "Understanding custom llm models: A 2024 guide," 2024. https://botpenguin.com/blogs/understanding-custom-llm-models.

[22] "Building domain-specific llms: Examples and techniques," 2024. https://kili-technology.com/large-language-models-llms/building-domain-specific-llms-examples-and-techniques.

[23] "Getting started with customizing a large language model (llm)," 2024. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/customizing-llms.

[24] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *IEEE International Conference on Distributed Computing Systems Workshops*, 2017.

[25] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2017.

[26] "Huggingface," 2024. https://huggingface.co.

[27] 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.

[28] "Aws sagemaker," 2024. https://docs.aws.amazon.com/sagemaker/.

[29] "Kserve," 2024. https://github.com/kserve/kserve.

[30] 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.

[31] M. Yu, A. Wang, D. Chen, H. Yu, X. Luo, Z. Li, W. Wang, R. Chen, D. Nie, and H. Yang, "Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping," *arXiv preprint arXiv:2306.03622*, 2024.

[32] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Neural Information Processing Systems*, 2019.

[33] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. X. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *International Conference on Machine Learning (ICML)*, 2021.

[34] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019.

[35] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-sgd: a decentralized pipelined sgd framework for distributed deep net training," in *Neural Information Processing Systems*, 2018.

[36] Y. Chen, C. Xie, M. Ma, J. Gu, Y. Peng, H. Lin, C. Wu, and Y. Zhu, "Sapipe: staleness-aware pipeline for data-parallel dnn training," in *Neural Information Processing Systems*, 2024.

[37] Z. Jiang, H. Lin, Y. Zhong, Q. Huang, Y. Chen, Z. Zhang, Y. Peng, X. Li, C. Xie, S. Nong, Y. Jia, S. He, H. Chen, Z. Bai, Q. Hou, S. Yan, D. Zhou, Y. Sheng, Z. Jiang, H. Xu, H. Wei, Z. Zhang, P. Nie, L. Zou, S. Zhao, L. Xiang, Z. Liu, Z. Li, X. Jia, J. Ye, X. Jin, and X. Liu, "MegaScale: Scaling large language model training to more than 10,000 GPUs," in *USENIX NSDI*, 2024.

[38] 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.

[39] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *USENIX OSDI*, 2024.

[40] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *USENIX OSDI*, 2024.

[41] 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.

[42] "PyTorch." https://pytorch.org/.

[43] "TensorFlow." https://www.tensorflow.org/.

[44] 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.

[45] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, *et al.*, "Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2022.

[46] "TensorRT-LLM," 2023. https://github.com/NVIDIA/TensorRT-LLM.

[47] "Triton," 2020. https://developer.nvidia.com/triton-inference-server.

[48] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[49] 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.

[50] 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.

[51] J. Huang, M. Zhang, T. Ma, Z. Liu, S. Lin, K. Chen, J. Jiang, X. Liao, Y. Shan, N. Zhang, M. Lu, T. Ma, H. Gong, and Y. Wu, "Trenv: Transparently share serverless execution environments across different functions and nodes," in *ACM SOSP*, 2024.

[52] "Safetensors: Ml safer for all," 2024. https://github.com/huggingface/safetensors/.

[53] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *ACM SOSP*, 2021.

[54] "Torchscript," 2024. https://pytorch.org/docs/stable/jit.html.

[55] "NVIDIA NVLink," 2024. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[56] "Kubernetes." https://kubernetes.io/.

[57] "RyokoAI/ShareGPT52K · Datasets at Hugging Face," 2023. https://huggingface.co/datasets/RyokoAI/ShareGPT52K.

[58] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[59] Y. Bai, X. Lv, J. Zhang, H. Lyu, J. Tang, Z. Huang, Z. Du, X. Liu, A. Zeng, L. Hou, Y. Dong, J. Tang, and J. Li, "Longbench: A bilingual, multitask benchmark for long context understanding," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024.

[60] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *USENIX ATC*, 2021.

[61] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *ACM ASPLOS*, 2022.

[62] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *ACM Symposium on Cloud Computing*, 2023.

[63] Y. Yang, D. Du, H. Song, and Y. Xia, "On-demand and parallel checkpoint/restore for gpu applications," in *ACM Symposium on Cloud Computing*, 2024.

[64] R. Ma, X. Yang, J. Wang, Q. Qi, H. Sun, J. Wang, Z. Zhuang, and J. Liao, "Hpipe: Large language model pipeline parallelism for long context on heterogeneous cost-effective devices," 2024.

[65] B. Butler, S. Yu, A. Mazaheri, and A. Jannesari, "Pipeinfer: Accelerating llm inference using asynchronous pipelined speculation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2024.

[66] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, "Pipeedge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices," in *25th Euromicro Conference on Digital System Design (DSD)*, 2022.

[67] 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.

[68] 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 *USENIX OSDI*, pp. 521–538, 2022.

[69] 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.

[70] A. Agrawal, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, and R. Ramjee, "Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills," *arXiv preprint arXiv:2308.16369*, 2023.