HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds

Chiheng Lou[1] Sheng Qi[1] Chao Jin[1] Dapeng Nie[2] Haoran Yang[2] Yu Ding[2]
Xuanzhe Liu[1] Xin Jin[1]

[1]*School of Computer Science, Peking University* [2]*Alibaba Group*

Abstract

With the proliferation of large language model (LLM) variants, developers are turning to serverless computing for cost-efficient LLM deployment. However, public cloud providers often struggle to provide performance guarantees for serverless LLM serving due to significant cold start latency caused by substantial model sizes and complex runtime dependencies. To address this problem, we present HydraServe, a serverless LLM serving system designed to minimize cold start latency in public clouds. HydraServe proactively distributes models across servers to quickly fetch them, and overlaps cold-start stages within workers to reduce startup latency. Additionally, HydraServe strategically places workers across GPUs to avoid network contention among cold-start instances. To minimize resource consumption during cold starts, HydraServe further introduces pipeline consolidation that can merge groups of workers into individual serving endpoints. Our comprehensive evaluations under diverse settings demonstrate that HydraServe reduces the cold start latency by 1.7×– 4.7× and improves service level objective attainment by 1.43×–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]. While the industry is releasing numerous LLMs with varying structures, sizes, or training corpora for diverse scenarios [1–3, 9–20], businesses of different scales are also seeking to develop custom LLMs tailored to their unique needs [21–24]. As a result, a global ecosystem comprising over one million models has emerged, as documented by Hugging Face [25].

To efficiently deploy these models, serverless LLM serving has emerged as a compelling choice [25–33]. In serverless inference, the cloud platform automatically provisions GPUs for models based on the actual request load, and model owners are only charged for the resource consumption during request processing. This approach offers significant cost savings for long-tail models with infrequent usage.

However, for cloud providers, delivering performance guarantees for serverless LLM users is challenging, primarily due to the long latency of cold starts, i.e., provisioning new workers to meet the current load. Production traces reveal that bursty traffic patterns are prevalent in both LLM serving workloads [34] and serverless workloads [35], making cold starts inevitable in serverless LLM serving [30–33]. Cold starts can directly impact service level objectives (SLOs) for LLM inference. For instance, in our production environment, a cold-start instance produces the first token after more than 40 seconds, whereas subsequent generation takes only ∼30ms per token. This contrast stresses the need for faster cold starts.

Addressing this problem is non-trivial, as LLM cold starts introduce two unique challenges compared to traditional serverless workloads. First, LLM applications require massive external state, with model weights fetched on demand from remote registries during cold starts. Since serverless instances typically operate with constrained network bandwidth, model fetching becomes the primary bottleneck in cold start latency. Second, the LLM runtime is highly complex, involving multiple specialized libraries including CUDA runtime [36], AI frameworks [37, 38], and inference engines [39–41]. The complex dependencies of these libraries lead to substantial container creation and library loading overhead [42]. In public clouds, applications run in isolated containers with varying library versions, making it impractical to prepare runtimes in advance for every user.

Prior works focus on reducing the model fetching latency and propose two approaches: (1) caching LLMs in memory or SSDs [30, 33, 43] and (2) retrieving model weights from active workers holding the same model [31–33]. While effective for frequently accessed models, these optimizations fall short for long-tail customers, who benefit the most from the serverless paradigm [44]. Additionally, retrieving data from peer workers requires high-bandwidth networking, which is not cost-efficient for serverless inference clusters.

To this end, we present HydraServe, a serverless LLM serving system designed to minimize cold start latency in public clouds. HydraServe incorporates two key insights to mitigate the overhead of model fetching and runtime preparation. First, while a single server has limited bandwidth for model fetching, distributing workers across multiple servers enables efficient bandwidth aggregation. Second, since model fetching and runtime preparation exhibit weak execution dependencies, these stages can be parallelized rather than executed sequentially, thereby overlapping their latencies.

Specifically, to distribute models across servers, we take advantage of the layered structure of LLMs. HydraServe partitions LLM layers across servers and exchanges intermediate results during inference. This design, known as *pipeline parallelism*, has been well studied in distributed model training and inference [45–54]. Pipeline parallelism is also used to quickly start inference when scaling up multiple workers [32]. HydraServe further adopts pipeline parallelism to reduce the blocking effect of model fetching—we proactively create a group of workers in parallel even if the model only needs one.

Although pipeline parallelism can effectively reduce the cold start latency for the initial inference, it may impact performance in subsequent rounds due to GPU sharing. To address this, we propose *pipeline consolidation*. Specifically, we allow workers to fetch and load the layers previously assigned to other workers in the background, eventually transitioning back to local inference with a fully-loaded model. Depending on the load, each worker can independently decide whether to release its resources or load the full model after cold starts. Therefore, with a single cold start, HydraServe can choose to only create a single worker in the end or scale up as many workers as the pipeline size, allowing for high elasticity.

HydraServe takes a three-level *hierarchical* approach to realize the above optimizations. At the *cluster level*, we design an algorithm to allocate resources for cold-start models based on the model size, cluster resource usage, and user SLOs. A network-contention-aware worker placement policy is adopted to avoid potential network contention. At the *worker level*, we overlap remote-to-host model fetching, host-to-GPU model loading, and the initialization of container and GPU runtime to further reduce the cold start latency. Finally, at the *inference level*, we consolidate workers that are previously created in pipeline groups and migrate their runtime states to ensure peak performance with all weights loaded.

In summary, we make the following contributions:

- We articulate the main bottleneck of LLM cold starts in public clouds and identify opportunities to mitigate the overhead.
- We propose a hierarchical approach that combines cluster-level, worker-level, and inference-level optimizations to reduce cold start latency for serverless LLM serving in public clouds.
- We implement HydraServe in both testbed and production environment. Experiments on real-world datasets show that HydraServe reduces the cold start latency by 1.7×– 4.7× and improves SLO attainment by 1.43×–1.74× compared to baselines. The brownfield evaluation further confirms these gains, yielding an average latency reduction of 2.6×.

2 Background and Motivation

This section provides an overview of large language models (LLMs) and serverless LLM serving. We highlight the critical issue of limited network bandwidth for LLM serving in public clouds, which can substantially increase cold start latency. We then identify opportunities for mitigating this performance bottleneck in a cost-effective manner.

2.1 LLM Inference

An LLM takes as input a sequence of tokens, called *prompt*, and generates an output sequence in an *autoregressive* manner, producing one token at a time based on both the prompt and previously generated 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.

In LLM serving, users often specify service level objectives (SLOs) that represent performance expectations [51, 55–57]. These SLOs typically focus on two metrics: time to first token (TTFT) and time per output token (TPOT). TTFT measures the latency from request submission to the generation of the *first* token, while TPOT represents the average time taken to generate each *subsequent* token. The relative importance of these metrics varies across different applications. For example, real-time chatbots emphasize low TTFT, whereas article writing prefers lower TPOT [55].

Pipeline parallelism is a model parallelism strategy that is commonly used in LLM training to scale up involved GPUs [45–50, 58]. This approach distributes a model's layers across multiple workers, with intermediate results transmitted sequentially between workers during computation.

2.2 Serverless LLM Serving in Public Clouds

Recently, serverless inference has gained widespread adoption among cloud providers, including Aliyun [27], AWS [26], and Azure [59]. In serverless LLM serving, users upload (1) model weights and (2) an image containing the serving framework and runtime dependencies to a cloud storage. This serving framework is responsible for fetching model weights

Instance	Mem.(GB)	Band.(Gbps)	#GPU	Cost($/h)	Cost/GPU($/h)
g6e.xlarge	32	up to 20	1	1.861	1.861
g6e.2xlarge	64	up to 20	1	2.24208	2.24208
g6e.4xlarge	128	20	1	3.00424	3.00424
g6e.8xlarge	256	25	1	4.52856	4.52856
g6e.16xlarge	512	35	1	7.57719	7.57719
g6e.12xlarge	384	100	4	10.49264	2.62316
g6e.24xlarge	768	200	4	15.06559	3.76640
g6e.48xlarge	1536	400	8	30.13118	3.76640

Table 1: Configurations and costs of L40S instances on AWS EC2. The number of vCPU is proportional to memory capacity.

from remote storage, loading them into GPU memory, and running inference upon user requests. The serverless platform automatically scales model workers to accommodate fluctuating loads, including the ability to scale down to zero workers when idle. One of the most attractive offerings of serverless LLM serving is its pay-per-use billing, where users are charged only for the running periods of each worker. In production environments, long-tail models typically exhibit sporadic and unpredictable workload patterns [60], making them ideal candidates for serverless serving [30].

Network bandwidth constraints. Contrary to conventional wisdom, serverless LLM inference may operate with substantially lower network capacity than commonly anticipated. This constraint arises partly from multiple instances sharing a single server, but the primary driver is the cost-efficiency requirement of serverless computing. LLM inference performance mainly depends on the GPU capabilities. Since serverless customers prioritize cost savings, cloud providers aim to minimize **cost per GPU** when configuring their infrastructure. This cost-optimization strategy favors servers with reduced CPU, memory, and network resources.

Table 1 illustrates this economic principle by comparing configurations and costs of AWS EC2 instance types equipped with NVIDIA L40S GPUs [61]. As Table 1 reveals, resources other than GPUs constitute a significant portion of server costs. Using single-GPU instances as an example, compared to the instance type with the lowest cost per GPU (g6e.xlarge), adding extra resources can increase costs by 20% to 300%. Consequently, servers with constrained non-GPU resources deliver substantial cost savings, making them the preferred choice for serverless providers. However, these economical instances impose network bandwidth limitations that significantly increase cold start latency. While a network-only upgrade is less expensive, the additional network bandwidth would be severely underutilized, as it is only used during cold starts for model fetching. This paper aims to address this trade-off by minimizing cold start latency while preserving the cost benefits of resource-constrained servers.

Breakdown of LLM cold starts. 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, necessitating the creation of a new worker. Figure 1 provides a detailed breakdown of cold start latency in our

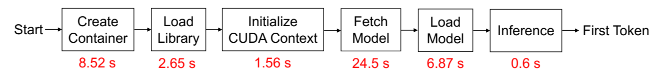


Figure 1: Cold start latency breakdown.

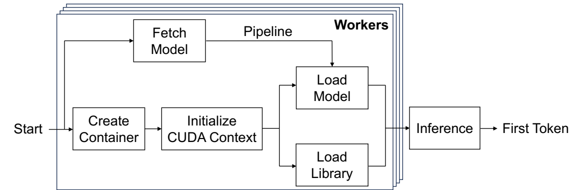


Figure 2: Optimized cold-start workflow.

public serverless inference platform [27], using vLLM [40] to run a Llama2-7B model on NVIDIA A10 GPUs. In general, a cold start involves the following stages:

- **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 libraries like PyTorch [37] and TensorFlow [62], along with LLM serving frameworks such as vLLM [40], SGLang [41], and TensorRT-LLM [39].
- **CUDA Context Initialization.** The runtime initializes 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 and model states including CUDA graph and KV cache are initialized.
- **Inference.** Run the model and generate the first token.

As shown in Figure 1, a cold start requires more than 40 seconds to produce the first token. Model fetching dominates this latency due to limited network bandwidth and contention among colocated containers. Additionally, container creation incurs substantial overhead due to the large image sizes of LLM workloads (8.31 GB in our setup). For model loading, prior work has largely eliminated CUDA graph and KV cache initialization overhead through state materialization [63]. The remaining bottleneck lies in transferring model weights to the GPU, which takes approximately 2 seconds in our setup.

2.3 Opportunities

This paper achieves maximum overlapping to reduce cold start latencies. At the cluster level, we leverage pipeline parallelism for coarse-grained overlapping of model fetching across workers. Upon a cold start, we create a pipeline parallelism group across GPU servers, with each worker only hosting a part of the model. This approach can significantly reduce the single-worker startup latency. At the worker level, we perform fine-grained overlapping by carefully reorganizing the intra-worker workflow in Figure 1. We offload model fetching to a system-level service, allowing fetching to begin before container creation. Furthermore, we observe that model loading (GPU-bound) and library loading (CPU-bound) use

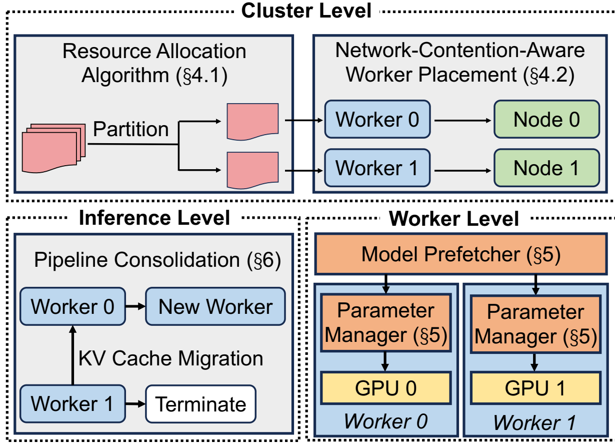


Figure 3: HydraServe system overview.

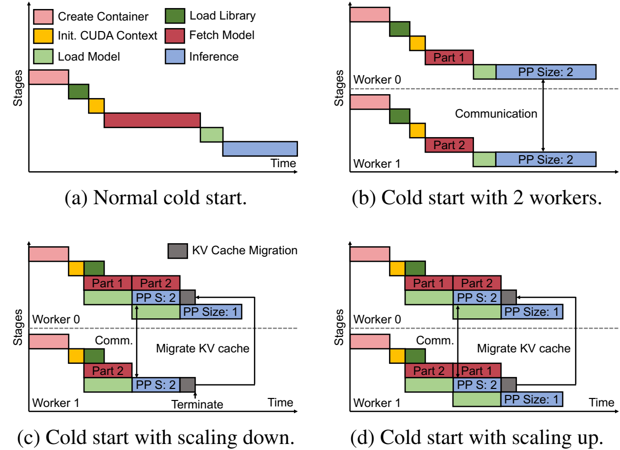


Figure 4: Comparison of cold-start workflows for different methods.

different resources, allowing parallel execution. We reorder operations to initialize CUDA context first, and then simultaneously start to load model and library. Finally, we pipeline the fetching and loading at tensor granularity to hide the loading overhead. Figure 2 shows this optimized workflow.

3 HydraServe Overview

HydraServe is a serverless LLM serving system designed to minimize the cold start latency. As shown in Figure 3, HydraServe adopts a three-level hierarchical architecture. At the cluster level, the central controller employs a resource allocation algorithm (§4.1) and a network-contention-aware worker placement strategy (§4.2) for parallel model fetching. The algorithm allocates resources for each cold-start model based on user SLOs, while the placement strategy identifies potential network contentions on GPU servers and ensures that such contention does not cause SLO violations. HydraServe also strives to distribute pipeline workers across GPUs to mitigate performance degradation caused by GPU sharing.

At the worker level, HydraServe applies fine-grained overlapping across cold-start stages to accelerate worker startup (§5). First, a node-level model prefetcher is used to proactively fetch model weights for cold-start workers, overlapping model fetching with container creation and runtime initialization times. Second, HydraServe prioritizes the CUDA context initialization and introduces a parameter manager that loads parameters to the 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, at the inference level, HydraServe consolidates pipeline workers into standalone workers that host all parameters (§6). Specifically, HydraServe allows a cold-start worker to continue loading the remaining model parts while serving requests, eventually evolving into an individual worker that hosts the entire model. Each worker can independently decide whether to evolve or terminate after completing the cold start process. Ongoing requests in the pipeline parallelism group are then migrated to the standalone workers.

HydraServe ensures that cold starts incur no higher first-token latencies or request completion times than standard cold-start processes. This guarantee in request completion time is achieved through HydraServe's request migration mechanism, which ensures peak inference speed after all model weights have been prepared.

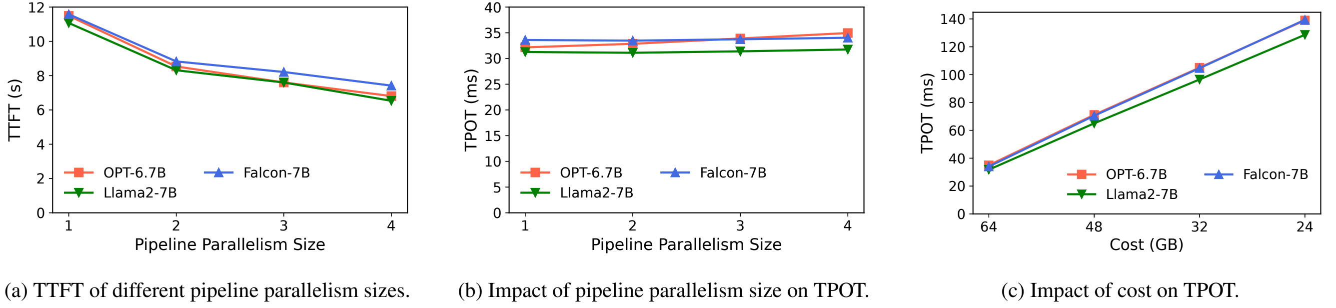
4 Cluster-Level Controller

In this section, we present the design of the cluster-level central controller in HydraServe. Figure 4(a) illustrates the normal cold-start workflow, which goes through six stages sequentially. HydraServe leverages pipeline parallelism to optimize model fetching, the most time-consuming stage. As shown in Figure 4(b), upon a cold start, we launch multiple workers on different servers, each fetching a part of the model. During inference, workers exchange intermediate results but the sizes of these messages are relatively small. By reducing the portion of the model each worker needs to fetch, pipeline parallelism significantly reduces cold start latency. HydraServe finds the optimal resource allocation scheme for each cold-start model based on the model size and user SLOs (§4.1), and employs a network-contention-aware worker placement strategy to prevent SLO violations (§4.2).

4.1 Resource Allocation

To design an efficient resource allocation algorithm for pipeline workers, we begin by quantifying the tradeoffs involved in pipeline parallelism. Based on this analysis, we propose a TTFT and worst-case TPOT prediction method.

Tradeoff analysis. The benefits of pipeline parallelism come with performance and resource overhead. To better understand these tradeoffs, we conduct experiments using four GPU



(a) TTFT of different pipeline parallelism sizes. (b) Impact of pipeline parallelism size on TPOT. (c) Impact of cost on TPOT.

Figure 5: Tradeoff analysis of pipeline parallelism.

servers, each equipped with a NVIDIA A10 GPU and 188 GB memory. The network bandwidth of each server is 16 Gbps.

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 in TTFT diminishes as other cold-start stages also consume non-negligible time. This limitation is further addressed by worker-level optimizations in §5.

Pipeline parallelism has a modest impact on TPOT, as shown in Figure 5(b). This is because the messages transmitted between workers are relatively small. For example, Llama2-7B incurs only 8 KB of inter-layer results per token.

However, pipeline parallelism may lead to worker colocations across different models. For a model initially requiring M GPU memory, distributing it across S GPUs reduces per-GPU memory consumption to M/S. Under heavy load, the cluster must co-place workers whose combined memory requirements fit within a single GPU's capacity to maximize resource utilization. This colocation of workers from different models can degrade performance.

Figure 5(c) illustrates this issue by changing the GPU memory allocated to each model, with pipeline parallelism size fixed at 4. Allocated GPU memory represents the cost of a model, and lower memory usage per model leads to worker colocation. For example, allocating 32GB GPU memory to each model makes three models share four GPUs. As the cost decreases, each GPU hosts multiple concurrently running workers. Consequently, the GPU's computational resources are allocated proportionally to each worker's reserved memory, resulting in longer TPOT.

In conclusion, while increasing pipeline parallelism size reduces the cold start latency, it may impact inference performance due to GPU sharing. Allocating additional GPU memory to workers can mitigate this performance loss, but at the expense of higher resource consumption. HydraServe's resource allocation algorithm systematically balances these tradeoffs to optimize system performance.

Algorithm design. The resource allocation algorithm finds an appropriate resource allocation strategy for each cold-start model according to user SLOs. Apart from pipeline parallelism size, the algorithm should decide how to place pipeline workers and allocate how much GPU memory to each worker.

To narrow down the search space, we limit the GPU memory allocation choices for each worker to two cases: (a) the same as the non-parallelized setup, and (b) the minimal memory required to perform inference (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 deployment choices with different pipeline parallelism sizes or GPU memory usage and predict the TTFT and worst-case 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 yield little improvement.

The TTFT and TPOT prediction takes historical information as the input. Formally, it includes 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 the TCP network between GPU servers. The prefill and decoding time costs are model-specific metrics obtained from the model's previous executions. Assume the pipeline parallelism size for the current deployment choice is s and there are w full-memory workers ($0 \le w \le s$), while the network and PCIe bandwidth of servers on which workers reside 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.

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

where M is the model size. Eq. 1 is made up of four parts: runtime initialization, model fetching, model loading, and prefilling. The model fetching and loading time is determined by the partitioned model size and speed of the slowest worker. During prefilling, a full-memory worker costs $t_p/s + t_n$ units of time, while a low-memory worker costs $t_p + t_n$ units.

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 work-

Algorithm 1 Resource Allocation Algorithm

Input: time cost of container creation and runtime initialization t_c, data transmission t_n, prefill t_p, and decoding t_d; model size M; GPU server network bandwidth b_i and PCIe bandwidth p_i; user specified requirements SLO_{TTFT} and SLO_{TPOT}.
Output: pipeline parallelism size s, #full-memory workers w, and selected GPU servers g.

$S \leftarrow \emptyset$
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.
    $j_1, j_2, \cdots, j_l \leftarrow$ Servers that fit a model of size M/s.
    $j'_1, \cdots, j'_{l'} \leftarrow \text{MergeSort}((j_1, \cdots, j_l), (i_{w+1}, \cdots, i_k))$
    $g \leftarrow (i_1, i_2 \cdots, i_w, j'_1, \cdots, j'_{s-w})$
    $\text{max_ratio} \leftarrow \max_{x \in g} \left(\frac{1}{b_x} + \frac{1}{p_x}\right)$
    $\text{TTFT} \leftarrow t_c + \frac{M}{s} \times \text{max_ratio} + t_p \times (s - w + \frac{w}{s}) + t_n \times s$
    $\text{TPOT} \leftarrow t_d \times (s - w + \frac{w}{s}) + t_n \times s$
    if $\text{TTFT} \leq \text{SLO}_{\text{TTFT}}$ **and** $\text{TPOT} \leq \text{SLO}_{\text{TPOT}}$ **then**
      $S \leftarrow S \cup \{(s, w, g)\}$
if S is $\emptyset$ **then**
  return $(1, 1, (i_1))$ ▷ Use single worker if no solution
else
  $c \leftarrow$ Scheme that incurs minimal GPU sharing from S
  return c

ers 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 the same strategy and allocate them to all low-memory workers.

Similar to the calculation of prefilling latency, 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, HydraServe prioritizes free GPUs during worker placement. When the cluster is not under heavy load, this approach improves the inference performance of models. Algorithm 1 outlines the resource allocation 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 among cold-start workers on the same GPU server, which leads to unpredictable cold start performance. To solve this problem, we propose a network-contention-aware worker placement policy that places cold-start workers based on user SLOs.

Policy design. Initially, colocated 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 whether adding a cold-start worker to this server would lead to SLO violations for existing workers. If the check passes, we estimate and record the network bandwidth that the new worker can obtain. Note that PCIe bandwidth 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 per work decreases, and we check whether all cold-start workers on the server are able to finish timely under the new bandwidth. Formally, we check whether the following condition is true for all workers.

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

where N represents the number of existing workers on the server and T represents the current time. This server accepts the new worker if all workers passed the check.

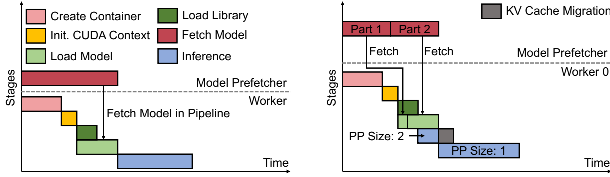
To estimate the pending model size, we record the time of last network bandwidth change T'. The start and completion of a cold start change the network bandwidth, informing the controller to adjust the pending model size. Here, we assume the current time is T and the number of cold-start workers before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

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

$S'_i < 0$ means that the worker has fetched the model ideally, thus we delete it from the cold-start worker list.

5 Worker-Level Overlapping

As described in §4.1, the TTFT of pipeline parallelism has diminishing marginal returns due to container initialization stages, including container creation, library loading and CUDA context initialization. In this section, we present the worker-level overlapping in HydraServe that carefully reorganizes the startup workflow to reduce worker initialization



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

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

time. Basically, HydraServe 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

HydraServe 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 the SafeTensors format [64]. This 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

HydraServe 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 the 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) shows the optimized cold-start workflow after adopting model prefetching and parameter manager. Model fetching and runtime preparation 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 HydraServe 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 transforms itself into a standalone worker.

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 resource allocation (Eq. 1) into the following formula.

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

6 Inference-Level Consolidation

As HydraServe creates a group of workers in parallel when the model only needs one, we propose 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 the whole model in local GPU, thereby achieving optimal performance. The second part of model is loaded in low-priority CUDA streams, so that the performance of inference task will not be affected.

6.1 Worker Scaling

HydraServe provides two scaling choices. First, after workers have loaded corresponding parts of models, we can perform *scaling down* by allowing only one of them to fetch the unloaded model parts in background. Once all parameters are loaded, we migrate all existing requests to that worker along with 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 produce the first token earlier by parallelized model fetching and finally obtain a worker with whole model, similar to standard cold starts.

The second scaling choice is *scaling up*, converting all cold-start workers into individual serving endpoints, as illustrated in Figure 4(d). This approach tackles load spikes, where the autoscaler aims to deploy multiple workers for the model simultaneously. By enabling the rapid creation of workers within pipeline parallelism groups, this design allows the system to achieve maximum throughput more quickly.

By default, HydraServe adopts the scaling down mechanism to reduce model fetching latency with minimal overhead. To promptly react to bursty loads and transition to the scaling up mechanism, we manage worker lifecycles with a sliding window strategy. For each model, the number of requests received in the previous window is recorded and used to predict the maximum number of requests likely to arrive in the next window. The required number of workers is then determined based on the current waiting queue length 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. To handle sudden request surges, multiple pipeline parallelism groups can be created as needed.

6.2 Key-Value Cache Migration

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

HydraServe performs KV cache migration inside a pipeline parallelism group, where layers are 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 with a *gather* operation. Blocks are gathered to the worker with whole model and placed at different layers, according to which worker it comes from.

The KV cache migration process in HydraServe 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 HydraServe in both testbed environments and our public serverless inference platform. Our implementation is based on vLLM [40], with a modification of around 3200 lines of code in C++ and Python. For testbed deployment, we develop a custom serverless LLM serving framework with around 3000 lines of code in Python. This framework makes scheduling decisions and automatically scales number of vLLM workers for each model to match current loads. The code of HydraServe is open-source and is publicly available at https://github.com/LLMServe/hydraserve.

Instance startup optimizations. We perform several optimizations to the initialization process of vLLM. First, we postpone the allocation of key-value cache swapping space on CPU to reduce cold start latency. Second, vLLM runs a profiling forward to obtain the amount of free memory during inference. Unlike prior work that profiles this information offline [63], we directly calculate the available free memory based on sizes of intermediate results to skip the online profiling phase. Third, as vLLM first initializes the model on CPU, we allow the instance to directly use GPU tensors provided by the parameter manager through overriding tensor metadata.

Support for large models. HydraServe is applicable to both models that reside in a single GPU as well as those deployed in multiple servers with tensor and pipeline parallelism. Similarly, to improve the cold-start performance of models that are distributed across GPUs, we can increase their pipeline parallelism size and implement worker-level overlapping.

8 Evaluation

In this section, we present experimental results to validate the efficiency and effectiveness of HydraServe. Our evaluation shows that HydraServe achieves a substantial reduction in TTFT, outperforming baselines by $1.7\times$– $4.7\times$ (§8.2). In end-to-end experiments, HydraServe improves TTFT SLO attainment by $1.43\times$–$1.74\times$ across various loads and constraints while maintaining minimal TPOT and cost penalties (§8.3). Our analysis of pipeline consolidation reveals that scaling down reduces end-to-end generation time by up to $2.67\times$, while scaling up shortens the average TTFT under bursty requests by $1.87\times$ (§8.4). Lastly, we wrap up with brownfield

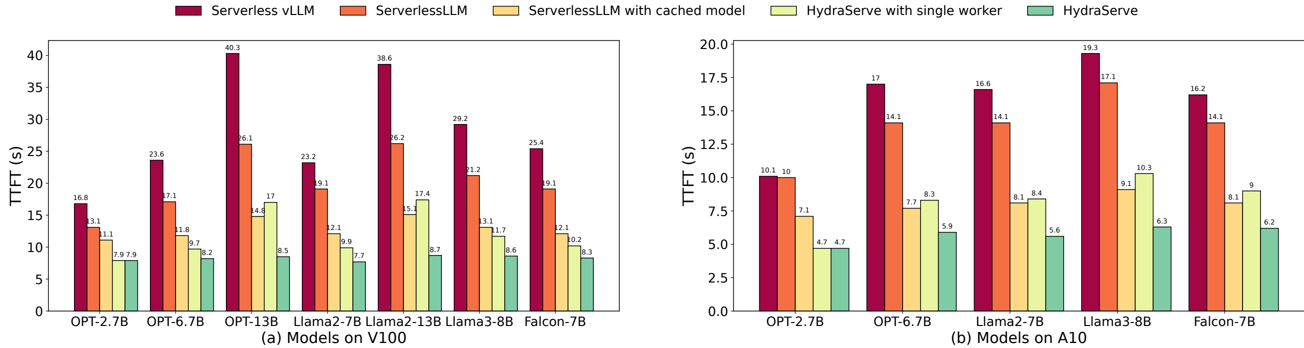


Figure 7: Cold start latency of systems for different models.

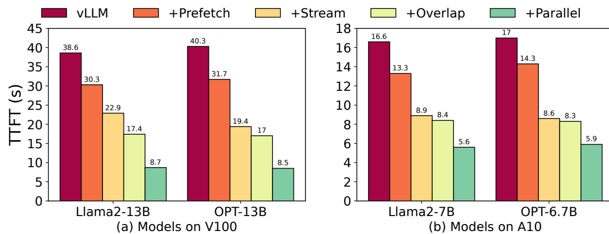


Figure 8: Performance breakdown of techniques in HydraServe.

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

Table 2: Measured TTFT and TPOT of warm requests.

Application	TTFT	TPOT	Dataset
Chatbot Llama2-7B	7.5s	200ms	ShareGPT [67]
Chatbot Llama2-13B	12s	200ms	ShareGPT [67]
Code Completion Llama2-7B	7.5s	84ms	HumanEval [68]
Code Completion Llama2-13B	12s	116ms	HumanEval [68]
Summarization Llama2-7B	15s	84ms	LongBench [69]
Summarization Llama2-13B	24s	116ms	LongBench [69]

Table 3: Summary of applications in end-to-end experiments.

results and show that HydraServe 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 188 GB memory, while each V100 server has four NVIDIA V100 GPUs and 368 GB memory. The network bandwidth per server is 16 Gbps. (ii) A GPU cluster consisting of 2 servers, each having four NVIDIA A10 GPUs, and another 4 servers, each having four NVIDIA V100 GPUs. Each A10 server has 752 GB memory and a network bandwidth of 64 Gbps, while each V100 server has 368 GB memory and 16 Gbps bandwidth. Both testbeds are connected to a remote model storage that has sufficient network capacity. Due to the lack of NVLink [65], all evaluated models can reside in a single GPU to avoid performance degradation.

Baselines. We compare HydraServe against two baselines.

• Serverless vLLM. vLLM [40] is a LLM serving engine that can serve single model. To serve multiple models, we equip it with the serverless LLM serving framework of HydraServe. During cold starts, the scheduler iterates through all GPU servers and selects the one with sufficient GPU resources to create a new vLLM serving endpoint.

• ServerlessLLM. ServerlessLLM [30] is the 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 testbeds, we allocate all available server memory for model caching. We deploy ServerlessLLM on Kubernetes [66] and pre-create its containers to eliminate container creation overhead during serving. The system also uses vLLM [40] as the backend.

We exclude systems that accelerate scaling up via worker cooperation [31, 32] from baselines because (1) they are closed-source and (2) peer fetching offers no benefit over remote storage in our environment.

8.2 Cold Start Latency

We evaluate the cold start latency for systems on testbed (i), while HydraServe is configured at a parallelism size of 4. For ServerlessLLM, we measure its performance with and without model caching. Figure 7 illustrates the TTFT of systems for different models, showing that HydraServe achieves the shortest cold start latency across all models. Specifically, HydraServe reduces cold start latency by 2.1×– 4.7× compared to serverless vLLM and 1.7×–3.1× compared to ServerlessLLM. This improvement is attributed to parallelized model fetching and our worker-level optimizations.

Even with a single worker, HydraServe also performs better than ServerlessLLM. For smaller models like OPT-6.7B, HydraServe with single worker even achieves shorter cold start latency than ServerlessLLM with cached models. This advantage comes from HydraServe's worker-level overlapping strategies and optimizations to vLLM's cold-start process.

Performance breakdown. To better comprehend the performance of HydraServe, we conduct a detailed breakdown of

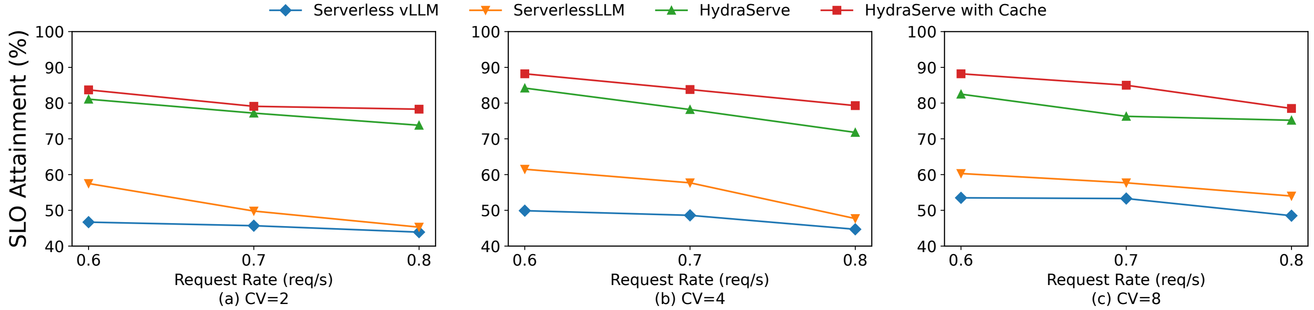


Figure 9: TTFT SLO attainment of systems under different CVs.

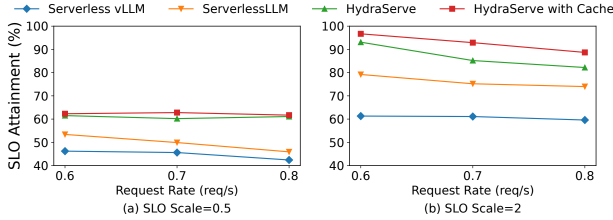


Figure 10: TTFT SLO attainment of systems under different SLOs.

techniques employed in HydraServe. Figure 8 shows 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 and library loading (+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 HydraServe 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 [55], we use three typical LLM applications in experiments: chatbot, code completion, and summarization. Requests for these applications are from ShareGPT [67], HumanEval [68] and Longbench [69], respectively. Since there is no available SLO settings for these applications, we derive SLOs based on the performance of warm requests. Specifically, we first measure the TTFT and TPOT for warm requests, with each request containing 1024 input tokens and a batch size of 8. The results are shown in Table 2. We then set the global TTFT SLO to five times the TTFT of warm requests, while applying a stricter TPOT SLO at twice the TPOT of warm requests. This TTFT SLO is already quite stringent for serverless LLM serving, considering industrial LLM serving platforms operate with TTFT SLOs as high as 30s [70].

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 [30, 51]. Table 3 shows the summary of applications.

Following prior work [30, 51], we leverage Microsoft Azure Function Trace [35] to generate workloads. Models are mapped to functions in the trace using a round-robin approach, and requests are sampled from the trace with Gamma distribution. We control the sampling process by changing coefficient of variance (CV) and requests per second (RPS).

Effectiveness under different CVs. Figure 9 demonstrates the TTFT SLO attainment of systems under different CVs. To assess the effectiveness of cache, we also evaluate HydraServe with caching enabled. The results indicate that as RPS increases, TTFT SLO attainment decreases due to a lack of resources during bursty requests. Nevertheless, HydraServe consistently satisfies most of TTFT SLO requirements under different loads, achieving 1.43×–1.74× higher TTFT SLO attainment compared to baselines in all scenarios. This is because HydraServe selects appropriate pipeline parallelism size based on user SLOs and distributes cold-start workers to mitigate network contention. Although some workers are allocated with additional resources to reduce TPOT, HydraServe promptly reclaims these resources after pipeline consolidation to maintain performance under heavy loads. In contrast, ServerlessLLM exhibits high SLO violations due to limited effectiveness of caching for long-tail models. Enabling caching in HydraServe further improves the TTFT SLO attainment by up to 1.11× due to benefits for hot models.

For TPOT SLO attainment, both HydraServe and baselines achieve over 95% attainment in most scenarios, and more than 90% attainment under all CVs and RPS configurations. This level of performance is sufficient for most use cases. Detailed results are provided in the appendix due to space limitations.

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. Figure 10(a) shows that, under tight SLOs, all systems experience significant SLO violations since 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%.

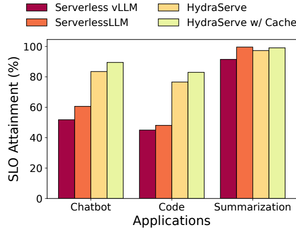


Figure 11: TTFT SLO attainment for different applications.

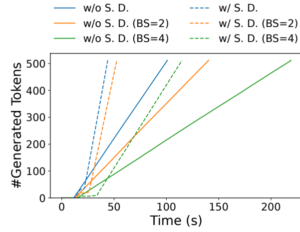
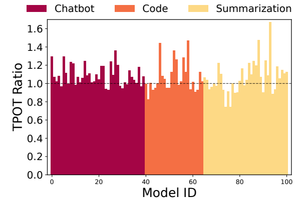
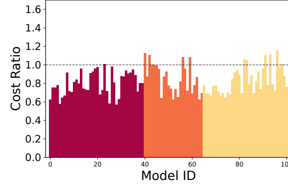


Figure 12: Total tokens generated over time for different inputs.



(a) TPOT ratios for different models.



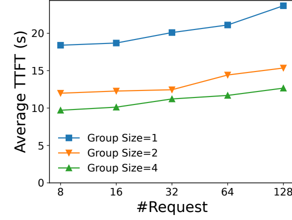
(b) Cost ratios for different models.

Figure 13: Relative TPOT and cost ratios (HydraServe vs. serverless vLLM) for different models.

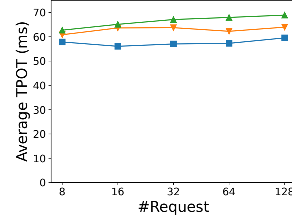
However, HydraServe still outperforms baselines, as its faster worker initialization reduces the waiting time for subsequent requests, even if the first request violates SLO. Figure 10(b) demonstrates the performance under looser SLO conditions. HydraServe achieves 1.38×–1.52× improvement in TTFT SLO attainment compared to baselines, with caching further enhancing this improvement to 1.49×–1.58×.

Application analysis. Figure 11 illustrates the TTFT SLO attainment of chatbot, code completion, and summarization applications under CV=8 and RPS=0.6. First, the results show that HydraServe significantly enhances the TTFT SLO attainment for chatbot and code applications, with up to 1.61× and 1.70× improvement, respectively. The code application has lower SLO attainment compared to others because code completion tasks (i.e., requests in HumanEval dataset [68]) have shorter average output length than chat tasks (i.e., requests in ShareGPT dataset [67]) [55]. Therefore, workers for code completion models keep alive for shorter time, leading to more cold starts. Since the summarization application has loose SLOs, it has few SLO violations for all systems.

TPOT and resource usage penalties. As discussed in §4.1, pipeline parallelism increases the worst-case TPOTs, whereas allocating additional resources helps to reduce them. We evaluate HydraServe's TPOT and resource usage of different models to examine these trade-offs. Figure 13 illustrates the relative TPOT and cost ratios of HydraServe compared to serverless vLLM under CV=8 and RPS=0.6. The cost is proportional to the GPU memory-time product. Regarding TPOT, HydraServe exhibits only a 1.06× average increase compared to serverless vLLM, with most increases occurring in chatbot and code models that have stringent TTFT requirements. The TPOT penalty is modest because HydraServe can quickly merge pipeline groups into single workers, limiting performance degradation to only the first few tokens. For some models, HydraServe achieves shorter TPOT due to GPU performance fluctuations.



(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.

Figure 13(b) reveals that, surprisingly, HydraServe consumes lower cost for most models. This is because (1) for models with additional resource consumption, HydraServe can quickly merge pipeline groups, and (2) HydraServe enables faster worker startup, thereby reducing GPU usage during cold starts. On average, HydraServe reduces costs by 1.12× compared to serverless vLLM.

8.4 Pipeline Consolidation

To evaluate the effectiveness of pipeline consolidation, we measure HydraServe's performance with two scaling methods. We deploy Llama2-13B on V100 servers in testbed (i) and set the input and output length of each request to 512 tokens.

Scaling down. We demonstrate the benefits of scaling down by presenting the generation time of each token. We use different batch sizes and set pipeline parallelism size to 4. As shown in Figure 12, with scaling down, the system loads the remaining parts of the model in parallel with inference, and migrates the key-value cache of the ongoing request once loading has completed. This allows subsequent tokens to be generated at a faster speed. As a result, scaling down reduces the end-to-end generation time by 1.90×–2.67×, while maintaining almost same inference speeds during the early cold-start period.

Scaling up. We evaluate scaling up by measuring HydraServe'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 performance overhead, with the average TPOT increasing by 1.08×–1.19×. This increase is attributed to the transmission overhead of intermediate results.

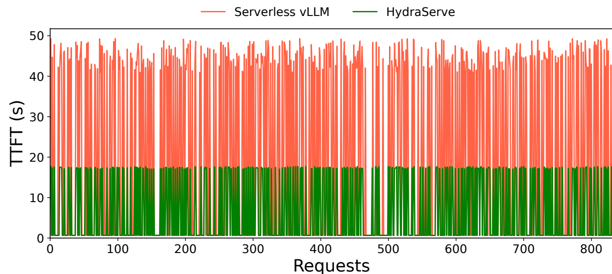


Figure 15: TTFT of requests in brownfield evaluation.

8.5 Brownfield Evaluation

We evaluate the prototype of HydraServe in our production environment. In this environment, functions cannot establish direct TCP connections with one another due to security constraints, so we leverage a shared object in remote storage to enable inter-worker communication. In evaluation, we use vLLM to run Llama2-7B on NVIDIA A10 GPUs with 24 GB GPU memory, with requests generated following Microsoft Azure Function Trace [35]. Figure 15 compares the cold start latency of HydraServe to serverless vLLM. The results demonstrate that HydraServe significantly reduces cold start latency versus the original vLLM system, achieving an average 2.6× reduction in cold-start TTFT. This substantial improvement stems from HydraServe's parallelized model fetching and optimized worker initialization processes.

9 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 [30, 31, 33, 43, 63, 71–73]. For instance, FaaSwap [43] and ServerlessLLM [30] cache models in local memory or SSDs, while INFless [72] and InstaInfer [42] prewarm instances according to historical request patterns. DeepFlow [33] combines model caching and NPU forking to optimize cold start latencies. BlitzScale [31] and λScale [32] achieve fast autoscaling using high-capacity networks, while Medusa [63] speeds up CUDA graph and KV cache construction through state materialization.

HydraServe addresses the cold-start problem in public clouds. Its advantages are twofold. First, it imposes no requirement on the hotness of models and effectively tackles the cold-start problem without relying on caching or worker cooperation. Second, it reduces the end-to-end runtime preparation overhead without any offline profiling.

LLM serving optimizations. Many LLM serving optimizations have been proposed to improve serving performance [40, 51, 55–57, 74–77]. In the area of request scheduling, Orca [75] introduces iteration-level scheduling to run more requests in parallel, while Llumnix [56] employs runtime scheduling to meet user SLOs. For key-value cache management, vLLM [40] leverages the concept of virtual memory in operating systems and InfiniGen [76] optimizes KV block placement to improve inference efficiency. Additionally, DistServe [55] disaggregates prefill and decoding phases to mitigate their performance interference. HydraServe specifically focuses on reducing cold start latency in serverless LLM serving and can easily integrate these optimizations.

Pipeline parallelism. Pipeline parallelism has been widely used to scale GPU resources for training large models [45–50, 58]. Recently, researchers have also explored its usage during inference [31, 32, 51–54]. While AlpaServe [51] observes that model parallelism improves GPU utilization under load spikes and develops a model placement policy, HPipe [52] and PipeEdge [54] apply pipeline parallelism to improve the inference performance for LLMs on edge devices. λScale [32] also adopts pipeline parallelism to allow scaled-up workers to start inference earlier. HydraServe utilizes pipeline parallelism in a proactive manner that creates multiple workers even if the model only needs one. It further introduces pipeline consolidation to merge pipeline workers during inference.

10 Conclusion

This paper presents HydraServe, a serverless LLM serving system designed to reduce cold start latency in public clouds. HydraServe targets the most time-consuming stages of a cold start: model fetching and runtime preparation. To optimize model fetching, HydraServe proactively distributes models across multiple servers, alleviating the burden on any single server. Furthermore, HydraServe incorporates pipeline consolidation that merges workers back into individual endpoints to ensure efficient resource usage and high performance for warm requests. For runtime preparation, HydraServe accelerates the local worker startup by overlapping distinct stages. Evaluation results show that HydraServe reduces cold start latency by 1.7×– 4.7× compared to baselines and improves TTFT SLO attainment by 1.43×–1.74× under various constraints. By integrating these optimizations, HydraServe offers a robust and efficient solution to meet user-defined SLOs in serverless LLM serving, especially for long-tail models.

Acknowledgments. We sincerely thank our shepherd Arvind Krishnamurthy and the anonymous reviewers for their valuable feedback on this paper. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500700, the Scientific Research Innovation Capability Support Project for Young Faculty under Grant ZYGXQNJSKYCXNLZCXM-I1, the Fundamental Research Funds for the Central Universities, Peking University, and the National Natural Science Foundation of China under Grant 62172008 and 62325201. Xin Jin is the corresponding author. Chiheng Lou, Sheng Qi, Chao Jin, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] OpenAI, "Gpt-4 technical report," *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, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," *arXiv:2205.01068*, 2022.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *NeurIPS*, 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," 2025. https://blog.google/technology/developers/gemma-open-models/.

[13] "Gemma 2," 2025. 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, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *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," 2023. https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models.

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

[18] "Mistral," 2025. https://mistral.ai/.

[19] "Introducing Phi-3: Redefining what's possible with SLMs," 2025. https://azure.microsoft.com/en-us/blog/introducing-phi-3-redefining-whats-possible-with-slms/.

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

[21] 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 *ICLR*, 2022.

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

[23] "Building domain-specific llms: Examples and techniques," 2025. https://kili-technology.com/large-language-models-llms/building-domain-specific-llms-examples-and-techniques.

[24] "Getting started with customizing a large language model (llm)," 2025. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/customizing-llms.

[25] "Huggingface," 2025. https://huggingface.co.

[26] "Aws sagemaker," 2025. https://docs.aws.amazon.com/sagemaker/.

[27] "Introduction to serverless GPUs ," 2025. https://www.alibabacloud.com/help/en/functioncompute/fc-2-0/use-cases/introduction-to-serverless-gpus.

[28] "Kserve," 2025. https://github.com/kserve/kserve.

[29] "Nuclio: Automate the Data Science Pipeline with Serverless Functions ," 2025. https://nuclio.io/.

[30] 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.

[31] D. Zhang, H. Wang, Y. Liu, X. Wei, Y. Shan, R. Chen, and H. Chen, "Fast and live model auto scaling with o(1) host caching," *arXiv:2412.17246*, 2024.

[32] 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:2502.09922*, 2025.

[33] J. Hu, J. Xu, Z. Liu, Y. He, Y. Chen, H. Xu, J. Liu, B. Zhang, S. Wan, G. Dan, Z. Dong, Z. Ren, J. Meng, C. He, C. Liu, T. Xie, D. Lin, Q. Zhang, Y. Yu, H. Feng, X. Chen, and Y. Shan, "Deepflow: Serverless large language model serving at scale," *arXiv:2501.14417*, 2025.

[34] 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:2401.17644*, 2024.

[35] 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.

[36] "cuDNN." https://developer.nvidia.com/cudnn.

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

[38] 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:1712.06139*, 2017.

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

[40] 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.

[41] "SGLang." https://docs.sglang.ai/.

[42] Y. Sui, H. Yu, Y. Hu, J. Li, and H. Wang, "Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading," in *ACM SoCC*, 2024.

[43] 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:2306.03622*, 2024.

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

[45] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *NeurIPS*, 2019.

[46] 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 *ICML*, 2021.

[47] 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.

[48] 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 *NeurIPS*, 2018.

[49] 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 *NeurIPS*, 2024.

[50] 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.

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

[52] 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," in *ACL*, 2024.

[53] B. Butler, S. Yu, A. Mazaheri, and A. Jannesari, “Pipeinfer: Accelerating llm inference using asynchronous pipelined speculation,” in *SC*, 2024.

[54] 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.

[55] 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.

[56] 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.

[57] 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:2305.05920*, 2023.

[58] B. Jeon, M. Wu, S. Cao, S. Kim, S. Park, N. Aggarwal, C. Unger, D. Arfeen, P. Liao, X. Miao, M. Alizadeh, G. R. Ganger, T. Chen, and Z. Jia, “Graphpipe: Improving performance and scalability of dnn training with graph pipeline parallelism,” in *ACM ASPLOS*, 2025.

[59] “Microsoft Azure Serverless GPU.” https://learn.microsoft.com/azure/container-apps/gpu-serverless-overview/.

[60] 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,” *arXiv:2404.02015*, 2024.

[61] “Amazon EC2 On-Demand Pricing,” 2023. https://aws.amazon.com/ec2/pricing/on-demand/.

[62] “TensorFlow.” https://www.tensorflow.org/.

[63] S. Zeng, M. Xie, S. Gao, Y. Chen, and Y. Lu, “Medusa: Accelerating serverless llm inference with materialization,” in *ACM ASPLOS*, 2025.

[64] “Safetensors: Ml safer for all,” 2025. https://github.com/huggingface/safetensors/.

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

[66] “Kubernetes.” https://kubernetes.io/.

[67] “RyokoAI/ShareGPT52K · Datasets at Hugging Face,” 2023. https://huggingface.co/datasets/RyokoAI/ShareGPT52K.

[68] 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:2107.03374*, 2021.

[69] 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 *ACL*, 2024.

[70] 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 FAST*, 2025.

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

[72] 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.

[73] 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 SoCC*, 2023.

[74] 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.

[75] 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.

[76] 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.

[77] 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:2308.16369*, 2023.

A TPOT SLO Attainment

We provide the TPOT SLO attainment of systems under different CVs in Figure 16. All systems achieve over 95% TPOT SLO attainment in most scenarios, and more than 90% under all CVs and RPS configurations.

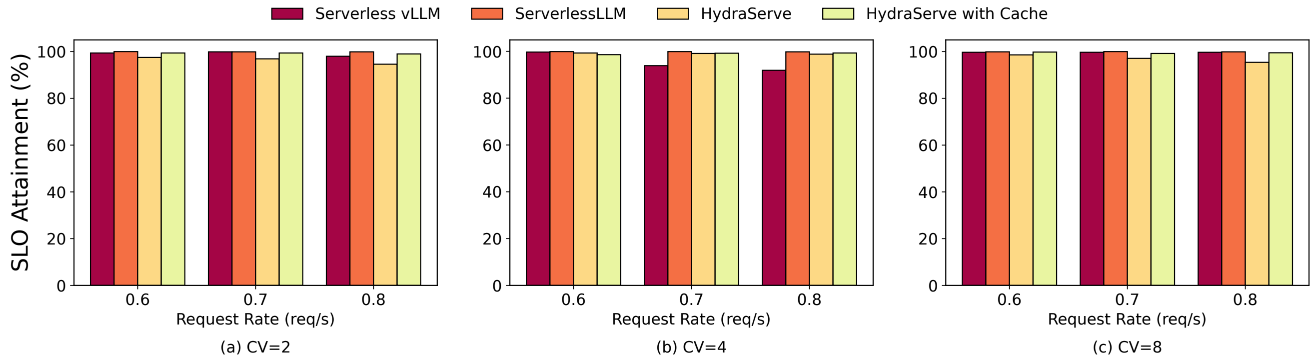


Figure 16: TPOT SLO attainment of systems under different CVs.