# ServerlessLLM

Low-latency serverless inference for large language models

# ServerlessLLM: Low-Latency Serverless Inference for Large Language Models

Yao Fu[1] Leyang Xue[1] Yeqi Huang[1] Andrei-Octavian Brabete[1] Dmitrii Ustiugov[2] Yuvraj Patel[1] Luo Mai[1]

[1]*University of Edinburgh* [2]*NTU Singapore*

## Abstract

This paper presents ServerlessLLM, a distributed system designed to support low-latency serverless inference for Large Language Models (LLMs). By harnessing the substantial near-GPU storage and memory capacities of inference servers, ServerlessLLM achieves effective local checkpoint storage, minimizing the need for remote checkpoint downloads and ensuring efficient checkpoint loading. The design of ServerlessLLM features three core contributions: (i) *fast multi-tier checkpoint loading*, featuring a new loading-optimized checkpoint format and a multi-tier loading system, fully utilizing the bandwidth of complex storage hierarchies on GPU servers; (ii) *efficient live migration of LLM inference*, which enables newly initiated inferences to capitalize on local checkpoint storage while ensuring minimal user interruption; and (iii) *startup-time-optimized model scheduling*, which assesses the locality statuses of checkpoints on each server and schedules the model onto servers that minimize the time to start the inference. Comprehensive evaluations, including microbenchmarks and real-world scenarios, demonstrate that ServerlessLLM dramatically outperforms state-of-the-art serverless systems, reducing latency by 10 - 200X across various LLM inference workloads.

## 1 Introduction

Large Language Models (LLMs) have recently been integrated into various online applications, such as programming assistants [26], search engines [21], and conversational bots [54]. These applications process user inputs, such as questions, by breaking them down into tokens (e.g., words). LLMs generate responses in an autoregressive manner, predicting each subsequent token based on the combination of input tokens and those already generated, until a sentence-ending token (EoS) is reached. To optimize this process, LLMs utilize key-value caches to store intermediate results, thereby minimizing redundant computations.

Serving LLMs at scale presents significant challenges due to the extensive GPU resources required and the stringent low response time constraints demanded by interactive services. Additionally, LLM inference latency is unpredictable as it depends on the output length, which varies significantly due to iterative token generation [27, 42, 81].

To achieve low latency, processing an LLM request often requires multiple GPUs for durations ranging from seconds to minutes. In practice, service providers hosting LLMs need to cater to a diverse range of developers, leading to substantial GPU consumption [20] and impacting the sustainability of LLM services [23]. Consequently, LLM inference services are compelled to impose strict limits on the number of requests users can send (e.g., 40 messages per 3 hours for ChatGPT [54]), highlighting the providers' current challenges in meeting demand. Researchers predict that LLM inference costs could escalate by more than 50 times as it approaches the popularity of Google Search [23].

To reduce GPU consumption, LLM service operators are turning to serverless inference, as demonstrated in platforms such as Amazon SageMaker [63], Azure [50], KServe [16], and HuggingFace [35]. In this model, developers upload their LLM checkpoints, which include both model execution and parameter files, to a checkpoint storage system. When a request is received, a model loading scheduler selects available GPUs to initiate these checkpoints. A request router then directs the inference request to the selected GPUs. This serverless approach allows infrastructure providers to efficiently multiplex LLMs on GPUs, improving resource utilization. Additionally, it offers economic advantages to infrastructure users, who incur costs only for each request's duration, thereby avoiding expensive long-term GPU reservations.

While serverless inference offers cost savings for deploying LLMs, it also introduces significant latency overheads. These overheads are commonly attributed to inference cold starts, a frequent issue in serverless workloads, as demonstrated by public traces [64, 92]. Cold starts are especially prolonged for LLM checkpoints, whose sizes can range from gigabytes [11, 76, 91] to terabytes [29]. The vast size is due to the immense number of parameters in such models, leading to notable delays when downloading from remote storage.

Moreover, these checkpoints consist of numerous tensors, each with unique structures and sizes. The complex process of loading these tensors onto GPUs, which involves file deserialization, memory allocation, and tensor shape parsing, further compounds these delays.

We aim to explore system designs that support low-latency serverless inference for LLMs. We note that GPU-based inference servers typically feature a sophisticated yet underutilized storage hierarchy, equipped with extensive host memory and storage capacities. Current serverless inference systems, such as KServe [16] and Ray Serve [73], often only utilize a fraction of the available host memory and minimally employ SSDs for caching checkpoints from the model repository. This observation has led us to propose a novel system design: leveraging the multi-tier storage hierarchy for local checkpoint storage and harnessing their significant storage bandwidth for efficient checkpoint loading.

However, several open concerns arise when implementing local checkpoint storage: (i) Given the complex storage architecture of a GPU server, which includes multiple GPUs, DRAM, SSDs, and remote storage, all interconnected through various links such as PCIe, NVMe, and network connections, how can we optimize the loading of LLM checkpoints to fully exploit the available bandwidth? (ii) Assigning requests to servers with pre-loaded checkpoints can avoid the need for remote checkpoint downloads, but this strategy might lead to prolonged queuing delays or high preemption costs. This is particularly challenging as LLMs typically have long, unpredictable inference durations, which differ markedly from traditional deep neural network inference. (iii) In a distributed cluster where model requests are concurrently served and checkpoints are preloaded onto various layers of local storage, which servers should be strategically selected to minimize the time required to start a model inference?

To address the above, we have designed and implemented ServerlessLLM, which includes three core contributions:

**(1) Fast multi-tier checkpoint loading.** ServerlessLLM can maximize the storage bandwidth usage of GPU servers for LLM checkpoint loading. It introduces (i) a new *loading-optimized checkpoint* that supports sequential, chunk-based reading and efficient tensor in-memory addressing, and (ii) an *efficient multi-tier checkpoint storage system* that can harness the substantial capacity and bandwidth on a multi-tier storage hierarchy, through an in-memory data chunk pool, memory-copy efficient data path, and a multi-stage data loading pipeline.

**(2) Efficient live migration of LLM inference.** We motivate the need for live migration of LLM inference and are the first to implement LLM live migration in serverless inference systems to enhance the performance when supporting locality-driven inference. To achieve high efficiency when migrating LLM inference, we have implemented two strategic designs: (i) the source server migrates only the tokens, rather than the large kv-cache, which significantly reduces network traffic during the migration; and (ii) it triggers an efficient re-computation of the kv-cache at the destination server, ensuring migration can complete in a timely manner.

**(3) Startup-time-optimized model scheduling.** ServerlessLLM aids serverless inference systems by enabling latency-preserving, locality-aware model scheduling. It integrates cost models for accurately estimating the time of loading checkpoints from different tiers in the storage hierarchy and the time of migrating an LLM inference to another server. Based on the estimation results, Phantom can choose the best server to minimize model startup latency.

We have conducted comprehensive evaluation to compare ServerlessLLM against various baseline methods in a GPU cluster. Micro-benchmark results revealed that ServerlessLLM's LLM checkpoint loading significantly outperforms existing systems such as Safetensors [36], and PyTorch [58], achieving loading times that are 3.6 - 8.2X faster. This performance enhancement is particularly notable with large LLMs like OPT [91], LLaMA-2 [76], and Falcon [11]. ServerlessLLM also supports emerging LoRA adaptors [34], achieving 4.4X speed-ups in checkpoint loading.

Furthermore, we evaluated ServerlessLLM with real-world serverless workloads, modeled on the public Azure Trace [64], and benchmarked it against KServe, Ray Serve, and a Ray Serve variant with local checkpoint caching. In these scenarios, ServerlessLLM demonstrated a 10 to 200 times improvement in latency for running OPT model inferences across datasets (i.e., GSM8K [25] and ShareGPT [83]). These results underscore ServerlessLLM's effectiveness in combining fast checkpoint loading, efficient inference migration, and optimized scheduling for model loading. The source code for ServerlessLLM is released at https://github.com/ServerlessLLM/ServerlessLLM.

## 2 Background and Motivation

### 2.1 Why Serverless Inference for LLMs

Numerous companies, including Amazon, Azure, Google, HuggingFace, Together AI [6], Deepinfra [2], Replicate [5], Databricks [7], Fireworks-AI [3], and Cohere [4], have introduced serverless inference services (also known as serverless model entrypoints). These services enable users to deploy standard open-source LLMs either in their original form or by modifying them through fine-tuning or by running custom-built models.

Serverless inference can significantly reduce costs for LLM users by charging only for the duration of inference and the volume of processed data. These serverless platforms also offer functionalities such as auto-scaling and auto-failure-recovery to keep instances in an "always-on" state. For the providing companies, serverless inference allows effective

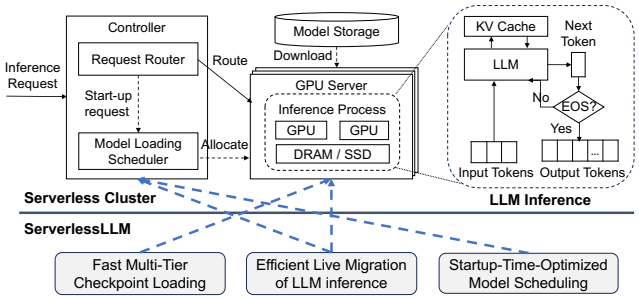



Figure 1: Overview of GPU serverless clusters, LLM inference and new designs introduced by ServerlessLLM.

multiplexing of models within a GPU cluster, improving resource utilization, and generating a software premium for managing infrastructure on behalf of users.

Serverless inference systems are especially advantageous for LLM applications with dynamic and unpredictable workloads. These may include newly launched products without clear predictions of user engagement (e.g., the launch of the ChatGPT service) or those facing spontaneous and unpredictable demands, which are typical in sectors such as healthcare, education, legal, and sales. Unlike global-scale LLM services, these applications are activated only when users access the LLM service.

## 2.2 Serverless Cluster and LLM Inference

We introduce the key components in GPU serverless clusters in Figure 1. Upon receiving a new inference request, the controller dispatches it to GPU-equipped nodes in a cluster running LLM inference service instances, and to cloud storage hosting model checkpoints. The controller typically consists of two main components: the *request router* and the *model loading scheduler*. The request router directs incoming requests to nodes already running LLM inference processes, or instructs the model loading scheduler to activate LLM inference processes on unallocated GPUs. The selected GPU node initiates a GPU process/container, setting up an inference library (e.g., HuggingFace Accelerate [32] and vLLM [42]). This inference process involves downloading the requested model's checkpoint from a remote model storage and loading it into the GPU, passing through SSD and DRAM.

The LLM inference process often handle requests that include user-specified input prompts, i.e., a list of tokens, as shown in Figure 1. This process iteratively generates tokens based on the prompt and all previously generated tokens, continuing until an end-of-sentence token (denoted as EoS) is produced, resulting in non-deterministic total inference time [55]. During each iteration, the LLM caches intermediate computations in a KV-cache to accelerate subsequent token generation [42, 56]. The tokens generated by each iteration are continuously streamed back to the requesting client, making LLM applications interactive by nature. Their performance is thus measured by both first-token latency (i.e., the time to return the first token) and per-token latency (i.e., the average time to generate a token).

## 2.3 Challenges with Serverless LLM Inference

The deployment of LLMs on serverless systems, although promising, often incurs significant latency overheads. This is largely due to the substantial proportions of cold-start in serverless clusters, as demonstrated by public data: the Azure Trace [64] shows that over 40% of functions exhibit a cold-start rate exceeding 25%, and approximately 25% of functions experience a cold-start rate greater than 60%, within a 5-minute keep-alive interval. These figures align with the findings from our experiments, underscoring the impact of cold-starts in real-world settings. Consequently, many serverless providers, including Bloomberg, have publicly acknowledged experiencing extremely high latencies, often reaching tens of seconds, when initializing state-of-the-art LLMs for inference on their platforms.

We observe several primary reasons for the prolonged LLM cold-start latency:

**(1) LLM checkpoints are large, prolonging downloads.** LLM checkpoints are significantly larger than conventional DNN checkpoints, which leads to longer download times. For instance, Grok-1 [82] checkpoints are over 600 GB, DBRX [72] are 250GB, and Mixtral-8x22B [71] are about 280GB [1] Downloading such large checkpoints from remote storage becomes costly. For example, acquiring an LLM checkpoint with a size of 130GB (e.g., LLaMA-2-70B [76]) from S3 or blob storage takes a minimum of 26 seconds using a fast commodity network capable of 5GB/s [19].

**(2) Loading LLM checkpoints incurs a lengthy process.** Even when model checkpoints are stored locally on NVMe SSDs, loading these checkpoints into GPUs remains a complex process including model initialization, GPU memory allocation, tensor creation, and tensor data copy, typically taking tens of seconds (as detailed in §7.2). For instance, loading the OPT-30B model into 4 GPUs requires 34 seconds using PyTorch, and loading LLaMA-2-70B into 8 GPUs takes 84 seconds. This loading latency far exceeds the time required for generating a token during the inference process, which is usually less than 100ms [55]. Consequently, the prolonged first-token latency can significantly disrupt user experience.

## 2.4 Existing Solutions and Associated Issues

To improve the latency performance when supporting LLMs, existing solutions show a variety of issues:

**(1) Over-subscribing GPUs.** The prevalent solutions [13, 84], aimed at circumventing model download and loading times

[1] Model size calculated in float16 precision.

in serverless inference clusters, frequently involve oversubscribing GPUs to accommodate peak demand scenarios. For instance, AWS Serverless Inference [13] maintains a certain number of GPU instances in a warmed state to alleviate the impacts of slow cold starts. While this strategy is effective for managing conventional smaller models, such as ResNet and BERT, it proves challenging for LLMs, which require substantially greater resources from costly GPUs.

**(2) Caching checkpoints in host memory.** Several solutions [33, 39] have been developed that cache model checkpoints in the host memory of GPU servers to eliminate the need for model downloads. This approach is typically effective for smaller conventional models (e.g., up to a few GBs [39]). However, solely relying on host-memory-based caching proves inadequate for LLMs. LLMs can easily exceed hundreds of GBs in size, challenging the capacity of host memory to store a sufficient number of their checkpoints adequately. The limited size of host memory leads to significant cache misses, resulting in frequent model downloads, as further discussed in §7.4.

**(3) Deploying additional storage servers.** Various strategies [19] recommend the deployment of additional storage servers within a local cluster to cache model checkpoints. Despite these enhancements, recent trace studies [19] indicate that model downloads can exceed 20 seconds through an optimized pipeline, even when connected to local commodity storage servers equipped with a 100 Gbps NIC. Although the integration of faster networks (e.g., 200 Gbps Ethernet or InfiniBand) could reduce this latency, the associated costs of implementing additional storage servers and high-bandwidth networks are substantial [18, 31]. For instance, utilizing network-optimized AWS ElasticCache servers [1] to support a 70B model can lead to a 100% increase in costs. Specifically, cache.c7gn.16xlarge servers, which provide 210GB of memory and 200 Gbps of network performance, are priced at $16.3/h, equivalent to the cost of an 8-GPU g5.48xlarge server.

# 3 Exploiting In-Server Multi-Tier Storage

ServerlessLLM addresses the challenges highlighted in the previous sections—namely, high model download times and lengthy model loading—using a design approach that is cost-effective, scalable, and long-term viable.

## 3.1 Design Intuitions

Our design is inspired by the simple observation that GPU servers used for inference feature a multi-tier storage hierarchy with substantial capacity and bandwidth. From a capacity standpoint, these servers are equipped with extensive memory capabilities. For example, a contemporary 8-GPU server can support up to 4 TBs of main memory, 64 TBs on NVMe SSDs, and 192 TBs on SATA SSDs [52]. Additionally, we observe that in the serverless inference context, a significant portion of the host memory and storage devices in GPU servers remains underutilized.

Regarding bandwidth, GPU servers typically house multiple GPUs, each connected to the host memory via a dedicated PCIe connection, providing significant aggregated bandwidth between the memory and GPU. NVMe and SATA SSDs also connect through their respective links and can be configured in RAID to enhance throughput. For instance, an 8-GPU server utilizing PCIe 5.0 technology can achieve an aggregated bandwidth of 512 GB/s between the host memory and GPUs, and around 60 GB/s from NVMe SSDs (RAID 0) to host memory.

Building on these observations, we propose a design approach that leverages the unused in-server multi-tier storage capacity to store models locally and load them more rapidly, thus reducing latency. This approach is (i) *cost-effective*, as it reutilizes existing, underutilized storage resources in GPU servers; (ii) *scalable*, given that the available local storage capacities and bandwidth can naturally increase with the addition of more inference servers; and (iii) *long-term viable*, as upcoming GPU servers will include even greater capacities and bandwidth (*e.g.*, each Grace-Hopper GPU features 1 TB on-chip DRAM and a 900GB/s C2C link between on-chip DRAM and HBM).

## 3.2 Design Concerns and Overview

In implementing our design, we identify three crucial concerns that must be addressed.

**(1) Support complex multi-tiered storage hierarchy.** Current checkpoint and model loading tools such as PyTorch [58], TensorFlow [75], and ONNX Runtime [62] are primarily designed to enhance the training and debugging phases of model development. However, these tools are not optimized for read performance, which becomes critically important in a serverless inference environment. In these settings, model checkpoints are stored once but need to be frequently loaded and accessed across multiple GPUs. This insufficient optimization for read operations results in significant loading delays. While solutions like Safetensors [36] can enhance loading performance, as demonstrated in Section 7, they still fail to fully leverage the capabilities of a multi-tiered storage hierarchy.

**(2) Strong locality-driven inference.** Supporting efficient model loading alone is insufficient; we also need approaches that can effectively schedule requests onto GPU servers with locally stored checkpoints. Implementing locality-driven LLM inference, however, presents challenges. Current ML model serving systems such as ClockWork [33] and Shepherd [90] take checkpoint locality into account. Yet, they either depend on accurate predictions of model inference time, which is problematic with LLMs, or they preempt ongoing model inferences, causing significant downtime and redun-

dant computations. Therefore, ServerlessLLM must adopt a new approach that is tailored to the unique characteristics of LLM inference (*i.e.*, this workload is interactive and features long, unpredictable durations), necessitating the support for inference live migration, which is further detailed in Section 5.

**(3) Scheduling models for optimized startup time.** ServerlessLLM is designed to minimize the model startup latency. The cluster scheduler (or controller) plays a crucial role in scheduling models onto GPU resources to answer incoming inference requests. However, the scheduler needs to carefully consider the checkpoint's locality in the entire cluster. Many factors may influence the overall startup latency, such as the difference in the bandwidth offered by each layer in the memory hierarchy. There may be instances where it is beneficial to move the current inference execution to a new GPU than to allocate the request to a GPU where the model may have to be loaded from the storage media. Hence, ServerlessLLM needs to accurately estimate the startup times considering the cluster's checkpoint locality status and accordingly allocate resources to minimize startup time.

**Overview.** ServerlessLLM addresses these concerns with three novel designs, as depicted in Figure 1. Firstly, it facilitates fast multi-tier checkpoint loading (Section 4) to fully utilize the storage capacity and bandwidth of each GPU server. It also coordinates GPU servers and the cluster controller for efficient live migration of LLM inference (Section 5), ensuring locality-driven inference with minimal resource overhead and user disruption. Lastly, ServerlessLLM features a startup-time-optimized model scheduling policy (Section 6) implemented in its controller, effectively analyzing the checkpoint storage status of each server within a cluster, and it chooses a server for initiating a model, minimizing its startup time.

## 4 Fast Multi-Tier Checkpoint Loading

In this section, we introduce the design of fast multi-tier checkpoint loading in ServerlessLLM, with several key objectives: (i) to fully utilize the bandwidth and capacity of multi-tier local storage on GPU servers, (ii) to ensure predictable loading performance, critical for ServerlessLLM's readiness in low-latency inference clusters, and (iii) to maintain a generic design that supports checkpoints from various deep learning frameworks.

### 4.1 Loading-Optimized Checkpoints

Our design is motivated by the observation that LLM checkpoints are often written frequently during training and debugging but loaded infrequently. Conversely, in serverless inference environments, checkpoints are uploaded once and loaded multiple times. This discrepancy has inspired us to convert these checkpoints into a loading-optimized format.

To ensure our design is generic for different frameworks,

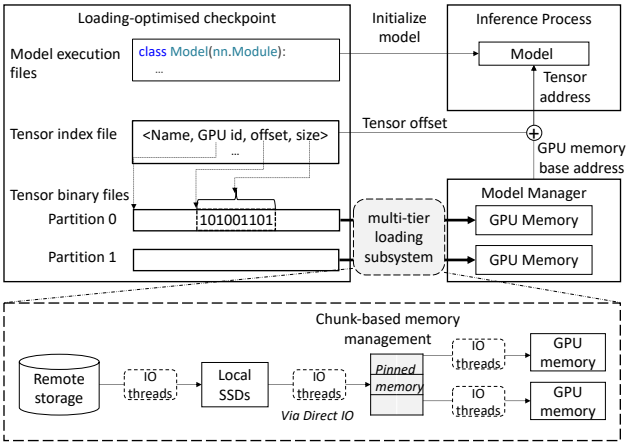



Figure 2: Components in fast multi-tier checkpoint loading.

we operate under a set of assumptions that are common in checkpoints. The checkpoints have: (i) *Model execution files* which define the model architecture. Depending on the framework, the format varies; TensorFlow typically uses protobuf files [74], while PyTorch employs Python scripts [57]. Beyond architecture, these files detail the size and shape of each tensor and include a model parallelism plan. This plan specifies the target GPU for each tensor during checkpoint loading. (ii) *Model parameter files* which stores the binary data of parameters in an LLM. Tensors within these files can be arranged in any sequence. Runtimes such as PyTorch may also store tensor shapes as indices to calculate the offset and size for each tensor.

To ensure fast loading performance, we implement two main features for the converted checkpoints: (i) *Sequential chunk-based reading*: To ensure efficient sequential reading, tensors for each GPU are grouped in partitions (shown in Figure 2). These files contain only the binary data of model parameters and exclude metadata such as tensor shapes, facilitating large chunk reading. (ii) *Direct tensor addressing*: We create a tensor index file (shown in Figure 2) that maps tensor names to a tuple of GPU id, offset, and size, facilitating the efficient restoration of tensors. The tensors are aligned with memory word sizes, facilitating direct computation of memory address.

We observe that decoupling the loading and inference processes can further enhance loading performance. This separation allows checkpoint loading to be pre-scheduled and overlapped with the initialization of the inference process. For this, ServerlessLLM uses a model manager to load tensor data, while allowing the inference process to focus on initializing the model by setting the data pointers for each tensor. More specifically, the model manager allocates memory on GPUs and loads the binary data of the checkpoint via a fast multi-tier loading subsystem (see details in 4.2). The inference process initializes the model object and sets the GPU memory address for each tensor. It acquires the base addresses for each GPU

(i.e., CUDA IPC handles) from the model manager and reads the tensor offset from the tensor index file, facilitating the computation of the tensor GPU memory address (i.e., *base + offset*). To ensure the model is fully initialized before inference, the inference process and the model manager perform a synchronization.

## 4.2 Multi-Tier Loading Subsystem

To achieve fast and predictable checkpoint loading performance, we design a multi-tier loading subsystem, integrated within the model manager. This subsystem incorporates several techniques:

**Chunk-based data management.** For fast loading performance, we have implemented chunk-based data management with three main features: (i) *Utilizing parallel PCIe links*. To mitigate the bottleneck caused by a single PCIe link from storage when loading multiple models into GPUs, we employ parallel DRAM-to-GPU PCIe links to facilitate concurrent checkpoint loading across GPUs. (ii) *Supporting application-specific controls*. Our memory pool surpasses simple caching by providing APIs for the allocation and deallocation of memory. This enables fine-grained management of cached or evicted data chunks, based on specific requirements of the application. (iii) *Mitigating memory fragmentation*. We address latency and space inefficiencies caused by memory fragmentation by using fixed-size memory chunks.

**Predictable data path.** We have created an efficient data path in our model manager with two main strategies: (i) *Exploiting direct file access*. We use direct file access (e.g., 'O_DIRECT' in Linux) to avoid excessive data copying by directly reading data into user space. This method outperforms memory-mapped files (mmap), currently adopted in high-speed loaders such as Safetensors [36], which rely on system cache and lack consistent performance guarantees (critical for predictable performance). (ii) *Exploiting pinned memory*. We utilize pinned memory to eliminate redundant data copying between DRAM and GPU. This approach allows direct copying to the GPU with minimal CPU involvement, ensuring efficient use of PCIe bandwidth with a single thread.

**Multi-tier loading pipeline.** We have developed a multi-tier loading pipeline to support various storage interfaces and improve loading throughput. This pipeline has three features: (i) *Support for multiple storage interfaces*. ServerlessLLM offers dedicated function calls for various storage interfaces, including local storage (e.g., NVMe, SATA), remote storage (e.g., S3 object store [12]), and in-memory storage (pinned memory). It utilizes appropriate methods for efficient data access in each case. (ii) *Support for intra-tier concurrency*. To leverage modern storage devices' high concurrency, ServerlessLLM employs multiple I/O threads for reading data within each storage tier, improving bandwidth utilization. (iii) *Flexible pipeline structure*. We use a flexible task queue-based pipeline design, supporting new storage tiers to be efficiently integrated. I/O threads read storage chunks and enqueue their indices (offset and size) for the I/O threads in the next tier.

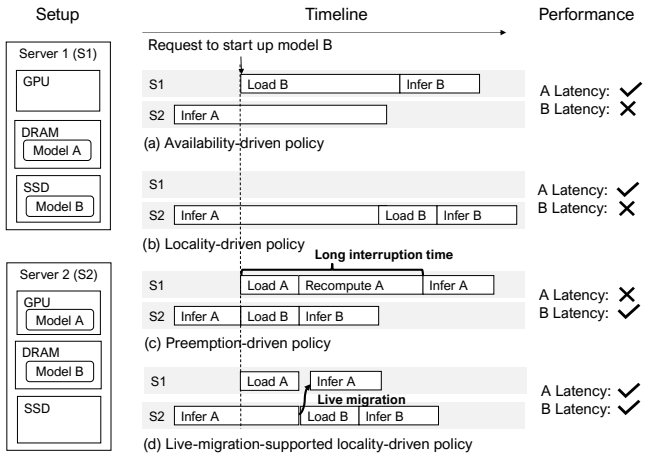



Figure 3: Analysis of different locality-driven policies

# 5 Efficient Live Migration of LLM Inference

In this section, we describe why live migration is the key to effective locality-driven LLM inference, and how to make such a live migration process particularly efficient.

## 5.1 Need for Live Migration

We consider a simple example to analyze the performance of different current approaches in supporting the checkpoint locality. In this example, we have two servers (named Server 1 and Server 2) and two models (named Model A and Model B), as illustrated in Figure 3. Server 1 currently has Model A in DRAM and Model B in SSD and its GPU is idle, while Server 2 currently has Model B in DRAM, and its GPU is running the inference of Model A.

In Figure 3, we analyze the performance of potential policies for starting up Model B. Our analysis is based on their impact on the latency performance of both Model A and B:

- *Availability-driven policy* chooses Server 1 currently with an available GPU, and it is agnostic to the location of Model B. As a result, the Model B's startup latency suffers while the Model A remains unaffected.

- *Locality-driven policy* opts for the locality in choosing the server and thus launching Model B on Server 2. However, it waits for Model A to complete, making Model B suffer from a long queuing delay. Furthermore, the locality policy leaves Server 1 under-utilized, preventing all servers from being fully utilized.

- *Preemption-driven policy* preempts Model A on Server 2 and startups Model B. It identifies that Server 1 is free and reinitiates Model A there. This policy reduces Model B's latency but results in significant downtime for Model A when it performs reloading and recomputation.
- *Live-migration-supported locality-driven policy* prioritizes locality without disrupting Model A. It initially preloads Model A on Server 1, maintaining inference operations. When Model A is set on Server 1, its intermediate state is transferred there, continuing the inference seamlessly. Following this, Model B commences on Server 2, taking advantage of locality. This policy optimizes latency for both Models A and B.

According to the examples above, live migration stands out in improving latency for both Model A and Model B among all locality-driven policies.

## 5.2 Making Live Migration Efficient

We aim to achieve efficient live migration of LLM inference, incurring minimal resource overhead and minimal user interruption. We initially considered using the snapshot method from Singularity [68], which involves snapshotting the LLM inference. However, this method is slow due to lengthy snapshot creation and transfer times (e.g., typically 10s seconds or even minutes). Dirty-page-based migration might be considered to accelerate virtual machine migration, but this approach is currently not supported in GPU-enabled containers and virtual machines. Hence, we decided to explore live migration methods that can be easily implemented in applications.

To make the live migration method effective for LLM inference, we aim to achieve two objectives: (i) the migrated inference state must be minimal to reduce network traffic, and (ii) the destination server must quickly synchronize with the source server's progress to minimize migration times.

For (i), we propose to migrate tokens (typically 10-100s KB) instead of the large KV-Cache (typically 1-10s GB), as recomputing the KV-Cache based on the migrated tokens on the destination GPU is generally much faster than transferring the dirty state over the network. In certain conditions (e.g., given high-bandwidth network and short input sequences), migrating KV-Cache might also be fast yet it still increases cluster network traffic compared to migrating tokens.

For (ii), we leverage an insight from LLM inference: recomputing the KV-Cache for current tokens on the destination GPU is significantly faster (usually an order of magnitude shorter) than generating an equivalent number of new tokens on the source GPU. This approach facilitates efficient convergence of multi-round token-based migration, with the quantity of tokens generated on the source diminishing with each round. For example, time to recompute the KV-Cache for 1000 tokens equals to the time to generate about 100 new tokens according to [70].

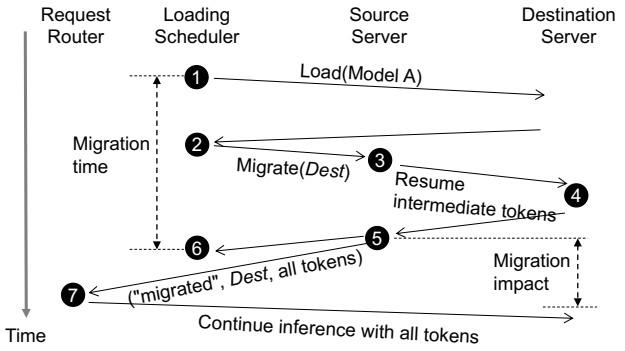



Figure 4: Live migration process for LLM inference

## 5.3 Multi-Round Live Migration Process

We implement the above proposal as a multi-round live migration process. In each migration round (step ❸, ❹ and ❺), the destination server (referred to as the *dest* server) recomputes the KV cache using the intermediate tokens sent by the source server (referred to as the *src* server). When the gap (i.e., the tokens generated after the last round) between the source server and the destination server is close enough, the *src* server stops generating and sends all tokens to the *dest* via the request router, ensuring minimal interruption on ongoing inference during migration. This migration process is depicted in Figure 4 with its steps defined below:

1. The model loading scheduler sends a model loading request to *dest* server to load model A into GPUs. If there is an idle instance of model A on *dest* server, the scheduler skips this step.
2. After loading, the scheduler sends a migration request carrying the address of *dest* server to *src* server.
3. Upon receiving a migrate request, *src* server sets itself as "migrating", sends a resume request with intermediate tokens (i.e., input tokens and the output tokens produced before step 3) to *dest* server if the inference is not completed. Otherwise, it immediately returns to the scheduler.
4. *dest* server recomputes KV cache given the tokens in the resume request.
5. Once *resume* request is done, *src* server stops inference, returns to the scheduler, and replies to the request router with all tokens (i.e., the intermediate tokens together with the remaining tokens produced between step 3 and step 5) and a flag "migrated".
6. The scheduler finishes the migration, unloads model A at *src* server and starts loading model B.
7. The request router checks the flag in the inference response. If it is "migrated", the request router replaces *src* server with *dest* server in its route table and sends all tokens to *dest* server to continue inference.

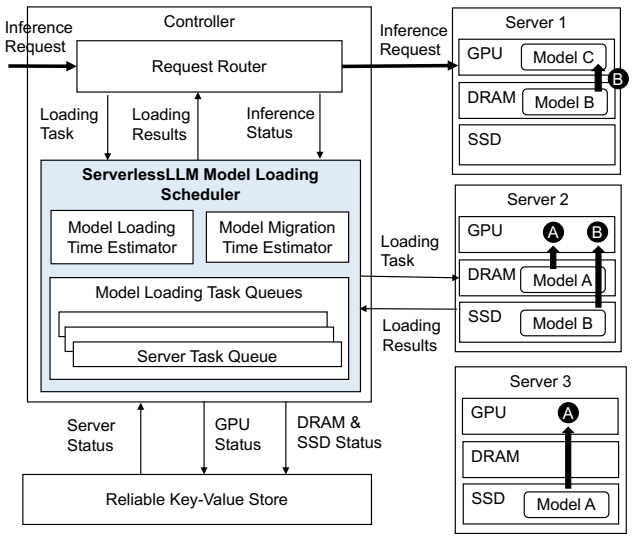



Figure 5: Overview of the model loading scheduler design

## 5.4 Practical Concerns

**Handling inference completion.** The autoregressive nature of LLM inference may lead to task completion at *src* server between steps ❸ and ❺. In such cases, *src* server informs the request router of the inference completion as usual. Additionally, it notifies the loading scheduler, which then instructs *dest* server to cease resuming, terminating the migration.

**Handling server failures.** ServerlessLLM can manage server failures during LLM inference migration. In scenarios where *src* server fails, if the failure happens during loading (i.e., before step ❷ in Figure 4), the scheduler aborts the migration and unloads the model from the destination. If the failure occurs during migration (i.e., between steps ❷ and ❸), the scheduler directs the destination to clear any resumed KV cache and unload the model.

In cases where *dest* server fails, if the failure takes place during loading, the migration is canceled by the scheduler. Should the failure occur while resuming, the source notifies the scheduler of the failure and continues with the inference.

# 6 Startup-Time-Optimized Model Scheduling

In this section, we describe the design of the startup-time-optimized model scheduling implemented in ServerlessLLM's cluster scheduler (denoted as controller), as shown in Figure 5. This scheduler processes loading tasks from the request router and employs two key components: a model loading time estimator and a model migration time estimator. The former assesses loading times from various storage media, while the latter estimates times for necessary model migrations. For example, as shown in Figure 5, the scheduler calculates the time to load Model A (indicated by Ⓐ) from different servers' DRAM and SSD, aiding in server selection. Similarly, for Model B (Ⓑ), it assesses whether to migrate Model C to another server or load Model B from Server 2's SSD.

To ensure robust time estimation, the ServerlessLLM scheduler employs distinct loading task queues for each server, effectively mitigating the impact of contentions caused by concurrent loading activities. Upon assigning a task, it promptly updates the server status—including GPU and DRAM/SSD states—in a reliable key-value store (e.g., , etcd [30] and ZooKeeper [38]). This mechanism enables ServerlessLLM to maintain continuity and recover efficiently from failures.

## 6.1 Estimating Model Loading Time

To estimate the time needed to load models from different storage tiers, we consider three primary factors: (i) *queuing time* ($q$), which is the wait time for a model in the server's loading task queue. This occurs when other models are pending load on the same server; (ii) *model size* ($n$), the size of the model in bytes, or its model partition in multi-GPU inference scenarios; (iii) *bandwidth* ($b$), the available speed for transferring the model from storage to GPUs. ServerlessLLM tracks bandwidth for network, SSD, and DRAM, allowing us to calculate loading time as $q + n/b$. Here, $q$ accumulates from previous estimations for the models already in the queue.

For precise estimations, we have implemented: (i) Sequential model loading per server, with single I/O queues for both Remote-SSD and SSD-DRAM paths (since these paths are shared by multiple GPUs on a server), reducing bandwidth contention which complicates estimation; (ii) In multi-tier storage, ServerlessLLM uses the slowest bandwidth for estimation because of ServerlessLLM's pipeline loading design. For example, when SSD and DRAM are both involved, SSD bandwidth is the critical bottleneck since it is orders of magnitude slower than DRAM; (iii) The scheduler monitors the loading latency returned by the servers. It leverages the monitoring metrics to continuously improve its estimation of the bandwidth through different storage media.

## 6.2 Estimating Model Migration Time

For live migration time estimation, our focus is on model resuming time (as shown in step ❹ in Figure 4), as this is significantly slower (seconds) than token transfer over the network (milliseconds). We calculate model resuming time considering: (i) *input tokens* ($t_{in}$), the number of tokens in the LLM's input prompt; (ii) *output tokens* ($t_{out}$), the tokens generated so far; and (iii) *model-specific parameters* ($a$ and $b$), which vary with each LLM's batch sizes and other factors, based on LLM system studies like vLLM [42]. With all the above factors, we can compute the model resuming time as $a \times (t_{in} + t_{out}) + b$.

However, obtaining real-time output tokens from servers for the scheduler can lead to bottlenecks due to excessive server interactions. To circumvent this, we developed a method where the scheduler queries the local request router for the inference status of a model, as illustrated in Figure 5. With the inference duration ($d$) and the average time to produce a token ($t$), we calculate $t_{out} = d/t$.

For selecting the optimal server for model migration, ServerlessLLM employs a dynamic programming approach to minimize migration time.

## 6.3 Practical Concerns

**Selecting best servers.** Utilizing our time estimations, ServerlessLLM evaluates all servers for loading the forthcoming model, selecting the one offering the lowest estimated startup time. The selection includes the server ID and GPU slots to assign. If no GPUs are available, even after considering migration, the loading task is held pending and retried once the request router informs the scheduler to release GPUs.

**Handling scheduler failures.** ServerlessLLM is built to withstand failures, utilizing a reliable key-value store to track server statuses. On receiving a server loading task, its GPU status is promptly updated in this store. Post server's confirmation of task completion, the scheduler updates the server's storage status in the store. Once recorded, the scheduler notifies the request router of the completion, enabling request routing to the server. In the event of a scheduler failure, recovery involves retrieving the latest server status from the key-value store and synchronizing it across all servers.

**Scaling schedulers.** The performance of the loading scheduler has been significantly enhanced by implementing asynchronous operations for server status reads, writes, and estimations. Current benchmarks demonstrate its capability to handle thousands of loading tasks per second on a standard server. Plans for its distributed scaling are earmarked for future development.

**Resource fairness.** ServerlessLLM treats all models with equal importance and it ensures migrations do not impact latency. While we currently adopt sequential model loading on the I/O path, exploring concurrent loading on servers with a fairness guarantee is planned for future work.

**Estimator accuracy.** Our estimator can continuously improve their estimation based on the monitored loading metrics returned by the servers. They offer sufficient accuracy for server selection, as shown in Section 7.

# 7 Evaluation

This section offers a comprehensive evaluation of ServerlessLLM, covering three key aspects: (i) assessing the performance of our loading-optimized checkpoints and model manager, (ii) examining the efficiency and overheads associated with live migration for LLM inference, and (iii) evaluating ServerlessLLM against a large-scale serverless workload, modelled on real-world serverless trace data.

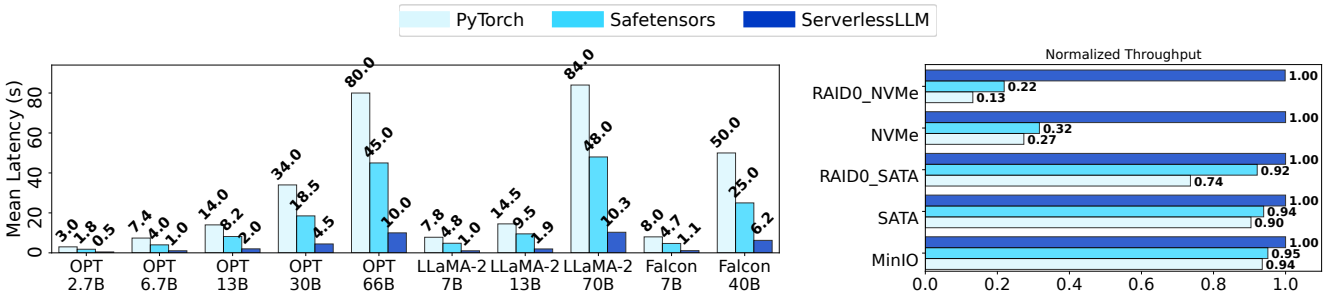
## 7.1 Evaluation Setup

**Setup.** We have two test beds: (i) a GPU server has 8 NVIDIA A5000 GPUs, 1TB DDR4 memory and 2 AMD EPYC 7453 CPUs, two PCIe 4.0-capable NVMe 4TB SSDs (in RAID 0) and two SATA 3.0 4TB SSDs (in RAID 0). This server is connected to a storage server via 1 Gbps networks on which we have deployed MinIO [51], an S3 compatible object store; (ii) a GPU cluster with 4 servers connected with 10 Gbps Ethernet connections. Each server has 4 A40 GPUs, 512 GB DDR4 memory, 2 Intel Xeon Silver 4314 CPUs and one PCIe 4.0 NVMe 2TB SSD.

**Models.** We use state-of-the-art LLMs, including OPT [91], LLaMA-2 [76] and Falcon [11] in different sizes. For cluster evaluation (§7.3 and §7.4) on test bed (ii), following prior work [44], we replicate OPT-6.7B/OPT-13B/OPT-30B models for 32/16/8 instances respectively (unless otherwise indicated) that are treated as different models during evaluation.

**Datasets.** We use real-world LLM datasets as the input to models. This includes GSM8K [25] that contains problems created by human problem writers, and ShareGPT [83] that contains multilanguage chat from GPT4. Since the models we used can handle at most 2048 context lengths, we truncate the input number of tokens to the max length. We also randomly sample 4K samples from each dataset to create a mixed workload, emulating real-world inference workloads.

**Workloads.** Since there are no publicly available LLM serverless inference workloads, we use Azure Serverless Trace [64] which is a representative serverless workload used in recent serverless studies [61] and model-serving studies [44, 90]. We designate functions to models and creates bursty request traces (CV=8 using Gamma distribution), following the workload generation method used in AlpaServe [44]. We then scale this trace to the desired requests per second (RPS). For cluster evaluation, we replicate each model based on its popularity and distribute them across nodes' SSDs using round-robin placement until the total cluster-wide storage limit is reached. Optimization of checkpoint placement is considered a separate issue and is not addressed in this paper. For all experiments (unless we indicate otherwise), we report the model startup latency, a critical metric for serverless inference scenarios. When migration or preemption is enabled, this latency is added with pause latency, accounting for the impacts of delays.




(a) Checkpoint loading speed.

(b) Normalized bandwidth utilization.

Figure 6: Checkpoint loading performance.

## 7.2 ServerlessLLM Checkpoint Loading

We now evaluate the model manager's effectiveness in reducing the model loading latency. For our experiments, we test the checkpoint read on test bed (i). We record reads from 20 copies of each model checkpoint to get a statistically significant performance report. We clear the page and inode caches after checkpoint copies are made to ensure a cold start. For each type of model, we randomly access the 20 copies to simulate real-world access patterns.

**Loading performance.** We aim to quantify the performance gains achieved by the ServerlessLLM checkpoint manager. We compare PyTorch [58] and Safetensors [36], representing the read-by-tensor checkpoint loading and mmap-based checkpoint loading, respectively. We use all types of models with all checkpoints in FP16 and run the test on RAID0-NVMe SSD having a throughput of 12 GB/s.

Figure 6a shows the performance comparison in terms of mean latency for all the models[2]. We observe that ServerlessLLM is 6X and 3.6X faster than PyTorch and Safetensors, respectively, for our smallest model (OPT-2.7B). We observe similar results with the largest model (LLaMA-2-70B) where ServerlessLLM is faster than PyTorch and Safetensors by 8.2X and 4.7X respectively. Safetensors is slower than ServerlessLLM due to a lot of page faults (112K for LLaMA-2-7B) on cold start. In contrast, ServerlessLLM's checkpoint manager leverages direct I/O and realizes chunk-based parallel loading, all contributing to the significant improvement in loading throughput. PyTorch is about 2X slower than Safetensors in our evaluation, consistent with the results in a public benchmark [37] reported by Safetensors. The primary reason is that PyTorch first copies data into host memory and then into GPU memory.

Furthermore, we observe that the loading performance of ServerlessLLM is agnostic to the type of the model. For example, the performance of both OPT-13B and LLaMA-2-13B is similar signifying the fact that the performance is only dependent on the checkpoint size.

[2]The number after the model name represents the number of parameters in the figure and B stands for Billion.

**Loading performance with LoRA adapters.** ServerlessLLM also supports loading LoRA adapters [34] in PEFT format [49]. We conducted experiments using the same setting in [65]. For an adapter (rank=32, size=1GB) of LLaMA-70B model, ServerlessLLM achieves 83.5ms loading latency which is 4.4X faster than Safetensors whose loading latency is 370ms. This demonstrates ServerlessLLM's loader design efficiency in small checkpoint loading.

**Harness full bandwidth of the storage devices.** We now move to understand if ServerlessLLM can utilize the entire bandwidth that a storage medium offers to achieve low latency. We use the same setup as described above. We choose LLaMA-2-7B to represent the SOTA LLM model. We use FIO [17] with the configuration of asynchronous 4M direct sequential read with the depth of 32 as the optimal baseline and optimized throughput using the result in all storage media. We test various settings of FIO to make sure the configuration chosen has the highest bandwidth on each storage media. For object storage over the network, we use the official MinIO benchmark to get the maximum throughput.

Figure 6b shows the bandwidth utilization across different storage devices, normalized relative to the measurements obtained using FIO and MinIO. The storage device from bottom to top is ascending in maximum bandwidth. We observe that ServerlessLLM's model manager is capable of harnessing different storage mediums and saturating their entire bandwidth to get maximum performance. Interestingly, we observe that ServerlessLLM is well suited for faster storage devices such as RAID0-NVMe compared to Pytorch and Safetensors. It shows that existing mechanisms are not adaptive to newer and faster storage technology. Despite the loading process passing through the entire memory hierarchy, ServerlessLLM is capable of saturating the bandwidth highlighting the effectiveness of pipelining the loading process.

**Performance breakdown.** We now move to highlight how each optimization within the model manager contributes towards the overall performance. We run an experiment using RAID0-NVMe with various OPT models. We start from the basic implementation (ReadByTensor) and incrementally add

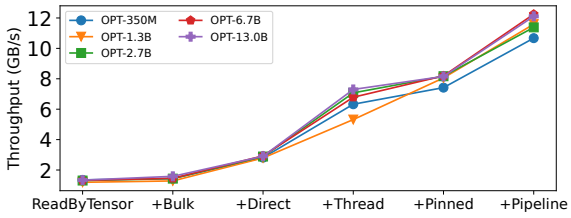



Figure 7: Performance breakdown of checkpoint loaders.

optimizations until the Pipeline implementation. Figure 7 shows the performance breakdown for each model. We observe similar contributions by different optimizations for all the models despite having different checkpoint size.

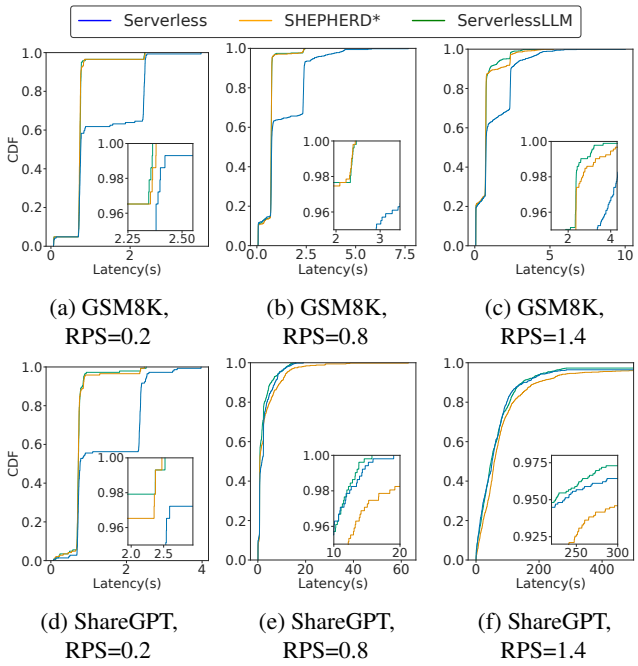
Bulk reading improves 1.2x throughput, mitigating the throughput degradation from reading small tensors one after another (on average one-third of the tensors in the model are less than 1MB). Direct IO improves 2.1x throughput, bypassing cache and data copy in the kernel. Multi-thread improves 2.3x throughput, as multiple channels within the SSD can be concurrently accessed. Pinned memory provides a further 1.4x throughput, bypassing the CPU with GPU DMA. Pipeline provides a final 1.5x improvement in throughput, helping to avoid synchronization for all data on each storage tier.

We run ServerlessLLM in a container to limit the CPU cores it can use. We find that with 4 CPU cores, ServerlessLLM can achieve maximum bandwidth utilization. We set a sufficiently large chunk size in bulk reading (16MB) to involve less number of reads and also pinned memory-based chunk pool does not need extra CPU cycles for data copy.

## 7.3 ServerlessLLM Model Scheduler

In this section, we evaluate the performance of the ServerlessLLM's cluster scheduler on test bed (ii). We compare ServerlessLLM against two schedulers – the de-facto serverless scheduler and Shepherd [90] scheduler. The serverless scheduler randomly chooses any GPU available and does not comprise any optimization for loading time. We implement Shepherd scheduler and use ServerlessLLM's loading time estimation strategy to identify the correct GPU. We call the modified scheduler as Shepherd*. Therefore, in principle, Shepherd* and ServerlessLLM will choose the same GPU. However, Shepherd* will continue to rely on preemption, while ServerlessLLM will rely on live migration to ensure lower latency times.

Figure 8a shows the result of a scenario where we run all three schedulers against OPT-6.7B model and GSM8K and ShareGPT dataset while increasing the requests per second. ShareGPT dataset's average inference time is 3.7X longer than GSM8K. Figure 8a and Figure 8d show the case where there is no locality contention for both datasets. The serverless scheduler cannot take advantage of locality-aware scheduling




(a) GSM8K, RPS=0.2 (b) GSM8K, RPS=0.8 (c) GSM8K, RPS=1.4

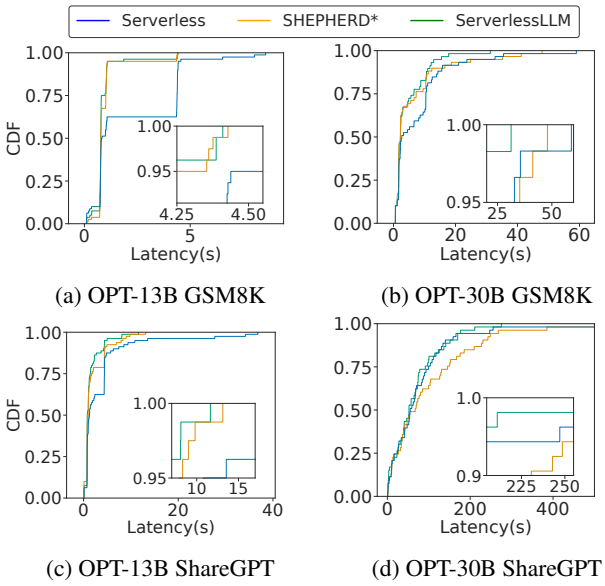
(d) ShareGPT, RPS=0.2 (e) ShareGPT, RPS=0.8 (f) ShareGPT, RPS=1.4

Figure 8: Impacts of RPS on model loading schedulers.

unlike ServerlessLLM and Shepherd* leading to longer latency. For 40% of the time, the model is loaded from SSD due to random allocation of the GPUs. As there is no migration or preemption, the performance of Shepherd and ServerlessLLM is similar.

When the schedulers are subjected to medium requests per second, for GSM8K (Figure 8b, without locality-aware scheduling, the loading times start causing queueing latency leading with Serverless scheduler resulting in increasing the P99 latency by 1.86X. As there is no migration or preemption, the performance of Shepherd and ServerlessLLM is similar. With a longer inference time with ShareGPT (Figure 8e, we even observe 2X higher P99 latency with Shepherd* compared to ServerlessLLM due to preemption. As ServerlessLLM relies on live migration in case of locality contention, ServerlessLLM performs better than the other schedulers despite the number of migrations is higher (114 out of 513 total requests) than the number of preemptions (40 out of 513 total requests).

On further stressing the system by increasing the requests per second to 1.4, for GSM8K, one can clearly observe the impact of live migration and preemption. ServerlessLLM outperforms Shepherd* and Serverless schedulers by 1.27X and 1.95X on P99 latency respectively. There are 9 preemptions and 53 migrations respectively for a total of 925 requests. As discussed in Section 5.1, preemptions lead to longer latency compared to migrations. We also observe that with Shepherd*, model checkpoints are read from SSD 2X times more than with ServerlessLLM. With ShareGPT (figure 8f, we observe that the GPU occupancy reaches 100% leading to requests




(a) OPT-13B GSM8K (b) OPT-30B GSM8K

(c) OPT-13B ShareGPT (d) OPT-30B ShareGPT

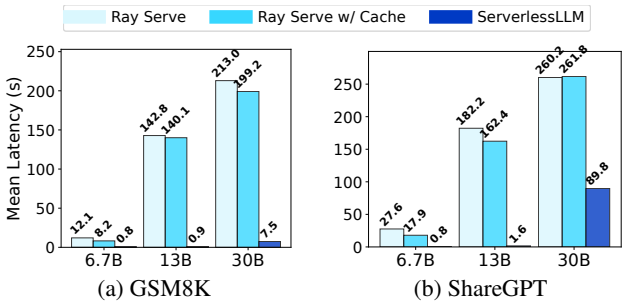
Figure 9: Impacts of datasets and models on model loading schedulers.

timeouts with all the three schedulers[3]. Shepherd behaves the worst compared to Serverless and ServerlessLLM schedulers, i.e., 1.43X and 1.5X higher P95 latency respectively. ServerlessLLM and Shepherd* issue 64 migrations and 166 preemptions, respectively for a total of 925 requests. In this scenario, ServerlessLLM's effectiveness is constrained by resource limitations.

We further stress the system by running even larger models (OPT-13B and OPT-30B) with GSM8K and ShareGPT datasets. Figure 9 shows the results for those experiments. locality-aware scheduling is more important for larger models as caching them in the main memory can reap better performance. As ServerlessLLM and Shepherd* are both locality-aware, they can make better decisions while scheduling the requests leading to better performance. As Serverless scheduler makes decisions randomly, for GSM8K, we observe that for 35-40% times, the model is loaded from SSD leading to poor performance. We see similar behavior for ShareGPT, OPT-13B experiment too. For the OPT-30B ShareGPT case, the model size is 66 GB. Hence, only two models can be stored in the main memory at any given time reducing the impact of locality-aware scheduling. Even in this extreme case, ServerlessLLM still achieves 35% and 45% lower P99 latency compared to Serverless and Shepherd* respectively.

**Time Estimation.** The GPU time estimation error is bounded at 5ms, while the SSD loading error is bounded at 40ms. However, we do observe instability in CUDA driver calls. For instance, when migrating a model, we noted that cleaning up GPU states (e.g., KV cache) using `torch.cuda.empty_cache()` can lead to inaccurate estimations, resulting in an average underestimation of 25.78 ms. While infrequent, we observed a maximum underestimation of 623 ms during GPU state cleanup in one out of 119 migrations (as depicted in Figure 8e).

[3]Based on the average inference time of OPT-6.7B on ShareGPT dataset, the maximum theoretically RPS is 1.79.




(a) GSM8K (b) ShareGPT

Figure 10: Impacts of datasets and models on overall serving systems.

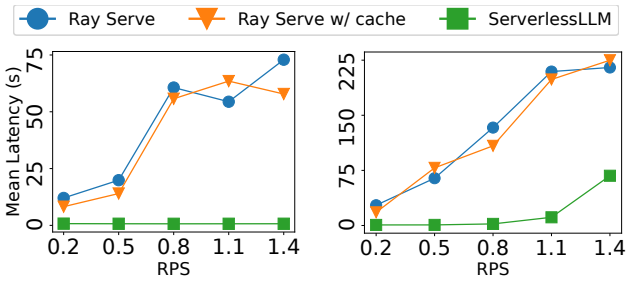
## 7.4 Entire ServerlessLLM in Action

We aimed to deploy the entire ServerlessLLM with a serverless workload on test bed (ii). Here, we compare ServerlessLLM against state-of-the-art distributed model serving systems: (i) Ray Serve (Version 2.7.0), a version we have extended to support serverless inference scenarios with performance that can match SOTA serverless solutions such as KServe; (ii) Ray Serve with Cache, a version we improved to adopt a local SSD cache on each server (utilizing the LRU policy as in ServerlessLLM) to avoid costly model downloads; and (iii) KServe (Version 0.10.2), the SOTA serverless inference system designed for Kubernetes clusters.

For best performance, Ray Serve and its cache variant are both enhanced by storing model checkpoints on local SSDs and estimating download latency by assuming an exclusively occupied 10 Gbps network. For each system, we set the maximum concurrency to one and set the keep-alive period equal to its loading latency, following prior work [60]. We launch parallel LLM inference clients to generate various workloads, where each request has a timeout threshold of 300 seconds.

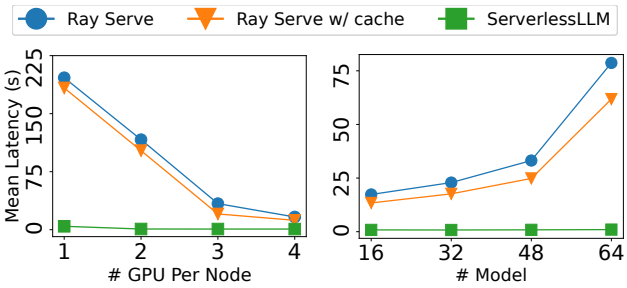
**Effectiveness of loading-optimized checkpoints.** We aimed to assess the effectiveness of loading-optimized checkpoints within a complete serverless workload, employing various model sizes and datasets to diversely test the checkpoint loaders.

In this experiment, as depicted in Figure 10, Ray Serve and Ray Serve with Cache utilize Safetensors. Owing to the large sizes of the models, the SSD cache cannot accommodate all models, necessitating some to be downloaded from the storage server. With OPT-6.7B and GSM 8K, ServerlessLLM starts models in an average of 0.8 seconds, whereas Ray Serve takes 12.1 seconds and Ray Serve with Cache 8.2 seconds, demonstrating an improvement of over 10X. Even




(a) GSM8k (b) ShareGPT

Figure 11: Impacts of RPS on overall serving systems.




(a) Impacts of # GPUs per node (b) Impacts of # models

Figure 12: System scalability and resource efficiency.

with a faster network (i.e., 100 Gbps), the average latency of Ray Serve could drop to 3.8 seconds, making it still 4.7 times slower than ServerlessLLM. The significance of the model loader becomes more pronounced with larger models, as ServerlessLLM can utilize parallel PCIe links when loading large models partitioned on multiple GPUs from pinned memory pool. For instance, with OPT-30B, ServerlessLLM still initiates the model in 7.5 seconds, while Ray Serve's time escalates to 213 seconds and Ray Serve with Cache to 199.2 seconds, marking a 28X improvement.

This considerable difference in latency substantially affects the user experience in LLM services. Our observations indicate that ServerlessLLM can fulfill 89% of requests within a 300-second timeout with OPT-30B, whereas Ray Serve with Cache manages only 26%.

With the ShareGPT dataset (Figure 10b), which incurs a 3.7X longer inference time than GSM 8K, the challenge for model loaders becomes even more intense. For models like 6.7B and 13B, ServerlessLLM achieves latencies of 0.8 and 1.6 seconds on average, respectively, compared to Ray Serve and Ray Serve with Cache, which soar to 182.2 and 162.4 seconds. When utilizing OPT-30B, ServerlessLLM begins to confront GPU limitations (with all GPUs occupied and migration unable to free up more resources), leading to an increased latency of 89.9 seconds. However, this is still a significant improvement over Ray Serve with Cache, which reaches a latency of 261.8 seconds

**Effectiveness of live migration and loading scheduler.** In evaluating the effectiveness of LLM live migration and the loading scheduler, we created workloads with varying RPS levels. Scenarios with higher RPS highlight the importance of achieving load balancing and locality-aware scheduling since simply speeding up model loading is insufficient to address the resource contention common at large RPS levels.

From Figure 11a, it is evident that ServerlessLLM, equipped with GSM8K, consistently maintains low latency, approximately 1 second, even as RPS increases. In contrast, both Ray Serve and Ray Serve with Cache experience rising latency once the RPS exceeds 0.5, which can be attributed to GPU resource shortages. Their inability to migrate LLM inference for locality release or to achieve load balancing, unlike ServerlessLLM, results in performance degradation.

With the more demanding ShareGPT workload, as shown in Figure 11b, ServerlessLLM maintains significant performance improvements — up to 212 times better — over Ray Serve and Ray Serve with Cache across RPS ranging from 0.2 to 1.1. However, at an RPS of 1.4, ServerlessLLM's latency begins to rise, indicating that despite live migration and optimized server scheduling, the limited GPU resources eventually impact ServerlessLLM's performance.

**Resource efficiency.** A major advantage of the low model startup latency in ServerlessLLM is its contribution to resource savings when serving LLMs. We vary the number of GPUs available on each server to represent different levels of resource provisioning. As shown in Figure 12a, ServerlessLLM scales well with elastic resources. With just one GPU per server, ServerlessLLM already achieves a 4-second latency by efficient migrations and swaps. In contrast, Ray Serve with Cache requires at least four GPUs per server to attain a 12-second latency, which is still higher than ServerlessLLM's performance with only one GPU per node. With larger clusters, the resource-saving efficiency of ServerlessLLM is expected to become even more pronounced, as larger clusters offer more options for live migration and server scheduling.

The resource efficiency of ServerlessLLM is further evident when maintaining a fixed number of GPUs while increasing the number of LLMs in the cluster. In Figure 12b, with a limited number of models, Ray Serve with Cache can match ServerlessLLM in latency performance. However, as the number of models grows, the performance gap widens, showcasing ServerlessLLM's potential suitability for large-scale serverless platforms.

**KServe comparison.** In our study, we assess KServe and ServerlessLLM within a Kubernetes cluster. Given that our four-server cluster is unsuitable for a Kubernetes deployment, we instead utilize an eight-GPU server, simulating four nodes with two GPUs each. Since KServe performs slower than the other baselines considered in our evaluation, we only briefly mention KServe's results without delving into details.

With KServe, the GPU nodes initially exhibited a first token latency of 128 seconds. This latency was primarily due to KServe taking 114 seconds to download an OPT-6.7B model checkpoint from the local S3 storage over a 1 Gbps network. However, after applying the same enhancement as those for Ray Serve, we reduced the first token latency to 28 seconds. Despite this improvement, KServe's best latency was significantly higher than those achieved by ServerlessLLM. Notably, ServerlessLLM was the only system able to reduce the latency to within one second.

## 8 Related Work

**Serverless inference systems.** Extensive research has focused on optimizing ML model serving in serverless architectures, targeting batching [10, 84, 89], scheduling [60, 87], and resource efficiency [24, 43]. Industry solutions like AWS SageMaker and Azure ML [50], along with the open-source KServe [16], demonstrate practical implementations. Despite these advancements, serverless inference systems still perform suboptimally with LLMs, as our paper demonstrates.

**Serverless cold-start optimizations.** Cold-start latency is a significant issue in serverless systems, addressed through various strategies including fast image pulling [79], lightweight isolation [45, 53], snapshot and restore [15, 22, 28, 67, 77], resource pre-provision [64], elastic resource provisioning [48, 78], and fork [9, 80]. These approaches mainly focus on reducing startup times for containers or VMs without loading large external states. Recent research has explored optimizing cold-starts by facilitating faster model swaps between GPUs and host memory [39, 88], though scalability with LLMs is still challenging. In contrast, ServerlessLLM effectively minimizes cold-start latency through LLM-specific innovations, such as optimized checkpoint formats and loading pipelines, live migration, and a cluster scheduler tailored to LLM inference characteristics.

**Exploiting locality in serverless systems.** Locality plays a crucial role in various optimization strategies for serverless systems. This includes leveraging host memory and local storage for data cache [41, 59, 69], optimizing the reading of shared logs [40], and enhancing communication efficiency in serverless Directed Acyclic Graphs (DAGs) [46, 47]. ServerlessLLM, distinct from existing methods, introduces a high-performance checkpoint cache for GPUs, markedly improving checkpoint loading from multi-tier local storage to GPU memory. Recent studies [8, 86] have also recognized the need for leveraging locality in orchestrating serverless functions. Beyond these studies, ServerlessLLM leverages LLM-specific characteristics in improving the locality-based server's selection and launching locality-driven inference.

**LLM serving systems.** Recent advancements in LLM serving have improved inference latency and throughput. Orca [85] uses continuous batching for better GPU utilization during inference. AlpaServe [44] shows that model parallelism can enhance throughput while meeting SLO constraints, though it has yet to be tested on generative models. vLLM [42] introduces PagedAttention for efficient KV cache management. SplitWise [55] improves throughput by distributing prompt and token generation phases across different machines. Some approaches [14, 66] also use storage devices to offload parameters from GPUs to manage large LLM sizes. However, these systems often overlook model loading challenges, leading to increased first token latencies when multiple models share GPUs. ServerlessLLM addresses this by focusing on minimizing loading latency to complement these throughput and latency optimizations.

## 9 Conclusion

This paper describes ServerlessLLM, a low-latency serverless inference system purposefully designed for LLMs. The design of ServerlessLLM uncovers significant opportunities for system research, including designing new loading-optimized checkpoints, discovering the need to support live migration when conducting locality-driven LLM inference, and enabling a serverless cluster scheduler to be aware of the locality of checkpoints in a cluster when optimizing its model scheduling decision. We believe our work can be extended to ensure fairness of resources across the cluster and explore the possibility of smart checkpoint placement. We look forward to addressing these issues in the future. We consider ServerlessLLM as the first step towards unlocking the potential of serverless computing for LLMs. We will continue to develop the open-source version of ServerlessLLM. Given its versatility, we envision it as a platform to test new research ideas.

## 10 Acknowledgments

We sincerely thank our shepherd, Amar Phanishayee, and the OSDI reviewers for their insightful feedback, which helped improve the quality of this paper. We also extend our thanks to Boris Grot for his feedback. We acknowledge Zhaonan Zhang, Mingyu Dai, and Yusen Fei for their work in the early stages of this project. We are grateful for the GPU resources provided by the School of Informatics and the Edinburgh International Data Facility. This work is supported by UK EPSRC and gifts from Tencent.

## References

[1] Amazon elasticache pricing. https://aws.amazon.com/elasticache/pricing/?nc=sn&loc=5. Accessed: 2024-05-31.

[2] Custom LLMs. https://deepinfra.com/docs/advanced/custom_llms, 2024. Accessed on 2024-06-02.

[3] Generative AI tailored to you. https://fireworks.ai/, 2024. Accessed on 2024-06-02.

[4] The leading enterprise AI platform. https://cohere.com/, 2024. Accessed on 2024-06-02.

[5] Run AI with an API. https://replicate.com/, 2024. Accessed on 2024-06-02.

[6] Serverless endpoints for leading open-source models. https://www.together.ai/products#inference, 2024. Accessed on 2024-06-02.

[7] Your data. your AI. your future. https://www.databricks.com/, 2024. Accessed on 2024-06-02.

[8] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 365–380, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[10] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[11] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023.

[12] Amazon. AWS S3. https://aws.amazon.com/s3/, 2023. Accessed on 2024-01-22.

[13] Amazon. Serverless inference - minimizing cold starts. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html, 2023. Accessed on 2024-01-22.

[14] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[15] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[16] The KServe Authors. Kserve. https://github.com/kserve/kserve, 2023. Accessed on 2024-01-22.

[17] Jens Axboe. Flexible I/O Tester, 2022. Accessed on 2024-01-22.

[18] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[19] Anyscale Technical Blog. https://www.anyscale.com/blog/loading-llama-2-70b-20x-faster-with-anyscale-endpoints, 2023. Accessed on 2024-01-22.

[20] Banana.dev Technical Blog. https://www.banana.dev/blog/turboboot, 2023. Accessed on 2024-01-22.

[21] Microsoft Official Blog. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-a

i-powered-microsoft-bing-and-edge-your-copilot-for-the-web/, 2023. Accessed on 2024-01-22.

[22] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Andrew A. Chien, Liuzixuan Lin, Hai Nguyen, Varsha Rao, Tristan Sharma, and Rajini Wijayawardana. Reducing the carbon impact of generative AI inference (today and in 2035). In *HotCarbon*, pages 11:1–11:7. ACM, 2023.

[24] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, Carlsbad, CA, July 2022. USENIX Association.

[25] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021.

[26] GitHub Copilot. https://github.com/features/copilot, 2023. Accessed on 2024-01-22.

[27] DataBricks. LLM Inference Performance Engineering: Best Practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023. Accessed on 2024-01-22.

[28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[29] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23(1), jan 2022.

[30] Cloud Native Computing Foundation. etcd. https://etcd.io, 2023. Accessed on 2024-01-22.

[31] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {rdma}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[32] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022.

[33] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI*, pages 443–462. USENIX Association, 2020.

[34] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[35] HuggingFace. https://huggingface.co, 2023. Accessed on 2024-01-22.

[36] HuggingFace. Safetensors: ML Safer for All. https://github.com/huggingface/safetensors, 2023. Accessed on 2024-01-22.

[37] HuggingFace. Speed Comparison. https://huggingface.co/docs/safetensors/speed, 2023. Accessed on 2024-01-22.

[38] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[39] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *EuroSys*, pages 249–265. ACM, 2023.

[40] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[42] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP*, pages 611–626. ACM, 2023.

[43] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[44] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, pages 663–679. USENIX Association, 2023.

[45] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *USENIX Annual Technical Conference*, pages 53–68. USENIX Association, 2022.

[46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[47] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[48] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. {KungFu}: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954, 2020.

[49] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[50] Microsoft. Azure ML. https://learn.microsoft.com/en-us/azure/machine-learning, 2023. Accessed on 2024-01-22.

[51] MinIO. MinIO. https://min.io, 2023. Accessed on 2024-01-22.

[52] NVIDIA. NVIDIA DGX H100. https://resources.nvidia.com/en-us-dgx-systems/ai-enterprise-dgx, 2023. Accessed on 2024-01-22.

[53] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[54] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/, 2022. Accessed on 2024-01-22.

[55] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting, 2023.

[56] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[57] PyTorch. PyTorch Documentation: Saving and Loading Models. https://pytorch.org/tutorials/beginner/saving_loading_models.html, 2023. Accessed on 2024-01-22.

[58] PyTorch. torch.load. https://pytorch.org/docs/stable/generated/torch.load.html, 2023. Accessed on 2024-01-22.

[59] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[61] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[62] ONNX Runtime. API Reference, onnxruntime.backend.prepare. https://onnxruntime.ai/docs/api/python/api_summary.html, 2023. Accessed on 2024-01-22.

[63] AWS SageMaker. Machine Learning Service - Amazon SageMaker. https://aws.amazon.com/pm/sagemaker/, 2023.

[64] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

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

[66] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[67] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[68] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads, 2022.

[69] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[70] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative LLM serving, 2024.

[71] Mistral AI Team. Better, faster, stronger. Mistral AI News, April 2024. Accessed: 2024-05-31.

[72] The Mosaic Research Team. Introducing DBRX: A new state-of-the-art open LLM. *Mosaic AI Research*, March 2024. Accessed: 2024-05-31.

[73] The Ray Team. Ray serve. https://docs.ray.io/en/latest/serve/index.html, 2023. Accessed on 2024-01-22.

[74] TensorFlow. TensorFlow Documentation: Using the Saved Model Format. https://www.tensorflow.org/guide/saved_model, 2023. Accessed on 2024-01-22.

[75] TensorFlow. tf.saved_model.load. https://www.tensorflow.org/api_docs/python/tf/saved_model/load, 2023. Accessed on 2024-01-22.

[76] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[77] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572,

New York, NY, USA, 2021. Association for Computing Machinery.

[78] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. Spotnik: Designing distributed machine learning for transient cloud resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[79] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[80] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.

[81] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models, 2023.

[82] xAI. Grok-1, 2024. Accessed: 2024-05-31.

[83] Ming Xu. https://huggingface.co/datasets/shibing624/sharegpt_gpt4, 2023.

[84] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 768–781, New York, NY, USA, 2022. Association for Computing Machinery.

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

[86] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[87] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148. IEEE, 2021.

[88] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *CoRR*, abs/2306.03622, 2023.

[89] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[90] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the Wild. In *NSDI*, pages 787–808. USENIX Association, 2023.

[91] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *CoRR*, abs/2205.01068, 2022.

[92] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.