PipeBoost: Resilient Pipelined Architecture for Fast Serverless LLM Scaling

Chongpeng Liu Xiaojian Liao* Hancheng Liu Limin Xiao Jianxin Li*

Beihang University

Abstract

This paper presents PipeBoost, a low-latency LLM serving system for multi-GPU (serverless) clusters, which can rapidly launch inference services in response to bursty requests without preemptively over-provisioning GPUs. Many LLM inference tasks rely on the same base model (e.g., LoRA). To leverage this, PipeBoost introduces fault-tolerant pipeline parallelism across both model loading and inference stages. This approach maximizes aggregate PCIe bandwidth and parallel computation across GPUs, enabling faster generation of the first token. PipeBoost also introduces recovery techniques that enable uninterrupted inference services by utilizing the shared advantages of multiple GPUs. Experimental results show that, compared to state-of-the-art low-latency LLM serving systems, PipeBoost reduces inference latency by 31% to 49.8%. For certain models (e.g., OPT-1.3B), PipeBoost achieves cold-start latencies in the range of a few hundred microseconds.

1 Introduction

Large Language Models (LLMs) have recently become popular and crucial for applications such as chatbots (e.g., ChatGPT) [6], search engines [8], code assistants (e.g., Copilot) [3], and AI-driven decision making [52, 59], where low latency and high reliability are critical for serving performance. LLMs require significant GPU memory space due to their massive number of parameters, posing a substantial challenge to large-scale LLM serving and contributing to its high cost [5]. For example, an OPT-66B model [61] requires 132GB A100 GPUs.

Deploying LLMs on serverless GPU clusters holds immense potential, enabling service providers to efficiently multiplex LLMs across GPUs [24]. Users submit inference requests and pay per use, without needing to manage the complexities of LLM deployment. This approach not only increases GPU resource utilization but also significantly lowers the barrier for users to access and leverage LLM capabilities.

*Corresponding author: {liaoxj, lijx}@buaa.edu.cn

However, serverless workloads are often unpredictable and face frequent load spikes [34, 45], which introduce challenges to serving LLMs at scale. Provisioning sufficient GPUs for potential load spikes is expensive, especially given the substantial memory demands of LLMs; GPUs reserved for peak traffic periods remain underutilized during off-peak times [20]. Starting LLM services on-demand retains the auto-scaling benefits of serverless computing but faces high latency caused by cold starts (i.e., loading models from CPU memory or SSDs and initiating inference from scratch). Our study (§3) shows that even state-of-the-art systems like ServerlessLLM [19], which store model checkpoints in local CPU memory, require around 90% of the time for model loading during a cold start, delaying the return of the first token to users.

The conventional wisdom is that a GPU can not start LLM inference until it fully loads model parameters. In this paper, we demonstrate that this is not the case for serverless LLM due to the base model sharing property in LLM serving, which can be prevalent in practice and is overlooked by existing systems. For instance, LLM service providers like OpenAI often deploy their proprietary LLMs (e.g., GPT-4 [40]) at scale to power AI-driven applications such as ChatGPT. Additionally, training an LLM from scratch is prohibitively expensive and requires extensive engineering expertise [44], making it inaccessible to many developers [13, 16, 28]. Instead, these developers commonly fine-tune existing base models for specific domains. LoRA (Low-Rank Adaptation) [22] is a widely adopted parameter-efficient fine-tuning technique. It keeps the base model intact and introduces smaller matrices to encode the delta adjustments, enabling efficient customization without modifying the original base model.

We introduce PipeBoost, a serverless LLM serving system that achieves low-latency cold starts (§4). Leveraging the base mode sharing property, PipeBoost breaks the sequential execution order of LLM loading and inference stages, allowing them to be performed in a pipelined fashion. The key idea lies in grouping LLMs with the same base model onto the same GPU server and introducing fault-tolerant pipeline parallelism across both the loading and inference stages. Specifically, it

includes the following three key designs:

(1) Pipeline-Parallel Model Loading (§4.2). PipeBoost reorders the model layers loaded by each GPU during the model loading stage, allowing GPUs to load distinct parts of the same base model and LoRA adapters. This design eliminates redundant model layer transfers along the critical path, efficiently utilizing aggregated PCIe and memory bandwidth, and enabling the system to reach a ready-to-infer state quickly.

(2) Pipeline-Parallel Model Inference (§4.3). When model layers across GPUs form a complete model, PipeBoost initiates the inference, leveraging pipeline parallelism, merged LoRA, and epoch-based adapter switching techniques to handle bursty requests while concurrently loading the remaining layers. This design exploits pipeline parallelism to rapidly activate multi-GPU parallel processing capabilities, effectively managing sudden traffic surges. PipeBoost also features a seamless strategy-switching mechanism, allowing inference processes to transition to alternative parallel strategies. This flexibility is crucial as different GPU servers may be optimized for specific use cases (e.g., long-sequence inference). Moreover, pipeline parallelism, while advantageous in launching LLM services, can perform worse compared to single-GPU inference (where each GPU independently performs inference) under high request arrival rates due to the significant communication overhead between GPUs.

(3) Pipeline-Parallel Recovery (§4.4). Pipeline parallelism is more fragile to failures. Crash in a single LLM instance can impact all instances and requests within the pipeline. PipeBoost introduces model layer reassignment and KV cache reconstruction techniques to swiftly restore the pipeline-parallel model loading and inference process. This design mitigates the drawbacks of pipeline parallelism, improving system reliability, reducing the recovery time and ensuring predictable cold-start latencies.

We implement PipeBoost based on PyTorch [41] and Transformers [55], supporting a wide range of LLMs, including OPT, Falcon, and Mistral. Our experimental evaluation (§5) compares PipeBoost against existing systems such as Transformers and ServerlessLLM. The results demonstrate that PipeBoost reduces the time to first token by 57%–84% and 30%–47% compared to Transformers and ServerlessLLM, respectively. Notably, PipeBoost can initialize OPT-1.3B and generate the first token within 1 second, with the model loading time costing only 0.64 seconds. Recovery experiments further show that PipeBoost reduces recovery time by 51% compared to traditional methods that restart LLM instances.

In summary, we make the following contributions.

- We analyze the limitations of existing serverless LLM serving systems and highlight the overlooked characteristic of base model sharing.
- We propose PipeBoost, introducing the Fault-Tolerant Pipeline Parallelism to maximize the parallel computing and data transfer capabilities of multi-GPU servers.
- We conduct extensive experiments to demonstrate that PipeBoost achieves faster startup and recovery latencies compared to state-of-the-art systems.

2 Background

This section first introduces the background of LLM serving, followed by an explanation of the cold-start challenges when deploying LLMs in serverless environments.

2.1 LLM Serving

LLM serving primarily consists of two parts: model loading and model inference. In the model loading stage, in addition to initializing the runtime (e.g., Python) and other prerequisites, it involves two steps. First, a model checkpoint is loaded from a persistent storage device (such as local NVMe SSDs or remote storage like S3) into CPU memory and deserialized into an executable format. Next, if inference is performed on the GPU, the model checkpoint or parameters must be transferred from CPU memory to GPU memory. Once the model is loaded, the LLM serving can perform inference.

LLM inference consists of two stages: prefill and decode. Taking a transformer-based model as an example, the model comprises multiple layers, where the output of one layer serves as the input to the next.

During the prefill stage, the inference engine inputs initial tokens into the first layer of the model. Each layer performs two key operations. First, it computes the relationships among input tokens using a self-attention mechanism. Then, it applies a feed-forward network (FFN) to produce the output for the current layer. This output is passed to the subsequent layer, where similar computations are performed. The final layer of the model generates the first token to be returned to the user, ending the prefill stage and starting the decode stage.

In the decode stage, the inference engine generates tokens iteratively, processing only the most recent token at each step until reaching either an end-of-sequence token or the maximum sequence length. KV cache technology eliminates the need to reprocess the full input prompt or prior tokens by storing intermediate key (K) and value (V) matrices in GPU memory. Each model layer maintains its own KV cache, enabling efficient self-attention computations and reducing latency.

Low-Rank Adaptation (LoRA) [22] introduces low-rank trainable matrices, referred to as adapters, into the model's architecture, specifically in the weight matrices of layers like attention and FFNs in transformer models. It reduces the number of parameters that need to be updated during fine-tuning while still enabling the model to adapt effectively to new tasks. In LoRA, the original weight matrix W remains fixed during training, and only the low-rank matrices A and B (with rank r and dimensions $d \times r$ and $r \times d$, respectively) are updated. Since the rank r is substantially smaller than the dimension of W (i.e., d), the memory footprint of the LoRA adapter is significantly reduced compared to the original model.

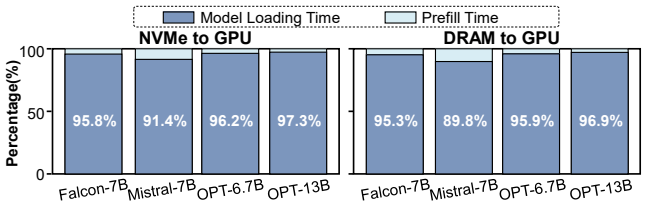

Figure 1: **Overhead of LLM model loading.**

2.2 Serverless LLM

Serverless computing has gained popularity in cloud computing due to its exceptional elasticity. Many companies, including Amazon, Azure, and Hugging Face, have implemented serverless LLM services in their cloud platforms [1, 2, 4]. Users only need to submit functions related to LLM inference, without the need to rent containers or virtual machines and deploy their own inference systems as in traditional methods, greatly reducing the development burden on users.

Serverless computing requires efficient scaling capabilities. In cloud workloads (e.g., Azure Functions [2]), load spikes are common and often unpredictable. To handle these spikes, mainstream serverless platforms typically use two approaches: over-provisioning and on-demand cold starts.

In the over-provisioning approach, the serverless platform reserves GPUs and pre-launches related services in advance. However, this method leads to significant GPU resource wastage. First, because the peak of burst traffic is generally unpredictable, it is difficult to determine the necessary number of reserved GPUs. Second, the reserved GPUs result in wasted GPU compute power and memory, and this waste is exacerbated in LLM inference scenarios, as LLM parameters can occupy up to 800 GB of memory (e.g., Llama 3.1 405B [7]), often requiring multiple GPUs for complete loading.

In the on-demand cold start approach, when existing GPUs cannot handle the current traffic load (i.e., when a load spike occurs), the serverless scheduler allocates additional GPUs to the LLM inference tasks. At this point, the LLM inference service must go through the model loading stage before it can begin inference, a process known as LLM cold start.

LLM cold starts can be prevalent in real-world settings. As demonstrated in the ServerlessLLM [19], if the distribution of LLM inference load follows the Azure trace [46], over 40% of functions experience a cold start rate exceeding 25%, and approximately 25% of functions face a cold start rate greater than 60%, within a 5-minute keep-alive interval.

3 Motivation

The severe performance overhead of serverless cold starts has been a known challenge in big data tasks [19]. This overhead is even more pronounced in LLM inference tasks, as LLM serving systems require loading a large number of parameters. If these parameters are stored on a slower remote storage system (e.g., S3), loading delays can exceed 60 seconds, which is intolerable in interactive services requiring low latency.

To mitigate the impact of LLM cold starts, ServerlessLLM [19] stores model parameters in local CPU memory and high-performance NVMe SSDs, leveraging the high-speed PCIe bus to significantly reduce LLM loading time. Although powerful, the model loading time in ServerlessLLM is significantly higher than the latency of the prefill stage, which could severely impact user experience when handling burst traffic.

3.1 Overhead of LLM Cold Starts

This section quantifies the LLM cold start overhead in ServerlessLLM for generating the first token. Similar to mainstream LLM inference frameworks such as Transformer [55] and vLLM [29], ServerlessLLM requires deserialization and loading parameters into GPU memory before inference can begin (i.e., entering the prefill stage). Experiments were conducted on a 2-GPU 40GB A100 machine with 512GB of DDR4 memory and a 2TB NVMe SSD. Detailed experimental environments are provided in §5.2. The experiment tests mainstream open-source LLMs like Falcon, Mistral and OPT. The models were placed either in CPU memory or on NVMe SSD, and measurements were taken for the latency ratio of model loading to the prefill stage for generating the first token.

Figure 1 reveals that even with local high-performance NVMe storage, the model loading time still accounts for an average of 95.2% of the overall time. Even when the model resides in CPU memory, the time taken to load the model from the CPU to the GPU still constitutes 94.6% of the total. The experimental results demonstrate that once an LLM cold start occurs, the latency for generating the first token will increase by at least 9.8×, severely impacting user experience.

3.2 Observation and Analysis

Base model sharing across multiple GPU inference instances, a unique property in serverless LLM inference, is common in private cloud and LLM serving platforms. For example, OpenAI's ChatGPT service uses its own GPT-4o or GPT-4o mini model [6], while Moonshot AI's LLM inference service utilizes its proprietary Kimi model [43]. Even in deployments with multiple models (such as a mix of LLaMa and OPT), inference clusters often contain numerous GPU instances running inference with the same model [30].

Another observation is that even when models are customized based on different professional domains or user preferences, these custom models typically share a common base model. Training a model from scratch is prohibitively expensive [22], so techniques like LoRA (details in §2.1) are often used to fine-tune an existing base model by adjusting parameters to suit specialized applications. LoRA adapters are notably compact, taking up only a small fraction of base model

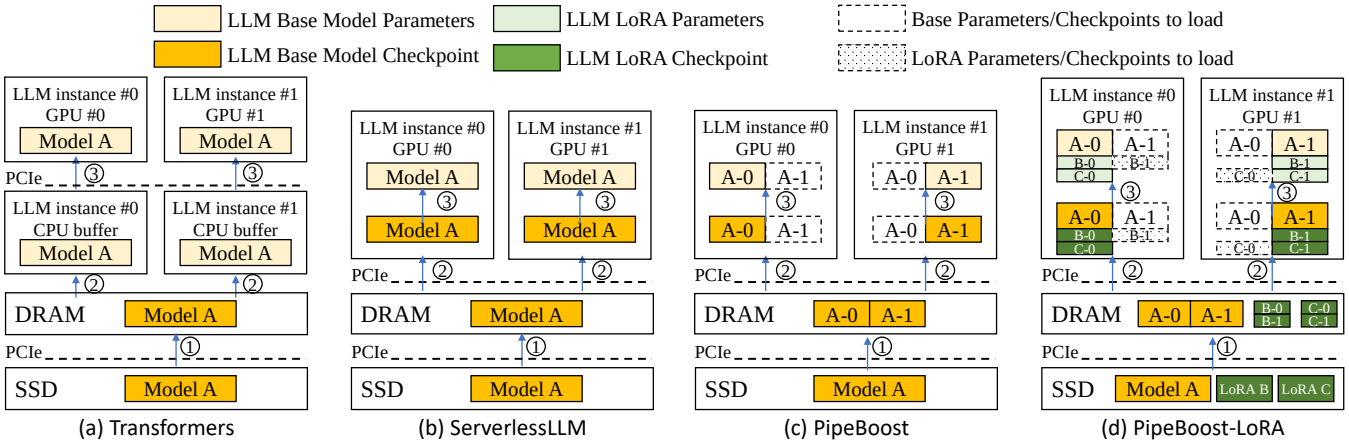

Figure 2: **Different methods of LLM model loading on multiple GPUs.** Transformers and ServerlessLLM require the entire set of LLM parameters to be fully loaded into GPU memory before inference can begin. In contrast, PipeBoost allows different GPUs to load distinct portions of the LLM layers, enabling inference to start without waiting for the full model to be loaded.

parameters—for example, just 0.01% in GPT-3 175B [22].

In summary, on an inference server equipped with multiple GPUs, it is common for multiple LLM instances to share the same full model or base model. Current serverless LLM inference systems do not account for this, which can lead to inefficient PCIe bandwidth utilization during the auto-scaling of LLM instances. We illustrate this issue by taking state-of-the-art serverless LLM inference systems including Transformers and ServerlessLLM as examples (Figure 2). For simplicity, we assume a server with only two GPUs, with the process extending similarly for more GPUs.

Transformers first load model A from the SSD into CPU DRAM. Next, the model checkpoint is converted into an executable inference format (i.e., LLM parameters). It is important to note that an LLM instance is launched on each GPU, resulting in two copies of the LLM parameters residing on the CPU side. Subsequently, the LLM parameters are separately transferred to each GPU for inference execution.

The LLM startup process in ServerlessLLM is similar to that of Transformers, with the only difference being that the LLM checkpoint is first transferred to the GPU, where the GPU handles the conversion from checkpoint to parameters.

A typical GPU server often has high CPU memory bandwidth, with multiple memory channels providing aggregate bandwidth exceeding hundreds of GB/s. Generally, each GPU device is connected to the CPU through a dedicated PCIe slot (e.g., PCIe 4.0 x16) by up to 32 GB/s, allowing for high-speed transfer of model checkpoints or parameters between CPU memory and GPU HBM. However, the model loading methods of Transformer and ServerlessLLM do not efficiently leverage the high bandwidth of CPU memory and PCIe links. In a given time window, all GPUs may be reading the same portion of the model checkpoint or parameters (e.g., all 8 GPUs reading the first 1 GB portion of the model), which

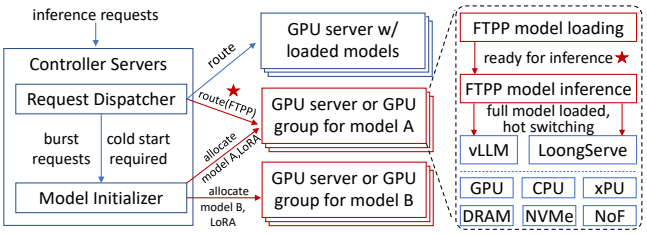

Figure 3: **PipeBoost overview.** Red rectangles and arrows are new designs introduced by PipeBoost. Model checkpoints or parameters are stored on CPU DRAM, local NVMe SSDs or remote storage, e.g., NVMe over Fabrics (NoF), and are loaded into GPU during LLM inference cold starts.

results in the redundant transmission of large amounts of duplicate data along the critical I/O path that requires low latency. Even worse, GPUs cannot begin inference until the entire model's parameters are fully loaded, and the GPU's computation capability is thus left underutilized.

4 PipeBoost

To address the high model loading latency discussed in §3, we propose PipeBoost, a low-latency LLM serving system for serverless computing.

4.1 Overview

Similar to traditional GPU serverless systems, PipeBoost consists of Controller Servers and GPU Servers (Figure 3). The Controller Servers handle requests and forward them, via the Request Dispatcher, to servers where the LLM inference service is initialized. In cases of sudden traffic spikes, the Model

Initializer launches LLM inference instances on additional GPU servers to handle subsequent requests.

Inspired by the observation that multiple GPUs can share a base model in the cluster (§3.2), PipeBoost categorizes models and ensures that LLM inference instances using the same class of models are launched on the same GPU server. Notably, the model parameters do not need to be identical; models based on the same base model but using different LoRA adapters are also scheduled to launch on the same GPU server.

Within each GPU server, PipeBoost introduces the Fault-Tolerant Pipeline Parallelism (FTPP) technique to accelerate the startup process of inference services. Specifically:

- **Pipeline-Parallel Model Loading** (§4.2): PipeBoost partitions the model checkpoints by layers (i.e., inter-operator partition) and loads it across GPUs, enabling the GPU server to reach a request-inference state as quickly as possible, achieving low-latency cold starts.
- **Pipeline-Parallel Model Inference** (§4.3): PipeBoost leverages the parallelism of multiple GPUs to handle burst requests while asynchronously loading the remaining layers of the model onto each GPU. This process continues until each GPU holds a complete copy of the model parameters. Once the full model is loaded, PipeBoost can either maintain the pipeline-parallel strategy or perform a seamless hot-switching to an existing inference strategy optimized for specific use cases.
- **Fault Tolerance** (§4.4): PipeBoost introduces rapid recovery techniques to minimize the performance impact of partial LLM instance failures on pipeline inference. By dynamically adjusting the number of model layers loaded on each GPU and reconstructing KV caches on the fly, PipeBoost maximizes the advantages of pipeline parallelism in handling bursty requests.

While FTPP is primarily designed for GPU inference, it is also adaptable to CPU and other heterogeneous computing devices compatible with PyTorch. For clarity, this paper focuses on describing FTPP in the context of GPU inference.

4.2 Pipeline-Parallel Model Loading

FTPP optimizes the transfer of model checkpoints by reordering the portions sent to different GPUs, effectively eliminating redundant data transfers from CPU memory to GPU HBM in the critical path.

4.2.1 Base Model Loading

Inspired by the observation that multiple GPUs share a base model, FTPP prioritizes the transfer of non-duplicated model checkpoints during the inference service startup to maximize the effective utilization of memory and PCIe bandwidth. As shown in Figure 2(c), PipeBoost first reads the model checkpoint from SSD into DRAM or uses a model checkpoint already residing in DRAM. Next, PipeBoost splits the model checkpoint in DRAM into N parts, with each part corresponding to a GPU, where N represents the number of GPUs. PipeBoost then transfers each part of the checkpoint in parallel to the GPUs. Crucially, during this step, PipeBoost utilizes the available PCIe bandwidth for transferring non-duplicated checkpoint parts, e.g., GPU 0 reads model A-0 while GPU 1 reads A-1. Finally, each GPU independently converts its checkpoint part into parameters. At this point, the inference service is successfully initialized and ready to serve. While inference is being executed on GPUs (which will be detailed in §4.3), the remaining portions of the model are progressively loaded into GPU HBM. For instance, model A-1 is loaded onto GPU 0, and model A-0 is loaded onto GPU 1.

4.2.2 LoRA Adapter Loading

PipeBoost also supports efficient loading of LoRA models. Suppose GPU 0 and GPU 1 need to load two models that share the same base model A but use different LoRA adapters B and C. The loading process for the base model follows the previously described approach (as shown in Figure 2(c)). The loading of LoRA adapters is similar to that of the base model, where each adapter is partitioned, and the segmented parts are distributed across GPUs. Notably, to handle requests requiring different LoRA adapters, each GPU also loads an additional portion of the other adapter. For example, GPU 0 loads C-0, while GPU 1 loads B-1. Since LoRA adapters are relatively lightweight, typically consuming only a fraction of the size of the base model (e.g., 0.01%), the time and memory overhead for loading these adapters is negligible. Once both the base model and LoRA adapters are ready, PipeBoost begins serving inference requests while asynchronously loading the remaining parts of the model into GPUs. After all model parameters have been fully loaded, the additional LoRA adapter segments can be discarded to free up GPU memory. For instance, GPU 0 discards C-0, and GPU 1 discards B-1.

4.3 Pipeline-Parallel Model Inference

When the model layers already loaded across GPUs can be combined into a complete model, PipeBoost immediately utilizes pipeline parallelism to begin inference.

4.3.1 Base Model Inference

As shown in Figure 4(b), all inference requests begin the prefill computation phase on GPU 0 which contains the first half portion of layers, while the remaining layers of the base model and LoRA adapters are concurrently loaded onto each GPU. Once GPU 0 completes computations, PipeBoost transfers the intermediate results to GPU 1 via NVLink or PCIe, where the remaining prefill computations are finalized. At the end of the prefill phase, the first token is generated and returned to both users and GPU 0. The GPU 0 then begins

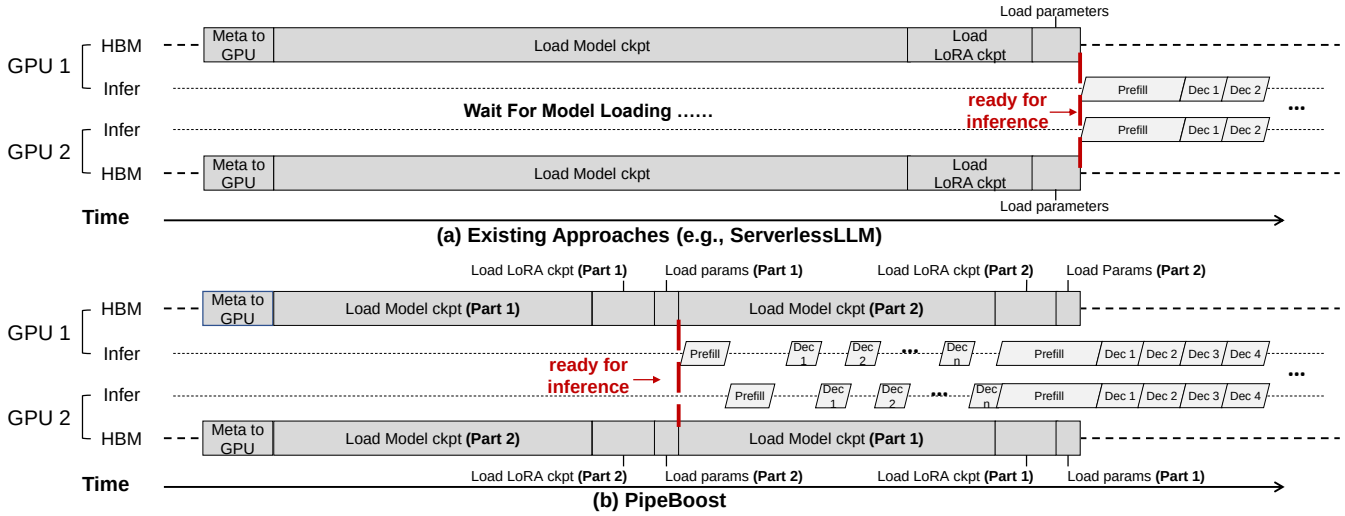

Figure 4: **Different methods of LLM model inference during LLM cold starts on multiple GPUs.**

the decode computation phase. Each decode step employs a similar pipeline parallelism approach as the prefill phase, continuing until a termination token is encountered.

4.3.2 Merged LoRA Adapter Inference

PipeBoost employs the merged LoRA approach for pipeline inference, where the parameters of the LoRA adapter are merged back into the base model prior to inference, forming a full model. Inference is then conducted on the full model, following the same process outlined in the previous paragraph. We chose the merged LoRA approach for the following reasons. First, compared to unmerged LoRA, merged LoRA eliminates additional computational overhead (which accounts for approximately 38% of the original computation [57]), enabling faster generation of the first token during request surges. Second, PipeBoost groups and batches requests with different LoRA adapters, significantly reducing the overhead caused by switching between LoRA adapters. Finally, PipeBoost is designed to eventually converge to a strategy where each GPU performs independent inference using a single LoRA adapter (reasons in §4.3.3). The merged LoRA approach allows for a smoother transition to this strategy.

In PipeBoost, requests associated with different LoRA adapters flow through all GPUs, potentially leading to frequent adapter switching. To address this issue, PipeBoost employs an epoch-based adapter switching technique (Figure 5). First, PipeBoost classifies requests and groups those belonging to the same adapter into the same queue. Second, leveraging the merged LoRA approach, PipeBoost prioritizes the scheduling of batches corresponding to the currently activated adapter. At regular intervals, PipeBoost switches adapters to serve requests from other batches. Notably, due to the use of pipeline parallelism, adapter switching across GPUs does not occur simultaneously but sequentially. Each GPU switches

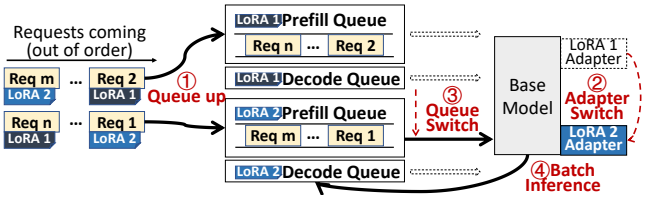

Figure 5: **Details of epoch-based adapter switching.**

adapters only after completing the batch received from the preceding GPU in the pipeline.

4.3.3 Seamless Strategy Switching

Once all GPUs in a server have fully loaded the complete model, PipeBoost can seamlessly switch to other inference strategies, such as vLLM's single-GPU independent inference. The switching process is straightforward: batches of requests submitted after the switching point are executed using the new inference parallelism strategy.

We propose transitioning PipeBoost from pipeline parallelism to other strategies for the following reasons. First, while pipeline parallelism excels in quickly initializing inference services to handle bursty requests, its advantages diminish once all GPUs have fully loaded the complete model.

Figure 6 shows that , independent of the total request arrival rates, single-GPU inference, where requests are evenly distributed across GPUs and each GPU independently executes inference, consistently achieves significantly lower latency compared to pipeline-parallel model inference. Furthermore, the gap between the two widens as the total rates increase. This phenomenon occurs because pipeline parallelism introduces communication overhead for transferring hidden states between GPUs. As the request density increases, this over-

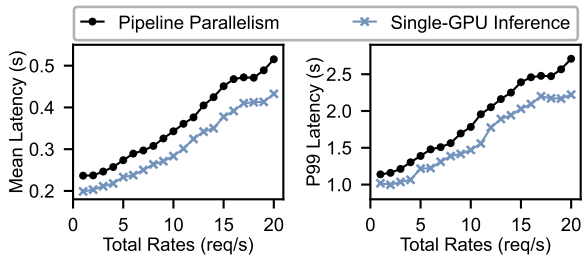

Figure 6: **Necessity of Seamless Strategy Switching.** Single-GPU inference, where each GPU independently performs inference, outperforms pipeline-parallel model inference.

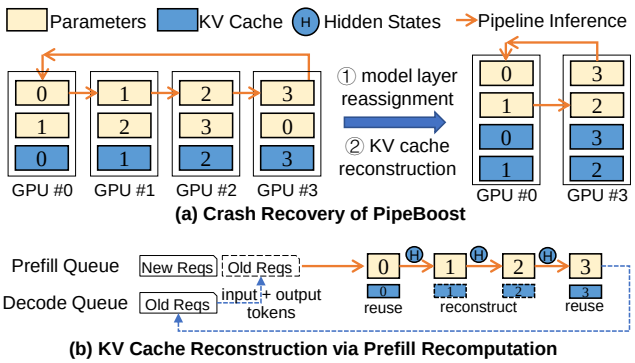

Figure 7: **PipeBoost Recovery.**

head scales accordingly, leading to a more pronounced latency difference. Hence, PipeBoost opts to switch the inference strategy to single-GPU inference.

Second, different workloads have distinct characteristics, which may require tailored optimization strategies. For example, in handling long-sequence input, sequence parallelism offers superior performance compared to other inference parallelism strategies, as shown in [56]. In such scenarios, repurposing the GPU server for long-sequence tasks may be a more suitable approach.

4.4 Fault Tolerance

This section first explains the necessity of providing efficient crash recovery support for pipeline-parallel model loading and inference, and then details the specific recovery process.

4.4.1 Need For Fault Tolerance

LLM instance crashes are common in large-scale GPU clusters, often caused by GPU hardware failures [35], driver errors, or bugs in the CPU-side initialization program [21, 23].

A typical approach to address such crashes is for the Controller Servers to select another available GPU and reload the model [33, 49, 53]. However, this method is inefficient and results in unpredictable model loading and inference latency.

Another consideration is isolation. In existing methods like Transformers and ServerlessLLM, each GPU independently loads and serves the model. A single GPU failure does not impact requests to other GPUs. In contrast, PipeBoost requires GPUs to collaborate during the model loading and the initial stage of inference. If a single GPU fails and the issue is not handled, it could affect all requests directed to the GPU server.

4.4.2 Pipeline-Parallel Recovery

PipeBoost introduces rapid recovery techniques to address the issues in §4.4.1. The recovery process consists of two steps. The first step is model layer reassignment, which quickly reconstructs an efficient inference pipeline. The second step is KV cache reconstruction, enabling fast continuation of the inference process by reusing existing results. We assume that all GPUs on a GPU server are initialized, as PipeBoost is designed to utilize the parallel bandwidth and computation capability of multiple GPUs. Crashes may occur during either the model loading stage (§4.2) or the model inference stage (§4.3). This section discusses both scenarios separately.

Recovery for model loading. When an LLM instance crash is detected during model loading, PipeBoost dynamically adjusts the number of model layers loaded by the remaining operational instances. The adjustment follows two principles:

- **Load Balance**: the workload among GPUs should be evenly distributed. This paper focuses on scenarios where homogeneous GPUs are deployed on a single GPU server, meaning that assigning the same number of model layers to each GPU ensures balanced load. For heterogeneous GPUs, a more sophisticated model partitioning strategy is required.
- **Layer Contiguity**: GPUs are assigned adjacent model layers whenever possible. PipeBoost leverages pipeline parallelism during inference (§4.3), so placing consecutive model layers on the same GPU minimizes the need for frequent inter-GPU synchronization.

Assume there are four GPUs, and the model is partitioned into 4 segments (Figure 7a). GPU 0 sequentially loads model segments 0, 1, 2 and 3, while GPU 3 loads 3, 0, 1 and 2. GPUs 1 and 2 follow a similar pattern. Under normal circumstances, GPU 0 and GPU 3 only need to complete loading segments 0 and 3, respectively, to begin inference. Now, suppose GPU 1 and GPU 2 crash during model loading. In this case, GPU 0's loading sequence remains unchanged, while PipeBoost adjusts GPU 3's loading order to 2, 3, 0 and 1. Once GPU 0 finishes loading 0 and 1, and GPU 3 completes 2 and 3, PipeBoost can proceed with inference execution.

Recovery for model inference. When an LLM instance crash is detected during model inference, PipeBoost first scans the GPUs to assess the distribution of loaded model layers and identifies a viable chain for pipeline parallelism to continue inference. If no such chain is found, PipeBoost initiates the aforementioned model loading recovery process.

After model loading recovery, PipeBoost reconstructs the KV cache of the missing model layers. Specifically, PipeBoost distributes requests into two queues according to their stages (Figure 7b). It inserts requests from the decode queue into the prefill queue to rebuild the KV cache. The input and output tokens processed so far are merged into a single input sequence and fed into the first model layer. For layers with an existing KV cache, PipeBoost recomputes the Q matrix and reuses the KV cache to generate hidden states for the next layer. For layers without a KV cache, PipeBoost performs a full prefill operation and stores the KV cache in GPU HBM. Once reconstruction is complete, the request is placed back into the decode queue.

5 Evaluation

PipeBoost is built upon Transformers [55] and PyTorch [41]. We adapt Transformers to enable PipeBoost to support LLMs such as OPT [61], Falcon [11], and Mistral [27], requiring only approximately 300 lines of code (LOC) modification for each model. This modular approach allows easy integration of additional models. Built on top of PyTorch, PipeBoost implements FTPP and incorporates continuous batching techniques [60] to improve efficiency. Instead of using Ray [36], PipeBoost employs threads (via Python's `concurrent.futures`) to coordinate LLM instances within a GPU server, avoiding the significant initialization overhead associated with Ray.

We evaluate PipeBoost's performance against state-of-the-art serverless LLM serving systems, focusing on low-latency startup. We first introduce the evaluation setup (§5.1), then analyze the startup process of base LLMs (§5.2) and LoRA-based LLMs (§5.3), followed by assessing scalability across varying GPU counts, batch sizes, prompt lengths and improvements from epoch-based adapter switching (§5.4), ending with evaluating fault-tolerance mechanisms (§5.5).

5.1 Evaluation Setup

Device Setup. We conduct our experiments on two servers, each equipped with 512GB DDR4 memory, an Intel Xeon Platinum 8338C CPU, and a Samsung 980 PRO 2TB NVMe SSD. One server is equipped with four NVIDIA RTX 4090 GPUs (24 GB memory), while the other contains two NVIDIA A100 GPUs (40 GB memory). Unless explicitly noted, all evaluations are performed on A100 GPUs.

Model and Datasets. We utilize advanced and widely-studied LLMs in system research including Mistral, OPT and Falcon, with parameter sizes of 1.3B, 2.7B, 6.7B, 13B, and 30B. We use the GSM8K datasets [15], including real user questions across varying lengths and domains, with dynamic response lengths, making it representative of practical workloads.

Baseline Systems. We evaluate PipeBoost by comparing it to two baselines: the widely used Transformers framework and an optimized implementation of ServerlessLLM [19]. For a fair comparison, the model loading and inference components of ServerlessLLM are included, while Ray [36], a cross-server scheduling component, is excluded. This exclusion is justified as Ray's initialization takes 4-6 seconds, which is time-intensive and beyond the scope of our study.

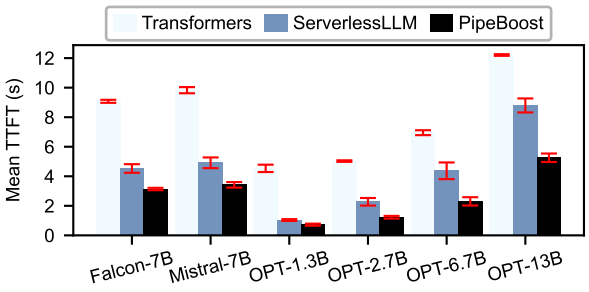

Figure 8: **Mean TTFT across various base LLMs.**

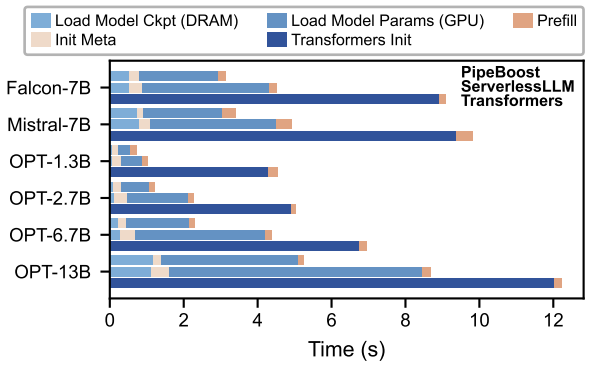

Figure 9: **Mean TTFT breakdown.** Load Model Ckpt: loading model checkpoints from SSD to DRAM, Init Meta: initializing model metadata, Load Model Params: transferring checkpoints to GPUs and loading into model parameters, Prefill: performing prefill operations. Notably, Transformers follows its own loading procedure (Transformers Init).

Metrics. To assess the startup latency for LLM serving, we measure ***Time to First Token (TTFT)***, defined as the time elapsed from receiving an inference request to generating the first output token. Notably, TTFT is also a key metric in existing LLM serving systems that mitigate cold starts by preloading LLM parameters into GPU memory. In our experiments, TTFT encompasses the time required for model loading. Additionally, we use ***request completion latency (referred to as latency)*** to assess the total time required to complete a request. We measure ***recovery time***, defined as the duration from the moment of a crash to the point where the system resumes service, to evaluate the system's fault-tolerance capabilities in recovering from failures.

5.2 Startup Performance of Base LLM

To evaluate PipeBoost's startup performance, we measure TTFT across various model architectures and sizes. The test

launches one model on each GPU, simulating user input with a prompt size of 64 and a batch size of 64. The evaluation begins by loading the model checkpoint from an NVMe SSD and ends when the first token is output. As the test involves cold starts, our TTFT results may be higher compared to works focused on warm starts or pre-loaded models.

Starup Time. We first compare the overall startup performance of PipeBoost with the baselines, as shown in Figure 8. The evaluation includes Falcon-7B, Mistral-7B, and the OPT series (OPT-1.3B, OPT-2.7B, OPT-6.7B, and OPT-13B). PipeBoost consistently outperforms both Transformers and ServerlessLLM, achieving TTFT reductions of 30% to 85% across all tested models. For example, the TTFT for Mistral-7B is reduced by 65% compared to Transformers and 30% compared to ServerlessLLM. These results demonstrate the efficiency of PipeBoost's pipeline-parallel model loading.

Furthermore, PipeBoost shows significant advantages in handling larger models. For example, the TTFT for OPT-1.3B is reduced by 30.2% compared to ServerlessLLM, while for OPT-13B, the reduction reaches 40.3%. These reductions are attributed to PipeBoost's ability to incrementally load model layers across multiple GPUs, minimizing idle GPU time and overlapping loading with inference.

Startup Time Breakdown. To analyze the sources of startup latency, we decompose the startup process into key stages, following the order in which they occur during model startup. Figure 9 illustrates the contributions of each stage to the overall startup time for various model types, comparing PipeBoost with Transformers and ServerlessLLM. The results show that checkpoint loading (ranging from 9.05% for OPT-2.7B to 22.1% for OPT-13B) and parameter loading (ranging from 45.6% for OPT-1.3B to 70.7% for OPT-13B) are the most time-consuming stages during startup. For instance, in the case of OPT-13B, these two stages take 7.9 seconds with ServerlessLLM, compared to 4.8 seconds with PipeBoost, accounting for 92.8% and 91.9% of the overall startup latency.

PipeBoost significantly shortens these two stages, leading to a faster overall startup. For instance, with the Mistral-7B model, it reduces the time for these stages from 4.1 seconds in ServerlessLLM to 2.8 seconds, contributing to a total startup latency reduction from 4.9 seconds to 3.4 seconds.

5.3 Startup Performance of LoRA LLM

LoRA, a parameter-efficient fine-tuning technique, is widely adopted for LLMs. PipeBoost is designed to efficiently support LoRA-enhanced models. In this section, we evaluate PipeBoost's startup latency for serving LoRA-equipped LLMs. The experimental setup mirrors §5.2. The key difference is that each model starts with one LoRA adapter. This consistent setup ensures comparability while specifically focusing on the performance of LoRA-enabled scenarios.

Startup Time. To assess the impact of LoRA on startup performance, we measure the TTFT for LoRA-based LLMs such as Mistral-7B, Falcon-7B, OPT-6.7B, and OPT-13B. As shown in Figure 10, PipeBoost maintains its advantage over both Transformers and ServerlessLLM in LoRA-based LLM. The reasons for PipeBoost's superior performance are similar to those observed in the base model evaluation (§5.2), where pipeline-parallel model loading and inference play a crucial role in reducing latency.

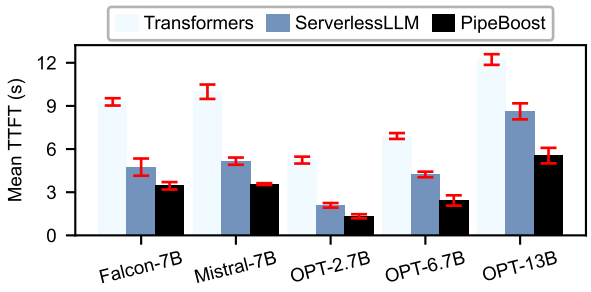

Figure 10: **Mean TTFT across various LoRA LLMs.**

Table 1: **Startup Time Breakdown with LoRA**

Stage	Mistral-7B Time(s) / Rate(%)	OPT-13B Time(s) / Rate(%)
PipeBoost		
Load Model Ckpts (DRAM)	1.105 / 30.75	1.157 / 20.86
Load Lora Ckpt (DRAM)	0.006 / 0.17	0.010 / 0.18
Init Meta	0.158 / 4.40	0.229 / 4.13
Load Model Params (GPU)	1.970 / 54.81	3.989 / 71.91
Load Lora Params (GPU)	0.025 / 0.70	0.021 / 0.38
Prefill	0.330 / 9.18	0.141 / 2.54
Total	**3.594 / 100.0**	**5.547 / 100.0**
ServerlessLLM		
Load Model Ckpts (DRAM)	1.091 / 20.92	1.167 / 13.54
Load Lora Ckpt (DRAM)	0.005 / 0.10	0.006 / 0.07
Init Meta	0.267 / 5.12	0.425 / 4.93
Load Model Params (GPU)	3.468 / 66.49	6.815 / 79.08
Load Lora Params (GPU)	0.053 / 1.02	0.041 / 0.48
Prefill	0.332 / 6.37	0.164 / 1.90
Total	**5.216 / 100.0**	**8.618 / 100.0**

A comparison of the performance between base LLMs (Figure 8) and LoRA-enhanced LLMs (Figure 10) reveals that PipeBoost introduces minimal latency overhead in supporting LoRA. For example, with Mistral-7B, PipeBoost exhibits a TTFT increase of only 0.18 seconds, approximately a 5% increase compared to the non-LoRA configuration. These results highlight PipeBoost's ability to seamlessly integrate LoRA while preserving its pipeline-parallel efficiency.

Startup Time Breakdown. Table 1 provide a breakdown of the startup process for LoRA LLMs, illustrating how LoRA impacts startup time. When LoRA is enabled, the checkpoint loading stage involves additional steps to load the LoRA adapters into main memory, which are subsequently transferred to the GPU. Results show that additional steps from LoRA add negligible overhead to the total startup latency. Specifically, for Mistral-7B, a single LoRA adapter introduces

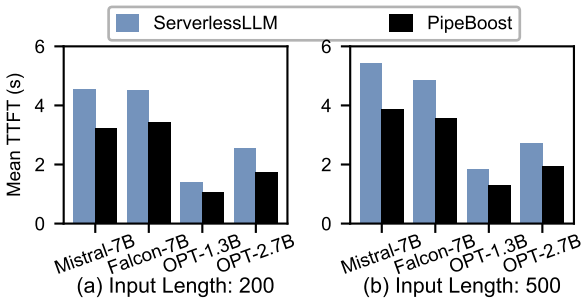

Figure 11: **Mean TTFT with varying input lengths.**

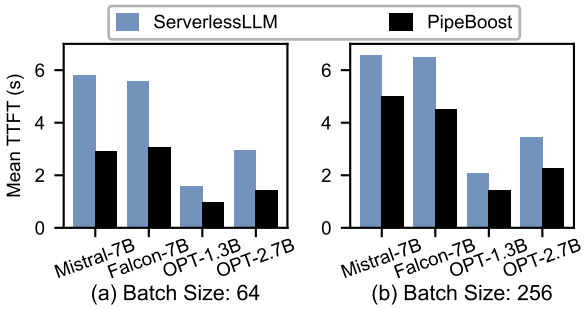

Figure 12: **Mean TTFT with varying batch sizes.**

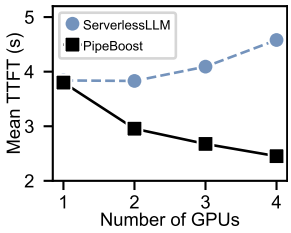

Figure 13: **Mean TTFT with varying GPU counts.** Lower is better.

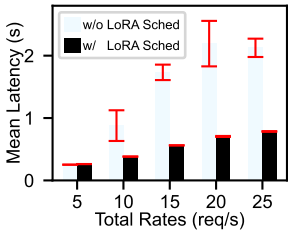

Figure 14: **Effect of epoch-based adapter switching technique.**

only a 0.87% overhead. For OPT-13B, the overhead is just 0.56%. PipeBoost's pipeline-parallel model loading technique further reduces the loading time of LoRA adapters by approximately N times, where N represents the number of GPUs concurrently involved in the loading process.

The results here also demonstrate PipeBoost's ability to efficiently infer with multiple LoRA adapters. Since the primary overhead remains in the base model, PipeBoost can load dozens of (and even more) LoRA adapters into GPU memory with minimal cost. Combined with the epoch-based adapter switching technique (which will be evaluated in §5.4), PipeBoost is capable of efficiently supporting inference with multiple LoRA adapters based on the same base model.

5.4 Scalability Analysis

This section analyzes the scalability of PipeBoost under varying workload conditions, including input length, batch size, the number of GPUs, and the impact of epoch-based adapter switching for LoRA LLMs. This analysis demonstrates PipeBoost's ability to handle dynamic workloads efficiently and scale effectively across hardware resources.

Scalability in Input Lenght and Batch Size. PipeBoost's inference performance is further evaluated under varying input lengths and batch sizes to assess its adaptability to diverse workload scenarios. As shown in Figure 11 and 12, PipeBoost consistently outperforms ServerlessLLM in terms of TTFT across all input length and batch size configurations.

Figure 11 illustrates the effect of input lengths (200 and 500 tokens) on TTFT. PipeBoost achieves significantly lower TTFT compared to ServerlessLLM across all tested models. For instance, with an input length of 200 tokens, PipeBoost reduces TTFT for Mistral-7B from 4.5 seconds to 3.2 seconds, a 28.9% reduction compared to ServerlessLLM. Similarly, for an input length of 500 tokens, PipeBoost reduces TTFT for Mistral-7B from 5.4 seconds to 3.9 seconds, a 27.8% reduction compared to ServerlessLLM.

Figure 12 illustrates the influence of batch sizes (64 and 256) on TTFT. PipeBoost effectively handles larger batch sizes with minimal degradation in TTFT, showcasing its scalability for high-throughput workloads. For a batch size of 64, PipeBoost reduces TTFT for Falcon-7B from 5.6 seconds to 3.1 seconds, a 44.6% reduction compared to ServerlessLLM. Even with a batch size of 256, PipeBoost maintains its advantage, reducing TTFT for Falcon-7B from 6.5 seconds to 4.5 seconds, a 30.8% reduction.

However, with the increase of input length and batch size, the performance gap between PipeBoost and ServerlessLLM narrows slightly, possibly due to the increased inter-GPU communication overhead of pipeline parallelism.

Scalability with Multiple GPUs. To assess the scalability of PipeBoost with increasing GPU counts, we further measure the TTFT for Mistral-7B, building on the experimental settings outlined in §5.2. The experiment is conducted on a server equipped with 4 NVIDIA RTX 4090 GPUs, with each GPU hosting one model instance.

As shown in Figure 13, ServerlessLLM experiences a latency increase of up to 19.3% as more GPUs are added. In contrast, PipeBoost achieves consistent TTFT reductions, ranging from 22.8% to 46.5%, compared to ServerlessLLM as the number of GPUs increases. Notably, with 4 GPUs, PipeBoost reduces TTFT by 35.5% compared to the single-GPU configuration. The results demonstrate PipeBoost's ability to scale, with the potential for further reductions in startup latency as GPU resources continue to grow.

Effect of epoch-based LoRA adapter switching technique. To evaluate the effectiveness of PipeBoost's epoch-based adapter switching mechanism, we conducted experiments with two LoRA-equipped LLMs on the GSM8K dataset. The experiment gradually increases request rates. Dynamic work-

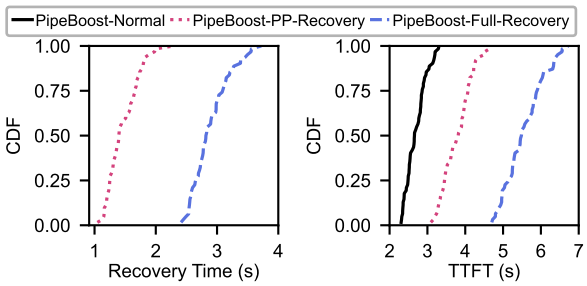

Figure 15: **Recovery time and TTFT of crash recovery for the model loading stage.**

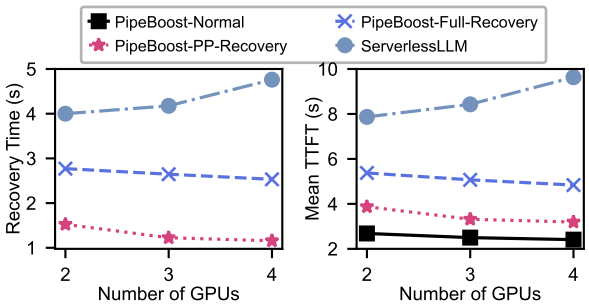

Figure 16: **Impact of GPU Count on PipeBoost's Crash Recovery.**

loads were simulated by switching LoRA adapters for user prompts with a switching probability of 20%.

We measure the completion latency to include the evaluation of decode stages. The epoch-based adapter switching technique performs batching and scheduling on requests for different LoRA adapters (details in §4.3.2). The baseline system serves requests as they arrive without LoRA scheduling, performing adapter switching eagerly.

As shown in Figure 14, PipeBoost (w/ LoRA Sched) significantly outperforms the baseline (w/o LoRA Sched). Without scheduling, mean latency increases from 0.3 seconds to 2.1 seconds as the request rate rises, with variance reaching 0.4 seconds at 20 RPS (requests per second). In comparison, epoch-based switching reduces mean latency by 63.1% at 25 RPS and maintains much lower variance, peaking at nearly 0.8 microseconds at 5 RPS.

5.5 Fault Tolerance and Robustness

In this section, we evaluate the fault tolerance of PipeBoost during both the model loading and inference stages. Fault tolerance is crucial for maintaining system reliability in large-scale deployments where GPU failures are common.

We perform evaluations with Mistral-7B under three configurations: (a) LLM startup with no crashes (PipeBoost-Normal), (b) crash recovery with Pipeline-Parallel Recovery (PipeBoost-PP-Recovery), and (c) full recovery involving a complete restart of the pipeline-parallel loading and inference processes (PipeBoost-Full-Recovery).

Recovery for model loading. PipeBoost's recovery performance for the model loading stage is evaluated on a server with 4 NVIDIA RTX 4090 GPUs. To simulate realistic failure scenarios, errors were introduced in 2 GPUs during the loading process to trigger the recovery mechanism. Figure 15 presents the recovery time and TTFT distributions.

PipeBoost-PP-Recovery demonstrates significant advantages over PipeBoost-Full-Recovery. For recovery time, PipeBoost-PP-Recovery achieves a median of 1.39 seconds, compared to 2.81 seconds for PipeBoost-Full-Recovery, yielding a 50.5% latency reduction. In terms of TTFT, PipeBoost-PP-Recovery reduces the median time to 3.8 seconds, compared to 5.5 seconds for PipeBoost-Full-Recovery, closely approaching the PipeBoost-Normal baseline of 2.66 seconds.

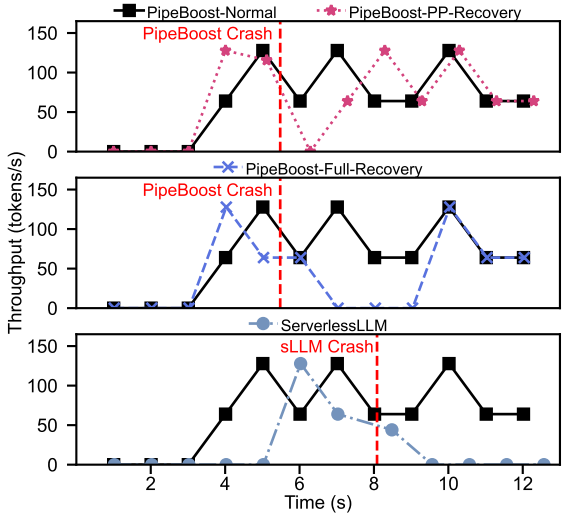

Figure 17: **System throughput during crash recovery for the model inference stage.**

PipeBoost minimizes disruption and ensures service continuity, by quickly reconstructing pipeline inference chains and redistributing workloads across operational GPUs.

Impact of GPU counts on PipeBoost's crash recovery. We have previously demonstrated that startup performance improves with an increasing number of GPUs (Figure 13). Here, we further investigate whether recovery can similarly gain benefits. Figure 16 shows the TTFT results for scenarios involving 2, 3, and 4 GPUs with a single GPU failure.

The results indicate that as the number of GPUs increases, the system is able to recover more quickly due to the additional resources available for redistributing workloads. For example, with 2 GPUs, the TTFT for recovery is 3.9 seconds, which improves to 3.3 seconds with 3 GPUs and further to 3.2 seconds with 4 GPUs, representing a 17.9% reduction from 2 to 4 GPUs. However, the improvement slows between 3 and 4 GPUs, possibly due to increased communication and synchronization overhead between GPUs.

Recovery for model inference. In addition to evaluating recovery during model loading, it is essential to assess the impact of GPU crashes during inference. Once PipeBoost successfully enters the pipeline-parallel inference stage, we simulate GPU failures by inducing errors during the inference chain and observe the behavior of the Pipeline-Parallel Recovery mechanism. Figure 17 shows the throughput (in tokens per second) over time for PipeBoost.

When a GPU crash occurs during inference (at approximately 6 seconds), the throughput of PipeBoost-PP-Recovery reduces from 128 tokens/s to 64 tokens/s but recovers to a normal state within 2 seconds. However, PipeBoost-Full-Recovery experiences a complete halt in throughput for nearly 4 seconds before resuming at a normal rate.

We observe that ServerlessLLM is slower than PipeBoost not only during startup but also during recovery. ServerlessLLM requires approximately 5 seconds to start, during which its throughput remains near zero. In contrast, PipeBoost's throughput begins increasing at the 3-second mark. Similarly, ServerlessLLM takes nearly 5 seconds to recover from failures, which is 2.5× longer than PipeBoost's recovery time.

We also observe that under both normal and crash recovery scenarios, PipeBoost's throughput fluctuates. This fluctuation is primarily due to the differing computational complexities of prefill and decode operations, which is a normal phenomenon and does not affect the argument for PipeBoost's low-latency startup. For instance, at throughput low points, there are more prefill requests, while at high points, there are more decode requests. Since prefill is computationally slower than decode, it reduces overall throughput.

6 Related Works

LLM serving systems. Recent advancements in LLM serving [26,30,42,60] primarily focus on optimizing inference performance, overlooking the model loading stage. However, several of these techniques can complement PipeBoost's pipeline-parallel model inference design. For example, Orca [60] introduces a continuous batching technique to enhance GPU utilization, which PipeBoost adopts. Splitwise [42] and DistServe [62] separate prefill and decode computations across different GPUs, a method that can be applied in PipeBoost when both GPU groups for prefill and decode computations complete loading an available pipeline inference chain.

Many works [9, 30, 48, 56, 57] propose inference strategies that PipeBoost can adopt once all GPUs complete loading the full model. vLLM [29] introduces paged attention to optimize GPU memory management. AlpaServe [30] employs model parallelism to expedite serving large DNN models across GPU clusters. Llumnix [48] dynamically reschedules inference requests across LLM instances, enabling systems to adapt to unpredictable workload dynamics. Techniques like chunked-prefill and stall-free batching in Sarathi-Serve [9] strike a balance between throughput and latency. dLoRA [57] orchestrates requests and LoRA adapters across replicas to achieve effective load balancing. LoongServe [56] introduces elastic sequence parallelism, offering adaptability to variable-length requests in different processing phases.

Optimizations for serverless cold-starts. The cold-start issue in serverless computing has been present since the early days of CPU-based computation. In GPU computing, this problem becomes even more severe due to the increased complexity of GPU application initialization [58], further exacerbated by the rapid growth in LLM parameters [19]. Numerous optimizations are proposed, including rapid image retrieval [51], streamlined isolation mechanisms [32, 39], snapshot-based restoration [12, 14, 17, 47, 50], dynamic pre-provisioning of resources [46], and process fork [10, 54]. While these methods effectively minimize startup times for containers or virtual machines, they do not address the high loading latency of LLM parameters.

ServerlessLLM utilizes high-performance storage within GPU servers to access LLMs. PipeBoost further enhances this process through FTPP techniques. To achieve the fastest cold start, ServerlessLLM estimates and compares model startup times and service migration durations across GPUs, selecting the most suitable server. In contrast, PipeBoost simplifies this process by initializing the same model on the same GPU server, avoiding the overhead of migration and estimation. With advancements like GPUDirect Storage technology and NVMe arrays, the time required for GPUs to load LLM parameters from DRAM and NVMe SSDs is expected to converge. Under such conditions, ServerlessLLM's selection strategy and live migration approach may become less effective, while PipeBoost stands to gain performance improvements.

Pipeline parallelism. Pipeline parallelism is commonly used in model training [18, 25, 31, 37, 38], but we apply the technique to the cold-start process of serverless LLMs, demonstrating its capability to rapidly initialize inference services. Additionally, we show how to provide fault tolerance for pipeline parallelism, thereby accelerating the recovery process of LLM serving systems. Moreover, we highlight the need to switch from pipeline parallelism to alternative parallel strategies when the request density on GPU servers exceeds a certain threshold to avoid excessive communication overhead.

7 Conclusion

We present PipeBoost, a low-latency inference engine for serverless LLM deployments, designed to achieve fast startup with robust fault-tolerant performance. PipeBoost significantly reduces cold-start times compared to state-of-the-art systems, supports LoRA-equipped LLMs with efficient inference enabled by its epoch-based adapter switching mechanism, and ensures minimal service disruption during GPU failures with its fault-tolerant recovery design. Extensive evaluations demonstrate PipeBoost's efficiency, scalability, and fault tolerance in multi-GPU environments.

References

[1] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[2] Azure Functions – Serverless Functions in Computing. https://azure.microsoft.com/en-us/products/functions.

[3] GitHub Copilot. https://github.com/features/copilot.

[4] Hugging Face Generative AI Services (HUGS). https://huggingface.co/docs/hugs/index.

[5] Information is Beautiful: the rise of generative-ai like chatgpt. https://informationisbeautiful.net/visualizations/.

[6] Introducing ChatGPT. https://openai.com/index/chatgpt/.

[7] Introducing Llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/.

[8] Introducing the new Bing. https://www.microsoft.com/en-us/edge/features/the-new-bing.

[9] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} Tradeoff in {LLM} Inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[10] 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 *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 923–935, USA, July 2018. USENIX Association.

[11] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The Falcon Series of Open Language Models, November 2023.

[12] 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, pages 730–746, New York, NY, USA, March 2022. Association for Computing Machinery.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, pages 1877–1901, Red Hook, NY, USA, December 2020. Curran Associates Inc.

[14] 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, pages 1–15, New York, NY, USA, April 2020. Association for Computing Machinery.

[15] 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, November 2021.

[16] DeepSeek-AI. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model, May 2024.

[17] 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*, ASPLOS '20, pages 467–481, New York, NY, USA, March 2020. Association for Computing Machinery.

[18] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 431–445, New York, NY, USA, February 2021. Association for Computing Machinery.

[19] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}: {Low-Latency} Serverless Inference

for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[20] Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, Keli Gui, Jie Tong, and Mao Yang. An Empirical Study on Low GPU Utilization of Deep Learning Jobs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–13, New York, NY, USA, April 2024. Association for Computing Machinery.

[21] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, et al. The Llama 3 Herd of Models, November 2024.

[22] 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, October 2021.

[23] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, 2024.

[24] Tao Huang, Pengfei Chen, Kyoka Gong, Jocky Hawk, Zachary Bright, Wenxin Xie, Kecheng Huang, and Zhi Ji. ENOVA: Autoscaling towards Cost-effective and Stable Serverless LLM Serving, May 2024.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 10, pages 103–112. Curran Associates Inc., Red Hook, NY, USA, December 2019.

[26] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 249–265, New York, NY, USA, May 2023. Association for Computing Machinery.

[27] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B, October 2023.

[28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, and Xin Jin. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs.

[29] 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 *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 611–626, New York, NY, USA, October 2023. Association for Computing Machinery.

[30] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[31] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models, September 2021.

[32] 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 *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, 2022.

[33] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal Checkpointing: Efficient and Flexible Checkpointing for Large Scale Distributed Training. *CoRR*, January 2024.

[34] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[35] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent, {Fine-Grained} {DNN} Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[36] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577, USA, October 2018. USENIX Association.

[37] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 1–15, New York, NY, USA, October 2019. Association for Computing Machinery.

[38] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning*, pages 7937–7947. PMLR, July 2021.

[39] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 57–69, USA, July 2018. USENIX Association.

[40] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, et al. GPT-4 Technical Report, March 2024.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 721, pages 8026–8037. Curran Associates Inc., Red Hook, NY, USA, December 2019.

[42] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132, June 2024.

[43] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving, July 2024.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, pages 1–16, Atlanta, Georgia, November 2020. IEEE Press.

[45] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 231–246, New York, NY, USA, October 2023. Association for Computing Machinery.

[46] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, pages 205–218, USA, July 2020. USENIX Association.

[47] Simon Shillaker and Peter Pietzuch. FAASM: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, pages 419–433, USA, July 2020. USENIX Association.

[48] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic

Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, 2024.

[49] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[50] 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, pages 559–572, New York, NY, USA, April 2021. Association for Computing Machinery.

[51] 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, 2021.

[52] Shenzhi Wang, Chang Liu, Zilong Zheng, Siyuan Qi, Shuo Chen, Qisen Yang, Andrew Zhao, Chaofei Wang, Shiji Song, and Gao Huang. Boosting llm agents with recursive contemplation for effective deception handling. In *The 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

[53] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 364–381, New York, NY, USA, October 2023. Association for Computing Machinery.

[54] 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, 2023.

[55] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[56] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 640–654, New York, NY, USA, November 2024. Association for Computing Machinery.

[57] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. {dLoRA}: Dynamically Orchestrating Requests and Adapters for {LoRA} {LLM} Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, 2024.

[58] Yanning Yang, Dong Du, Haitao Song, and Yubin Xia. On-demand and Parallel Checkpoint/Restore for GPU Applications. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, pages 415–433, New York, NY, USA, November 2024. Association for Computing Machinery.

[59] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *Advances in Neural Information Processing Systems*, 36:11809–11822, December 2023.

[60] 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, 2022.

[61] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona 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, June 2022.

[62] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.