USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FastServe: Iteration-Level Preemptive Scheduling for Large Language Model Inference

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin,
School of Computer Science, Peking University

https://www.usenix.org/conference/nsdi26/presentation/wu-bingyang

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

FastServe: Iteration-Level Preemptive Scheduling for Large Language Model Inference

Bingyang Wu* Yinmin Zhong* Zili Zhang* Shengyu Liu
Fangyue Liu Yuanhang Sun Gang Huang Xuanzhe Liu Xin Jin

School of Computer Science, Peking University

Abstract

Large language models (LLMs) power a new generation of interactive AI applications exemplified by ChatGPT. The interactive nature of these applications demands low latency for LLM inference. Existing LLM serving systems use run-to-completion processing for inference jobs, which suffers from head-of-line blocking and long latency.

We present FastServe, a distributed LLM serving system which exploits the autoregressive pattern of LLM inference to enable preemption at the granularity of each output token. FastServe uses preemptive scheduling to minimize latency with a novel skip-join Multi-Level Feedback Queue scheduler. Based on the new *semi* information-agnostic setting of LLM inference, the scheduler leverages the input length information to assign an appropriate initial queue for each arrival job to join. Queues with higher priority than the one the job joins are skipped to reduce demotions. We design an efficient GPU memory management mechanism that proactively offloads and uploads intermediate state between GPU memory and host memory for LLM inference. Evaluation shows that compared to the state-of-the-art solution vLLM, FastServe improves the throughput by up to 6.1×.

1 Introduction

Advancements in large language models (LLMs) open new possibilities in a wide variety of areas. For example, ChatGPT [1] enables users to interact with an AI agent in a conversational way to solve tasks in different topics. Many organizations follow the trend to release interactive LLM applications, such as the Microsoft Copilot [3], Google Gemini [21], Anthropic Claude [8], Alibaba Qwen [2], etc.

Inference serving is critical to interactive LLM applications. In order to provide engaging user experience, the interactive nature of these applications demands low latency for LLM inference. Users expect their inputs can be responded instantly. To fulfill the stringent requirement, enterprises provision expensive clusters with accelerators like GPUs.

Different from other deep neural network (DNN) model inference like ResNet [26], LLM inference has its own unique characteristics (§2). Given the model and hardware, traditional DNN inference jobs are typically deterministic and highly-predictable [24]. In contrast, due to the special *autoregressive* pattern, LLM inference jobs are highly variable. An LLM inference job generates output tokens iteratively based on input tokens and previous output tokens until a special <EOS> token is generated. As a result, the execution time depends on both the input length and the output length, the latter of which is not known *a priori*.

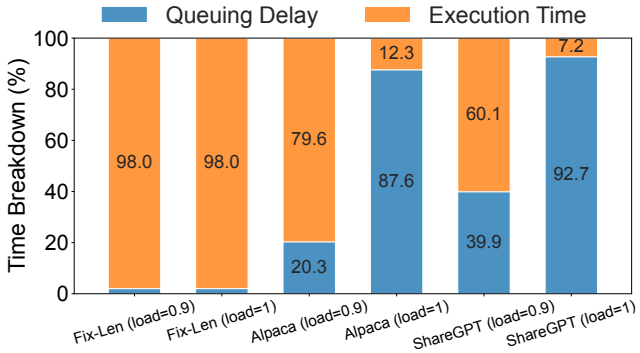


Figure 1: Head-of-line blocking in LLM inference.

To accommodate the unique characteristics of LLM inference, Orca [59] introduces iteration-level scheduling that dynamically adds new jobs or removes completed ones at the end of each iteration. vLLM [31] further introduces PagedAttention to reduce the memory fragmentation of LLM inference jobs. However, they both use first-come-first-served (FCFS) to process inference jobs. Once a job is scheduled, it runs until it finishes. Due to the limited GPU memory and the strict latency requirement, the current processing batch cannot be expanded with an arbitrary number of incoming jobs, thus a long job may block the incoming ones, known as head-of-line blocking [29]. The problem is particularly acute for LLM inference jobs. A large LLM inference job, i.e., with long input and output length, would run for a long time to block incoming short jobs.

Figure 1 demonstrates how workload characteristics critically impact LLM serving performance. The detailed setup is in §6. For an idealized workload with uniform request lengths, queuing delay is negligible even at high system loads. However, real-world LLM workloads (e.g., ShareGPT, Alpaca) are highly skewed. The long-tail distribution of their output lengths, in particular, creates severe head-of-line blocking. As the figure shows, this causes queuing delay to dominate, accounting for up to 90% of the total end-to-end latency. Consequently, optimizations targeting execution time alone are insufficient. The primary bottleneck that must be addressed

*Equal contribution.

is the queuing delay. While prior work like chunked prefill attempts to mitigate this, it misidentifies the core problem by splitting long inputs rather than addressing the more frequent bottleneck of long outputs. Furthermore, by breaking a single forward pass into multiple steps, this approach inherently degrades time-to-first-token (TTFT) performance due to additional memory access [57, 63].

We present FastServe, a distributed LLM serving system which exploits the autoregressive pattern and iteration-level scheduling to enable preemption at the granularity of each output token. Specifically, when one scheduled job finishes generating an output token, FastServe can decide whether to continue this job or preempt it with another job in the queue. This allows FastServe to use preemptive scheduling to eliminate head-of-line blocking problem and minimize latency.

The core of FastServe is a novel *skip-join* Multi-Level Feedback Queue (MLFQ) scheduler. MLFQ is a classic approach to minimize latency in information-agnostic settings [9]. Each job first enters the highest priority queue, and is demoted to the next priority queue if it does not finish after a threshold. The key difference between LLM inference and the classic setting is that LLM inference is *semi* information-agnostic, i.e., while the output length is not known *a priori*, the input length is known. Because of the autoregressive pattern of LLM inference, the input length decides the execution time to generate the first output token, which can be significantly larger than those of the later tokens (§4.1). For a long input and a short output, the execution time of the first output token dominates the entire job. We leverage this characteristic to extend the classic MLFQ with skip-join. Instead of always entering the highest priority queue, each arrival job joins an appropriate queue by comparing its execution time of the first output token with the quantum of the queues. The higher priority queues are skipped to reduce demotions.

Preemptive scheduling introduces a significant memory management challenge. To avoid redundant computation, LLMs maintain an intermediate state, the key-value (KV) cache (§2.2). While a non-preemptive scheduler's (e.g., FCFS) memory footprint is bounded by *running* jobs, a preemptive scheduler must preserve the KV caches for numerous preempted jobs. This additional overhead can exhaust the limited GPU memory. A naive solution, i.e., blocking new jobs when memory is full, is untenable as it reintroduces the head-of-line blocking. We therefore designed a *proactive* memory management mechanism. When GPU memory pressure becomes high, it offloads the KV caches of low-priority jobs to host memory. It then reloads these caches to the GPU immediately before the corresponding jobs are scheduled to resume. To hide the latency of these transfers, we employ pipelining and asynchronous memory operations, effectively overlapping communication with computation.

We implement a system prototype of FastServe and evaluate FastServe on different LLMs with real-world workloads.

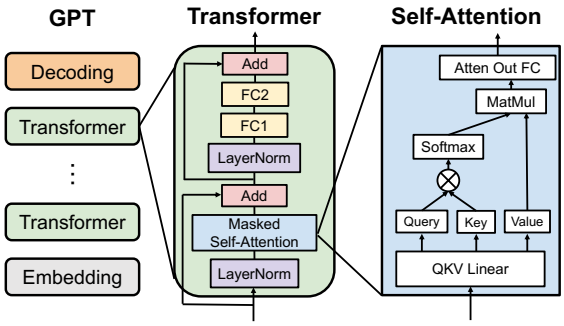


Figure 2: GPT-like model architecture.

The experiments show that compared to the state-of-the-art solution vLLM [31], FastServe improves the throughput by up to 6.1× under the same latency requirements.

2 Background and Motivation

2.1 LLM Inference and Applications

LLM inference. Current mainstream LLMs [12, 53, 61] are all based on the Transformer [54] architecture. As shown in Figure 2, the backbone of LLMs is a stack of Transformer layers where the attention module is the core component.

LLM inference proceeds iteratively, generating a single output token at each iteration. This autoregressive process relies on an attention mechanism that, in every iteration, requires the keys and values of all preceding tokens (i.e., both the original input and the generated output) to compute the subsequent token. This process continues until a unique <EOS> token, symbolizing the end of sequence, is generated, or a predetermined maximum output length is reached. This inference process markedly differs from traditional models like ResNet, where execution time is usually deterministic and predictable [24]. While each iteration's execution maintains these characteristics, the number of iterations (i.e., output length) is variable, resulting in an unpredictable total inference job execution time.

LLM applications. LLMs, renowned for their in-context and few-shot learning capabilities, are often transformed into powerful interactive agents through supervised fine-tuning and reinforcement learning techniques (e.g., PPO [46] and GRPO [47]). This transformation enables them to handle diverse tasks ranging from simple question-answering and summarization to complex, creative, and domain-specific problem-solving. However, the conversational paradigm of these applications introduces a critical challenge for the underlying serving infrastructure: the necessity of low-latency inference.

2.2 LLM Serving Systems

As the popularity of LLMs rapidly increases, inference serving systems have evolved to include optimizations specific to the unique architecture and iterative generation pattern of LLMs. Fairseq [41] suggests saving the keys and values in a *key-value cache* across iterations to avoid recomputation.

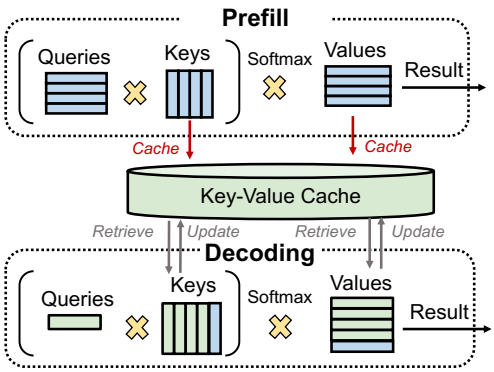


Figure 3: Demonstration of KV cache.

This optimization divides the inference procedure into two distinct phases: the *prefill phase* and the *decoding phase*. Figure 3 demonstrates the key-value cache usage in both phases. During the *prefill phase*, the LLM generates the key-value cache for the input prompt. In the *decoding phase*, the LLM only needs to compute for the newly generated token without re-computing previous key-value cache to reduce redundant computation.

As for scheduling optimization, Orca [59] proposes iteration-level scheduling where the serving engine schedules at the end of each iteration. After each iteration, completed jobs leave the batch, and newly arrived jobs can join in. Nevertheless, the GPU memory capacity limits the maximum batch size, and the strict service-level-objects (SLOs) of interactive applications also play a role in determining the appropriate batch size. vLLM [31] further improves the efficiency of LLM inference by introducing PagedAttention, which allocates the key-value cache gradually in block-grained during inference instead of allocating for the maximum output length at the beginning. To reduce interference between two phases, chunked prefill [5] further splits the prefill phase into multiple chunks, but it introduces extra memory access to LLM parameters and key-value cache [57, 63].

2.3 Opportunities and Challenges

Opportunity: preemptive scheduling. The major limitation of existing inference serving systems for LLMs [14, 59] is their reliance on simple FCFS (First-Come-First-Serve) scheduling and run-to-completion execution. As shown in Figure 1, this approach leads to severe head-of-line blocking. Queuing delay contributes up to 90% of the total latency in real workload, which significantly impacts the performance of LLM inference. The key opportunity to resolve this is to employ preemptive scheduling. Since inference is an autoregressive process that generates one token per iteration, preemption can occur at the fine granularity of one single token. Enabling a job to be paused after any token generation allows a scheduler to implement advanced policies that mitigate head-of-line blocking and optimize for metrics such as average latency. Nevertheless, integrating preemption into LLM serving systems introduces two significant challenges.

Challenge 1: variable job size. Shortest Remaining Processing Time (SRPT) [45] is a widely-used preemptive scheduling policy to minimize average latency. The effectiveness of SRPT hinges on knowing the total remaining service time for each job, a prerequisite that cannot be satisfied in LLM inference domain. Unlike one-shot prediction tasks like image classification, LLM inference involves multiple iterations. While the execution time for one iteration (generating one output token) can be determined based on the model architecture and hardware, the total number of iterations (i.e., the output sequence length) remains unknown and is challenging to predict since it depends on the semantics of the job. Real-world datasets collected from conversations with LLMs, like ShareGPT [4] and Alpaca [52], exhibit a long-tailed distribution of the output length and input length [31]. Consequently, SRPT cannot be directly employed for LLM inference to minimize the average latency.

Challenge 2: GPU memory overhead. A significant challenge of preemptive scheduling in LLM inference is its substantial GPU memory overhead. Unlike non-preemptive policies (e.g., FCFS) that only store the key-value (KV) cache for active jobs, a preemptive scheduler must also retain the KV caches of all preempted jobs in the pending state. This overhead is non-trivial. For instance, the KV cache for a single request to OPT 175B with a 512-token context requires 2.3 GB of memory for the key-value cache (§4.2). Given the scarcity of GPU memory, this memory pressure becomes a critical bottleneck that can undermine the viability of preemption. While prior works have proposed techniques to reduce KV cache memory, their solutions are insufficient in this context. Architectural modifications like Multi-Query [48] and Group-Query [6] Attention reduce memory by sharing KV tensors across attention heads, but this can degrade model quality, and memory consumption still grows linearly with sequence length. Orthogonally, memory management techniques like vLLM's paged attention mitigate memory fragmentation but do not reduce the intrinsic size of the KV cache itself. As models are increasingly trained for longer context lengths [33], the escalating memory footprint of the KV cache presents a formidable and increasingly urgent problem.

3 FastServe Overview

3.1 Desired Properties

LLMs come with unique characteristics that pose challenges to distributed computation and GPU memory consumption. Our goal is to develop an efficient inference serving system for LLMs that fulfills the following three requirements.

- **Low latency and high throughput.** For interactive LLM applications, where low latency is critical, our primary performance objective is to maximize serving throughput while adhering to a specific latency constraint.

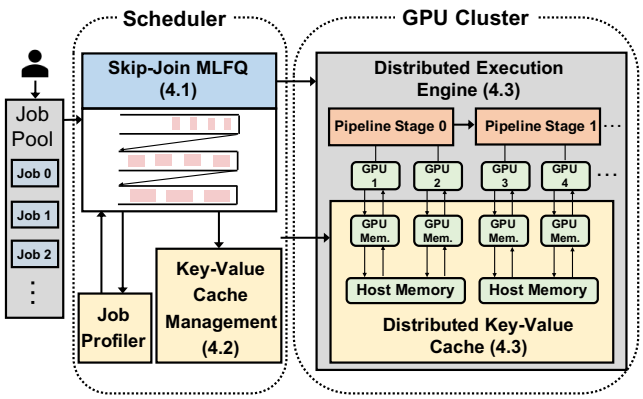


Figure 4: FastServe architecture.

- **Efficient GPU memory management.** LLMs pose a significant challenge in terms of GPU memory consumption, which necessitates an effective GPU memory management approach for both the model and intermediate states.
- **Scalable distributed execution.** The nature of LLMs demands multiple GPUs to enable distributed inference effectively, which requires the system to support scalable distributed execution cross GPU servers.

3.2 Overall Architecture

Figure 4 shows the architecture of FastServe. Jobs are submitted to the job pool. The scheduler utilizes information from the job profiler to determine the initial job priority and then places the job in the skip-join MLFQ (§4.1) to mitigate head-of-line blocking.

For execution, the scheduler picks the jobs based on their priority within the skip-join MLFQ to form a pre-defined maximum batch size and dispatches the batch to the distributed execution engine to perform one iteration. The distributed execution engine collaborates with the distributed key-value cache to access and update the key-value tensors relevant to the respective job. To tackle the challenge of limited GPU memory capacity, the key-value cache manager proactively swaps key-value tensors between GPU memory and host memory (§4.2).

To accommodate extreme large models such as OPT-175B, FastServe employs distributed inference, enabling both tensor parallelism and pipeline parallelism. FastServe incorporates extensions into the scheduler and key-value cache to enable seamless support for distributed execution (§4.3).

4 FastServe Design

In this section, we first introduce the skip-join MLFQ scheduler to minimize latency (§4.1). Then, we present a proactive KV cache management mechanism designed to effectively ameliorate the GPU memory capacity constraint (§4.2). Finally, we demonstrate how to apply these techniques to the distributed settings (§4.3).

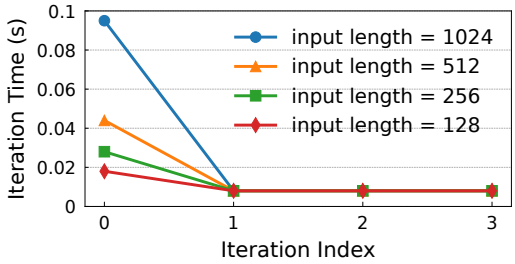


Figure 5: Execution time of the first four iterations.

4.1 Skip-Join MLFQ Scheduler

Strawman: fixed priority scheduling. To support preemptive scheduling, we need a priority-based scheduler to decide which jobs to preempt and which to execute. One naive solution is to assign a fixed priority to each job based on its input length. In this case, when prefill dominates the total latency, the fixed priority scheduling can approximate the optimal performance as the SRPT policy does. However, although this solution leverages the information of prefill, it ignores the characteristics of the decoding. Many real-world datasets like ShareGPT and Alpaca show a long tail distribution implying that jobs with a long output length also exist. When the decoding dominates the total latency, the fixed priority scheduling may deviate from the optimal performance of SRPT.

Strawman: naive MLFQ. Due to the indeterminate job size of LLM inference, directly applying SRPT is not feasible. In information-agnostic settings, Least-Attained Service (LAS) has been shown to approximate SRPT effectively. Due to the job switching overhead of LAS, the practical approach is Multi-Level Feedback Queue (MLFQ) which has gained popularity in various scheduling systems [7, 9, 13, 23, 27]. MLFQ operates multiple queues, each with a different priority level. Upon arrival, a job enters the highest priority queue and gets demoted to the next level queue if its execution time exceeds a quantum. The value of quantum is a tunable parameter assigned to each queue, e.g., higher priority queues typically have shorter quantum values.

Although MLFQ assumes no prior knowledge of the job size, it is not well suited for LLM serving due to the distinct two-phase execution pattern of LLM inference. Figure 5 shows the iteration time of OPT 2.7B on an NVIDIA A100, varying the input sequence length. Notably, the prefill (i.e., the first iteration) time exceeds the decoding duration. As the input sequence length increases, so does the prefill time. This behavior can be attributed to the key-value cache optimization (§2.2). During the first iteration, computations for all key-value tensors of input tokens are performed and cached. In subsequent iterations, only one newly generated token's key-value tensors are computed, and the rest are retrieved from the cache.

When employing the original MLFQ, a job is immediately assigned to the highest priority queue upon arrival. However, due to its substantial prefill time, the job may deplete its

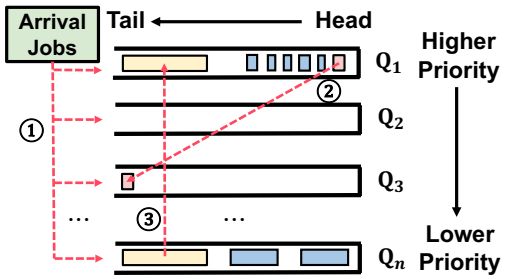


Figure 6: Skip-join MLFQ with starvation prevention.

quantum before completing its first iteration. This situation presents a scheduling dilemma. If the scheduler preempts the job, intermediate activations are dropped and recomputed later, resulting in a waste of valuable computing resources and time. On the other hand, if the scheduler chooses not to preempt the job, it violates the fundamental design purpose of MLFQ and potentially suffers from head-of-line blocking once again.

Our solution: skip-join MLFQ. The key insight of our design is to leverage the *semi* information-agnostic setting of LLM inference to address the aforementioned issues of the strawman solutions. While the number of iterations (i.e., the output length) remains unknown beforehand, the execution time of each iteration is predictable. For each iteration, the execution is similar to the traditional one-shot DNN inference, whose execution time is highly predictable [24, 35]. A lightweight profiling process can collect iteration times under the given hardware and model specification to construct an accurate latency model for the prefill and decoding phases.

Based on this insight, we propose a skip-join MLFQ scheduler tailored for LLM inference. Our scheduler efficiently manages the movement of jobs among various priority queues in a skip-join manner. In the MLFQ, we have n priority queues, namely $Q_1, Q_2, ..., Q_n$, each with a distinct quantum $q_1 < q_2 < ... < q_n$. The conventional MLFQ scheduler initially assigns a newly arrived job to the highest priority queue, i.e., Q_1. Once the job exhausts the allocated quantum in Q_1, it is subsequently demoted to Q_2. As shown in Figure 6, FastServe differs from the original MLFQ in that when a job arrives, FastServe leverages accurate profiling to predict the prefill time (t_{init}) and ❶ skip-joins the job to the highest priority queue (q_i) subject to $q_i \geq t_{init}$. When a job consumes its allotted quantum before completion, the scheduler demotes ❷ the job based on its current priority and next iteration time.

Avoiding perpetual starvation. It is important to note that the skip-join and demotion operations may result in starvation for jobs with long input and output. To address this problem, the scheduler periodically examines the starved jobs and ❸ promotes them to the highest priority queue, i.e., Q_1. This allows FastServe to address head-of-line blocking while mitigating starvation. We evaluate the effectiveness of the starvation prevention mechanism by showing the tail latency in §6.2.

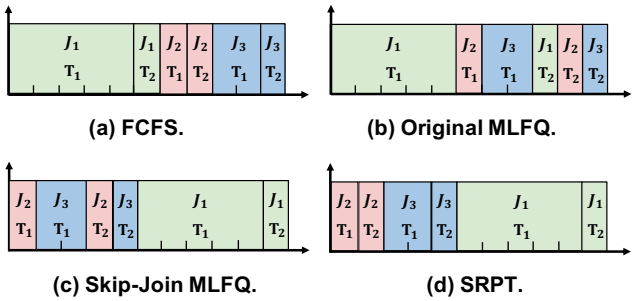


Figure 7: Execution timeline of three jobs with different scheduling algorithms.

Example. Figure 7 shows an example to demonstrate the effectiveness of FastServe's skip-join MLFQ scheduler. In the example, three jobs arrive at the same time in the order of J_1, J_2, J_3. $T_1(J_i)$ denotes the prefill time of job J_i, and $T_2(J_i)$ denotes the decoding time. We assume that both skip-join and original MLFQ utilize four priority queues with quantum values of 1, 2, 4, and 8. Additionally, SRPT serves as the oracle with the optimal average latency.

As Figure 7 shows, the average latency of FCFS, original MLFQ, skip-join MLFQ, and SRPT are 4.23, 5, 3.3, and 3, respectively. FCFS and original MLFQ encounter the head-of-line blocking problem, where job J_1 blocks the remaining jobs, leading to long average latency. Skip-join MLFQ addresses this issue by skip-joining job J_1 to the low-priority queue, achieving performance similar to optimal SRPT. Generally, algorithms that have access to more information perform better than those with limited information. We evaluate these scheduling choices under real workloads in §6.3

Algorithm. Algorithm 1 shows the pseudo-code of the skip-join MLFQ scheduler. The scheduler has a set of priority queues $Q_1, Q_2, ..., Q_n$ with quantum values $q_1, q_2, ..., q_n$, and receives a set of newly arrived jobs J_{new}. It schedules a batch of *MaxBatchSize* jobs for execution. The skip-join part (❶ in Figure 6) corresponds to lines 6–9, and the demotion and starvation prevention parts (❷ and ❸ in Figure 6) correspond to lines 16–17 and lines 19–21, respectively. There are two notable details. First, the scheduler demotes a job to an η times lower priority queue based on its next iteration time. FastServe sets the quantum of the lower priority queue to two times of that of the higher priority queue, which aligns with previous work [23] on MLFQ. The quantum of the highest priority queue is set to the minimum iteration time. The second detail concerns how the scheduler identifies starved jobs governed by the parameter α. FastServe tunes α based on the user-specified SLO, which is set to 300 ms by default.

4.2 Proactive Key-Value Cache Management

Although the skip-join MLFQ scheduler provides iteration-level preemption to approximate SRPT to achieve lower latency without prior knowledge of the exact job size, it exacerbates the pressure of GPU memory consumption. Figure 8

Algorithm 1 Skip-Join Multi-Level Feedback Queue Scheduler

Input: Queues $Q_1, Q_2, ..., Q_n$, newly arrived jobs J_{new}
Output: Jobs to be executed J_{out} for one iteration
procedure SKIPJOINMLFQSCHEDULER
    Prefill: $J_{out} \leftarrow \emptyset$.
    // Skip-join newly arrival jobs.
    for $job \in J_{in}$ **do**
        $init\_time \leftarrow P.getNextIterTime(job)$
        $p_{job} \leftarrow min\ i,\ s.t.\ q_i \geq init\_time$
        $Q_{p_{job}}.push(job)$
    for $job \in \{Q_1, Q_2, ..., Q_n\}$ **do**
        $job.outputNewGeneratedToken()$
        $p_{job} \leftarrow job.getCurrentPriority()$
        if $job.isFinished()$ **then**
            $Q_{p_{job}}.pop(job)$
        // Demote jobs.
        if $job.depleteQuantum()$ **then**
            $Q_{p_{job}}.pop(job), Q_{p_{job}+\eta}.push(job)$
        // Promote starved jobs.
        if $job.starveTime \geq \alpha$ **then**
            $Q_{p_{job}}.pop(job), Q_1.push(job)$
            $job.starveTime \leftarrow 0$
    // Schedule jobs to execute.
    for $job \in \{Q_1, Q_2, ..., Q_n\}$ **do**
        if $job.isReady()$ **and** $|J_{out}| < MaxBatchSize$ **then**
            $J_{out}.push(job)$

shows the key-value cache memory consumption of FCFS and skip-join MLFQ for OPT 2.7B model under a synthetic workload. Although we choose a relatively small model and limit the maximum output length to 20, the peak KV cache memory overhead for skip-join MLFQ can be 7× larger than that of FCFS. The GPU memory demand becomes even more pronounced when deploying larger LLMs like OPT 175B.

The reason under the hood is that compared to the run-to-completion policy in the existing serving systems, iteration-level preemption provided by the skip-join MFLQ increases the number of ongoing jobs in the system. Except for the key-value tensors of running jobs, the skip-join scheduler also needs to store the key-value tensors for preempted jobs at the pending state. Unlike process states in traditional operating system, the intermediate states in LLM inference—namely, the key-value tensors—can be substantially larger. Formally, assuming the model weights and all computations are in FP16, the memory required to store the key-value cache for a single job is $4 \times lh(s+t)$, where s denotes the input sequence length, t the output sequence length, h the transformer hidden dimension, and l the number of transformer layers. Taking OPT-175B as an example ($l = 96, h = 12288$), a single request with an input length $s = 512$ will incur a GPU memory overhead of up to 2.3 GB during the prefill phase. As generation proceeds, the output sequence length t increases, with each additional token incurring 4.6 MB of memory consumption. With the emergence of the test-time scaling law [17, 40], LLMs now tend to produce longer outputs, further exacerbating the memory overhead.

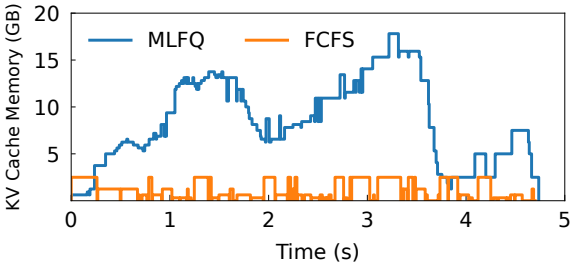


Figure 8: The key-value cache memory consumption for OPT 2.7B under different schedulers. The workload follows a Gamma Process with rate=64 and CV=4. The maximum output length is set to 20 to avoid GPU out of memory.

At the same time, GPU memory is a scarce resource when deploying LLMs. Typically, GPU memory is much smaller than the host memory. For instance, NVIDIA A100 GPU has a maximum of 80 GB GPU memory. Besides, a large portion of GPU memory is allocated to storing the weights of LLMs. The space available for key-value tensors is therefore limited, which constrains the runtime batch size and reduces GPU utilization. As a result, the GPU memory capacity constrains the potential benefits of the skip-join MLFQ scheduler.

Strawman solution 1: defer newly arrived jobs. To avoid out-of-memory (OOM) errors, a naive solution is to simply *defer* the execution of newly arrived jobs when the GPU memory is not sufficient and keep scheduling current in-memory jobs until they finish. This straightforward solution is widely used in existing serving systems, such as vLLM [31]. In this manner, although new jobs are assigned with higher priority, they are blocked to await the free memory space. Under extreme long sequence inference settings, this solution would degenerate MLFQ to FCFS, suffering from head-of-line blocking again.

Strawman solution 2: kill and re-compute low-priority jobs. Another straightforward solution is to *kill* some low-priority jobs and release their key-value cache to make room for newly arrived high-priority jobs. This solution has two problems. First, the killed jobs lose their generation states, necessitating to rebuild their key-value tensors. This results in the waste of valuable computational resources and time. Second, it may lead to a live-lock. When high-priority jobs arrive and the memory is not enough to accommodate, ongoing lower-priority jobs are killed. To avoid starvation, killed jobs may be promoted to the highest-priority queue once they have waited longer than the specified $STARVE\_LIMIT$. In such cases, a promoted job may preempt the currently executing job, which in turn might have just preempted the promoted job in the previous step.

Our solution: proactive key-value cache swapping. Under the strict GPU memory capacity constraints, the two strawman

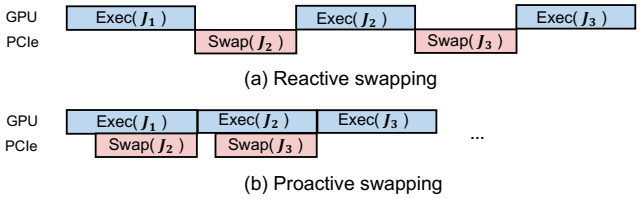


Figure 9: Comparison of reactive and proactive swapping.

solutions have to sacrifice either the performance of newly arrived jobs or the efficiency of low-priority jobs. To overcome this dilemma, our key observation is that the key-value tensors only need to be reserved in the GPU memory when their corresponding jobs get scheduled. Based on this observation, FastServe extends the space of the key-value cache from GPU memory to the host memory. FastServe swaps out inactive key-value tensors of jobs to the host memory to accommodate additional pending jobs, and swaps in key-value tensors back to the GPU memory for upcoming jobs.

However, the overhead of swapping is not negligible compared to the token generation time. When deploying OPT-175B, the key-value tensors of a single job can occupy tens of gigabytes of memory, and the resulting transmission overhead between host memory and GPU memory can reach hundreds of milliseconds even under the full bandwidth of PCIe 4.0×16. Meanwhile, the token generation time during decoding is typically less than 50 ms due to application requirements; therefore, a simple reactive swapping mechanism can incur substantial overhead.

Instead, FastServe employs a *proactive* key-value cache swapping algorithm to mitigate the adverse effects of swapping overhead. The key insight is to overlap the LLM inference for running jobs with the data transmission for pending jobs so that the swapping overhead is out of the critical path of LLM inference. Figure 9 illustrates an example. Instead of swapping key-value tensors of pending job J_2 after job J_1 is preempted or finished, the proactive algorithm swaps the key-value tensors of J_2 in advance. In this way, the swapping overhead of J_2 effectively overlaps with the GPU kernel execution of J_1, thereby achieving high GPU utilization. As swapping in one job consumes expensive GPU memory, the job swapping order is crucial for GPU memory efficiency.

Job swapping order. Frequently swapping in and out unnecessary key-value tensors incurs additional thrashing overhead if swapping in and out one job with high priority. The swapping overhead can increase to exceed the execution time, leading to a deterioration in the performance of overlapping. To address this issue, FastServe calculates the estimated next scheduled time (ENST) for each job to decide the swapping order. The ENST is the time when the job will be scheduled to execute next time. The job with the largest ENST will be swapped out first, and the job with the smallest ENST will be swapped in first. Typically, a job with lower priority is scheduled for later execution. However, owing to the starvation prevention mechanism, a job of lower priority might be elevated to a higher priority queue. Consequently, even a low-priority job can sometimes be executed first.

In this case, for job i, FastServe considers the time to promote this job and the sum of execution time of all jobs with higher priorities before executing i simultaneously. Formally, let the time threshold for promoting job i be $T_{promote}(i)$. As for the sum of execution time of all jobs with higher priorities before executing i, we assume those jobs do not finish earlier before being demoted to the priority queue of job i. The execution time of job j with a higher priority can be calculated as follows (i.e., job j is demoted from $j.priority$ to $i.priority$ one by one):

$$T_{execute}(i,j) = \sum_{i.priority<k\leq j.priority} q_k$$

where $i.priority$ is the priority of job i, and q_k is the quantum of the priority queue with priority k. Based on this, the sum of execution time of all jobs with higher priorities than job i is defined as:

$$T_{execute}(i) = \frac{1}{B} \sum_{i.priority<j.priority} T_{execute}(i,j)$$

where B is the maximum batch size of jobs. At last, taking both the promotion for starvation prevention and the execution of higher priority jobs into consideration, the ENST of job i is calculated as:

$$ENST(i) = \min(T_{promote}(i), T_{execute}(i))$$

This ENST definition serves as a means to estimate the expected scheduling time for the next generation of job i. Therefore, using this metric to decide the order of swapping makes the key-value tensors of active jobs reside predominantly in GPU memory, and those of inactive jobs are more inclined to reside in host memory.

Handling a burst of new jobs. The proactive key-value cache swapping strategy is designed for the skip-join MLFQ scheduler. In scenarios where a significant influx of new jobs (with high priority) occurs, the cache management system is forced to evict jobs reactively, adversely affecting the performance of these new jobs. To mitigate this, FastServe reserves some idle key-value cache slots specifically for new jobs, ensuring immediate availability without the need for reactive job swapping. This approach guarantees the performance of new jobs. The number of idle slots is based on historical job arrival patterns. A higher frequency of job bursts necessitates a larger number of reserved slots. We evaluate the effectiveness of proactive key-value cache swapping in §6.3

4.3 Support for Distributed LLM Serving

Previous research shows that the effectiveness of LLMs empirically adheres to the scaling law concerning the quantity of model parameters [30]. However, it is important to note

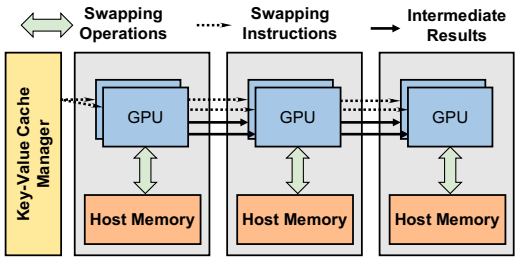


Figure 10: Overlapping key-value cache offloading with intermediate result transmission.

that the memory usage of an LLM also exhibits proportionality to the number of parameters. A prime example is OPT 175B, which, even when stored in half-precision, demands a staggering 350GB of GPU memory solely to accommodate its weights. Furthermore, additional memory is required for handling intermediate states during runtime. Therefore, LLM often needs to be split into multiple pieces and served in a distributed manner with multiple GPUs.

Tensor parallelism (TP) [38, 50] and pipeline parallelism (PP) [28, 37] are two most widely-used techniques for distributed LLM serving. FastServe supports the hybrid of these two parallel techniques for serving LLMs. An LLM is composed of a series of operators over multi-dimensional tensors. TP splits each operator across multiple devices, with each device executing a portion of the computation in parallel. It augments both computational and memory resources available to a single job, consequently reducing the latency of each iteration. However, substantial communication is required to split the input and gather the output from different GPUs, therefore TP is typically used intra-node.

PP splits an entire LLM computation graph into multiple stages and executes them on different GPUs in a pipeline fashion. During inference, each stage computes a part of the entire computation graph and transmits the intermediate results to the next stage in parallel. It incurs less communication overhead compared to TP, therefore can be used inter-nodes. Since multiple running batches are under processing simultaneously in different stages, PP can increase overall throughput, but for a single job, it may slightly increase latency. Supporting PP requires FastServe to handle multiple batches in the distributed engine at the same time.

Job scheduling in distributed serving. In the traditional MLFQ, if no new job arrives, the scheduler schedules the job with the highest priority and executes it until it finishes or is demoted. However, with pipeline parallelism, the scheduler schedules at the granularity of individual stage. Once a job completes its first stage and transmits the intermediate results to the subsequent stage, a decision point arises for the scheduler regarding the next job to set in motion. In this case, the scheduler cannot follow the traditional MLFQ that keeps scheduling the same job until demotion, because the job is still in progress. To preserve the semantics of MLFQ, FastServe still keeps the running job in the priority queue, but schedules the highest priority job in the pending state. Thus, the early jobs in a queue can expedite their quantum completion.

Model	Size	# of Layers	# of Heads	Hidden Size
LLama3-8B	16GB	32	32	4096
OPT-13B	26GB	40	40	5120
OPT-66B	132GB	64	72	9216
OPT-175B	350GB	96	96	12288

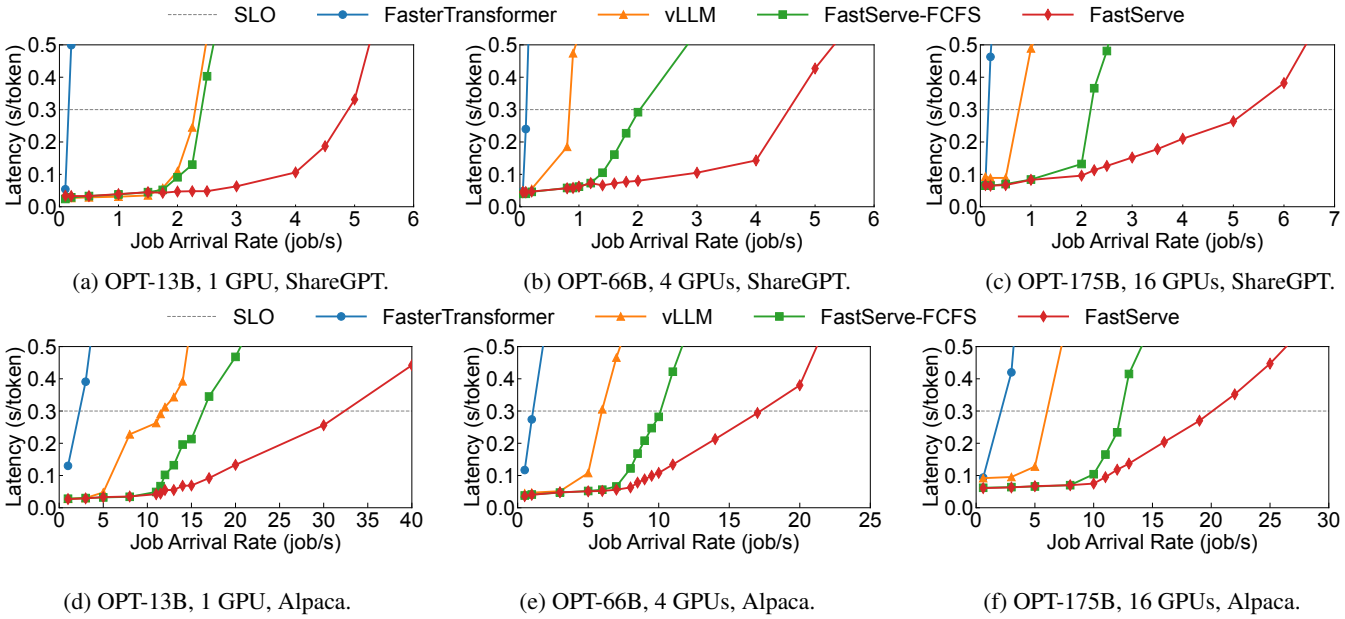
Table 1: Model configurations.

Key-value cache management in distributed serving. Given that the key-value cache occupies a large fraction of GPU memory, the key-value cache of FastServe is also partitioned across multiple GPUs. In LLM inference, each key-value tensor is used by the same stage of the LLM. Therefore, FastServe partitions key-value tensors as tensor parallelism requires, and assigns each key-value tensor to the corresponding GPU so that all computation on a GPU only needs local key-value tensors on the same GPU.

The proactive key-value cache swapping mechanism of FastServe is also distributed. Because different stages of the LLM process different jobs at the same time, each stage may offload or upload different key-value tensors independently. To reduce redundant control, before processing the intermediate result sent from the previous stage, the current stage does the same offloading or uploading action as the previous stage does. The intermediate result transmission and key-value cache swapping occur in parallel, so the overhead of key-value cache swapping is further reduced. As shown in Figure 10, when the intermediate result is sent to the next stage, the next stage receives the swapping instructions and can swap the key-value cache at the same time if needed. The key-value cache swapping mechanism only needs to decide the offloading or uploading of the first stage. When using tensor parallelism splitting the first stage into multiple chunks, a centralized key-value cache swapping manager instructs all chunks in the first stage to offload or upload the key-value tensors owned by the same job.

5 Implementation

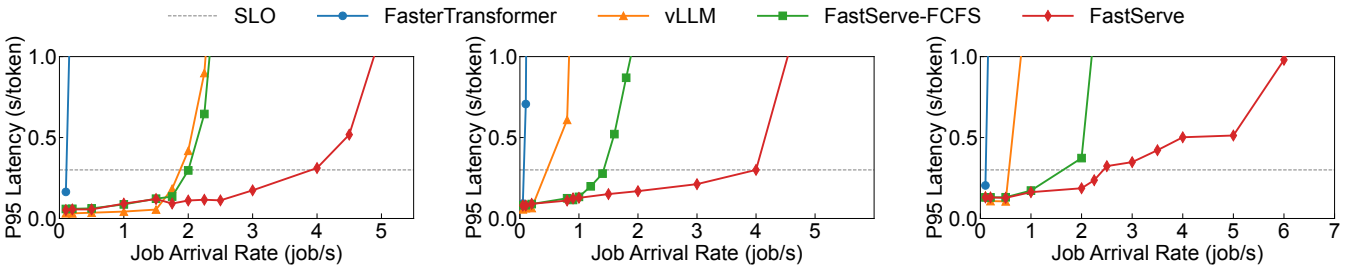
FastServe is a distributed LLM inference serving system with a RESTful API frontend, a scheduler, and a distributed execution engine. The frontend and scheduler are implemented with 2.9K lines of Python code. The distributed execution engine is implemented with 8.1K lines of C++/CUDA code. The frontend supports OpenAI API compatible interface where clients can specify the sampling parameters like maximum output length and temperature. The scheduler implements the skip-join MLFQ and proactive swapping policies. The distributed execution engine uses Ray [36] actor to implement GPU workers which execute the LLM inference and manage the key-value cache in a distributed manner. We implement popular open-source LLMs such as OPT in C++ to achieve



(a) OPT-13B, 1 GPU, ShareGPT. (b) OPT-66B, 4 GPUs, ShareGPT. (c) OPT-175B, 16 GPUs, ShareGPT.

(d) OPT-13B, 1 GPU, Alpaca. (e) OPT-66B, 4 GPUs, Alpaca. (f) OPT-175B, 16 GPUs, Alpaca.

Figure 11: Average latency of different serving systems with OPT models on real workloads.



(a) OPT-13B, 1 GPU, ShareGPT. (b) OPT-66B, 4 GPUs, ShareGPT. (c) OPT-175B, 16 GPUs, ShareGPT.

Figure 12: Tail latency of different serving systems with OPT models on real workloads.

better performance and scalability than the popular Python implementations in Huggingface [56]. We also implement custom CUDA kernels to support Orca's [59] iteration-level scheduling and vLLM's [31] PagedAttention.

6 Evaluation

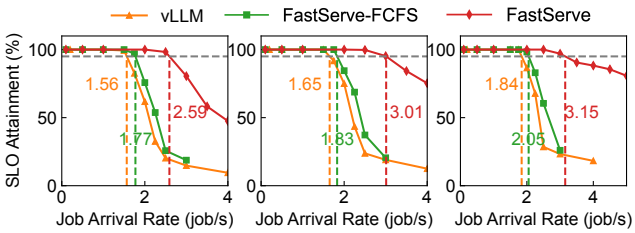
6.1 Methodology

Testbed. The end-to-end experiments (§6.2) use two AWS EC2 p4d.24xlarge instances. Each instance is configured with eight NVIDIA A100 40GB GPUs connected over NVLink, 1152 GB host memory, and PCIe 4.0×16. Due to the limited budget, the experiments for design choices (§6.3) use one NVIDIA A100 40GB GPU in our own testbed to validate the effectiveness of each component.

LLM models. We choose the representative LLM family, OPT [61], which is widely used in both academia and industry. We select common model sizes. We also include Llama3-8B which uses Grouped Query Attention (GQA) to show the generalizability of FastServe across different attention architectures. Table 1 lists the model configurations. We use FP16 precision in all experiments. Noteably, the techniques proposed in FastServe are orthogonal to the model size and we evaluate FastServe on multi-GPU setting in §6.2 to show its effectiveness on large models to show its generalizability.

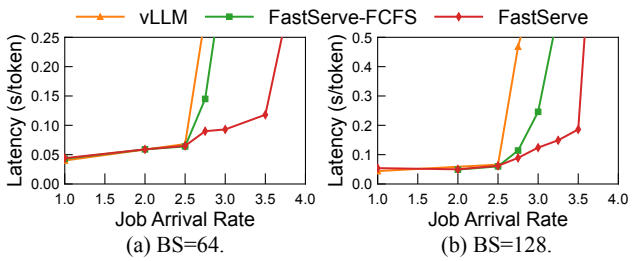
Workloads. Similar to prior work on LLM serving [31], we generate workloads based on ShareGPT [4] and Alpaca [52] datasets. These datasets contain real-world inputs and outputs of LLM services. The ShareGPT dataset is composed of user-shared conversations with ChatGPT [4]. The Alpaca dataset is generated by GPT-3.5 with self-instruct [52]. Since these datasets do not include the arrival time, we follow prior work [31] to generate the arrival time for each request following a Poisson process parameterized by the arrival rate.

Evaluation metrics. Similar to prior works [31, 59], we measure the average per-token latency, which is calculated as the mean of every job's end-to-end latency divided by its output length. In addition, we also report the P95 tail latency to reflect the fairness. For comparison, we set a latency SLO and



(a) 5× SLO. (b) 10× SLO. (c) 20× SLO.

Figure 13: P95 goodput under different SLOs.



(a) BS=64. (b) BS=128.

Figure 14: Effectiveness of FastServe under large batch size.

compare the maximum throughput each system can achieve under the SLO. We follow prior work [44] to set the latency SLO to 10× of the latency of a single iteration in the decoding. Specifically, we set SLO to 0.3 seconds based on our profiling. We also report the SLO attainment, which is the percentage of jobs that meet the SLO requirement.

Baselines. We compare FastServe with three baselines. For fair comparison, all baselines use the same tensor parallelism size, pipeline parallelism size, and batch size as FastServe. The maximum batch size is aligned with the actual maximum batch size reported in previous works [31, 59].

- **FasterTransformer** [14]: It is a production-grade inference engine from NVIDIA. It supports both tensor parallelism and pipeline parallelism. However, it adopts job-level scheduling and short jobs are blocked by long jobs in the same batch. We use FasterTransformer v5.3.
- **vLLM** [31]: It is the state-of-the-art LLM serving system that supports iteration-level scheduling [59] and PagedAttention [31] to reduce memory fragmentation caused by key-value cache. However, it uses a simple FCFS scheduler with run-to-completion execution, which suffers from head-of-line blocking. We use vLLM v0.6.1, which is the latest version when the experiments were conducted. We keep all configurations the same as FastServe except scheduling mechanism.
- **FastServe-FCFS**: It uses the same distributed execution engine of FastServe, but it does not use techniques proposed in §4. This baseline helps differentiate the speedup brought by the techniques proposed in this paper from that by the efficient implementation of FastServe.

6.2 End-to-End Performance

We first compare the end-to-end performance of four systems under two workloads when serving three models of different sizes on single-GPU, multi-GPU, and multi-node environments, respectively.

The first row of Figure 11 shows the results under the ShareGPT dataset. Because FasterTransformer does not support iteration-level scheduling, it cannot return early finished jobs in the batch and add new jobs into the batch to reduce latency, leading to the head-of-line blocking even when the job arrival rate is small. Therefore, FastServe outperforms FasterTransformer by 31.5–74.9× in terms of throughput under the SLO. As the state-of-the-art serving system, although vLLM significantly accelerate inference compared to FasterTransformer and reduce GPU memory consumption, vLLM suffers from FCFS scheduling. Because a large portion of the end-to-end latency is the queuing delay, optimizing the execution time is not enough. Equipped with the skip-join MLFQ scheduler, FastServe significantly reduces the queuing delay and outperform vLLM by 2.1–6.1×. Besides, FastServe-FCFS outperforms vLLM, because it uses more efficient C++ implementation and fuses more operations into fewer GPU kernels. But it still suffers from the head-of-line blocking problem, which makes it slower than FastServe by 2–4×.

The second row of Figure 11 shows the experiment results under the Alapca dataset. Since the job size of Alpaca is smaller than that of the ShareGPT dataset, all serving systems can maintain low latency even when the rate is relatively higher than that under the ShareGPT dataset. However, the performance gain of FastServe is similar. Without iteration-level scheduling, FasterTransformer is the slowest system and FastServe outperforms it by 9.5–15.8×. vLLM achieves better performance than FasterTransformer, but it still suffers from the head-of-line blocking problem. As a result, FastServe outperforms vLLM by 2.75–3.5×. With our efficient implementation, FastServe-FCFS also outperforms vLLM, but it is still slower than FastServe by 1.6–2×.

Impact on tail latency. A potential concern of preemptive scheduling and MLFQ is that it can cause starvation for long jobs and hurt tail latency. FastServe incorporates a starvation prevention mechanism in its skip-join MLFQ scheduler (§4.1). To demonstrate the effectiveness of the starvation prevention mechanism, we measure the 95% latency of all the systems under the ShareGPT dataset. As shown in Figure 12, FastServe significantly improves the throughput of LLM inference jobs under the same SLO requirement for tail latency. For example, when serving OPT-175B, compared to FastServe-FCFS, FastServe improves the throughput by up to 1.5×. FastServe also outperforms FastServe-FCFS by 2–2.8× when serving OPT-13B and OPT-66B. FastServe achieves up to 8.1× and 59.8× performance improvement compared to vLLM and FasterTransformer, respectively. The results show that although FastServe is designed to reduce average latency, it can also significantly reduce tail latency of LLM inference jobs. Prioritizing short jobs with the skip-join MLFQ scheduler can effectively reduce the head-of-line blocking problem and does not hurt the tail latency. Even for long jobs, Fast-

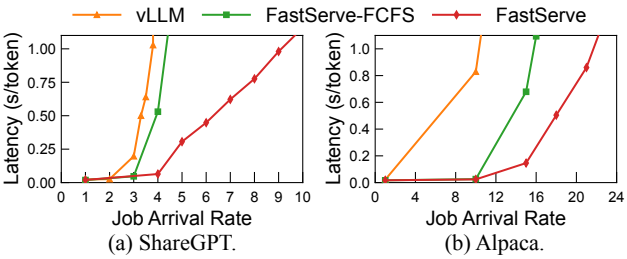


Figure 15: Effectiveness of FastServe under GQA architecture.

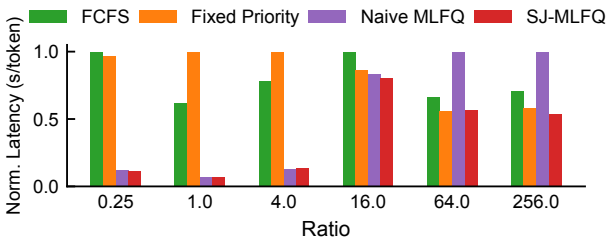


Figure 16: Effectiveness of skip-join MLFQ.

Serve can still accelerate them by reducing their queuing delay. The starvation prevention mechanism ensures that long jobs can be scheduled in a reasonable time.

Impact on goodput. To evaluate goodput under varying SLOs when serving the OPT-13B model, we measure the P95 goodput, i.e., the throughput at which 95% of jobs meet the SLO for both prefill and decoding phases, consistent with prior research [35,57,63]. We configured SLOs at 5×, 10×, and 20× the light-load latency of each respective phase. As shown in Figure 13, FastServe consistently achieves the highest P95 goodput, outperforming vLLM by 1.66–1.82× and FastServe-FCFS by 1.46–1.64×. These findings underscore FastServe's effectiveness in enhancing system throughput while adhering to the SLOs of the both stages.

Performance under large batch size. To evaluate the performance of FastServe under large batch size, we compare the performance of FastServe with vLLM and FastServe-FCFS when serving OPT-13B with batch size 64 and 128 on the ShareGPT dataset. These batch size is larger than the maximum batch size used by previous work [31] under the same setting. As shown in Figure 14, FastServe outperforms vLLM and FastServe-FCFS by up to 1.43× and 1.29× under batch size 64, and 1.51× and 1.17× under batch size 128. The results demonstrate that skip-join MLFQ can accelerate LLM inference and FastServe can effectively improve the system's throughput under large batch size.

Performance under GQA model architecture. To evaluate the performance of FastServe under increasingly adopted Grouped Query Attention (GQA) [6], we compare FastServe with vLLM and FastServe-FCFS when serving Llama3-8B [22] on both ShareGPT and Alpaca. As shown in Figure 15, FastServe consistently outperforms vLLM and FastServe-FCFS by 2.1× and 1.6× on ShareGPT, and 3.0× and 1.4× on Alpaca. This demonstrates that FastServe can effectively improve system throughput under GQA, an architecture more friendly to key-value cache efficiency.

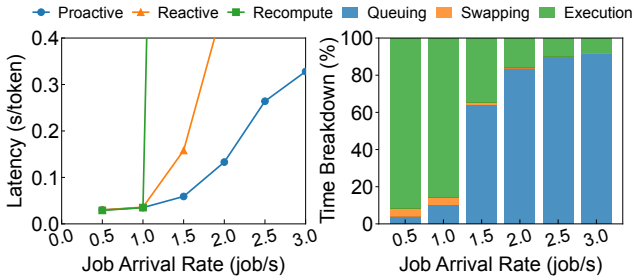
6.3 Design Choices

Effectiveness of skip-join MLFQ. To demonstrate the effectiveness of FastServe's skip-join MLFQ scheduler, we compare its performance against FCFS, naive MLFQ, and Fixed Priority while serving the OPT-13B model. We use the ShareGPT dataset to generate jobs and alter the ratios between input and output lengths while preserving the original length distribution. This adjustment reflects the current trend of expanding LLMs' context window size [8, 21, 57]. Figure 16 presents the experiment results across these ratios. FCFS consistently exhibite high latency due to head-of-line blocking, irrespective of the ratio. Naive MLFQ performe well at low ratios where prefill and decoding times are similar but struggle with longer prefill time as the ratio increases. Conversely, Fixed Priority excels at high ratios where prefill dominates but underperforms at low ratios by disregarding decoding. Leveraging its semi-information-agnostic approach, skip-join MLFQ consistently outperforms FCFS, naive MLFQ, and Fixed Priority by up to 8.9×, 1.87×, and 13.9×, respectively.

Effectiveness of proactive key-value cache management. We evaluate the effectiveness of FastServe's proactive key-value cache management by comparing its performance against two baselines, `Recompute` and `Reactive` (defined in §4.2), while serving OPT-13B on the ShareGPT dataset. The results are depicted in Figure 17(a). Under low request arrival rates, ample GPU memory allows all key-value caches to reside concurrently, resulting in comparable performance among the strategies. Conversely, higher arrival rates lead to GPU memory contention, forcing cache preemption and revealing distinct performance characteristics for each approach.

Under memory pressure, `Recompute` discards low-priority key-value (KV) caches, incurring recomputation overhead. As shown in Figure 17(a), avoiding this overhead allows proactive swapping to outperform `Recompute` by 2.7×. `Reactive` swaps low-priority caches to host memory when GPU memory is insufficient and retrieves them when needed, but these transfers block computation as they are on the critical path. In contrast, proactive swapping anticipates memory requirements, preemptively swapping low-priority caches out and high-priority caches in. This strategy overlaps data transfers with computation, achieving a 1.7× speedup over `Reactive`.

We further analyze the overhead of proactive swapping by dividing end-to-end latency into queuing delay, execution time, and swapping time. Figure 17(b) reveals that swapping time is less than 5% of the total latency. This is because proactive swapping largely overlaps with other job executions and the swapping time is negligible.



(a) OPT-13B on ShareGPT. (b) Latency breakdown.

Figure 17: Effectiveness of proactive key-value cache management. (a) Comparison between different key-value cache management policy. (b) Latency breakdown of FastServe.

Performance comparison with chunked prefill. To further evaluate FastServe, we compare it against chunked prefill, a method that divides long-context prefill into smaller segments to reduce decoding slowdowns caused by continuous batching. While chunked prefill partially mitigates the head-of-line blocking issue, it cannot fully eliminate it due to the additional overhead introduced by repeated partitioning. In contrast, FastServe addresses this limitation by employing an online scheduling approach, resulting in up to 1.5× performance improvement over chunked prefill. The experimental results are demonstrated in Figure 18.

7 Related Work

Preemptive scheduling. Many job scheduling solutions use preemptive scheduling. For DL workloads, Tiresias [23] uses MLFQ to optimize job completion time for DL training jobs. Pipeswitch [11] and REEF [25] provide efficient GPU preemption to run latency-critical and best-effort DL tasks together. Different from them, FastServe targets a new scenario, LLM inference serving. In the field of LLM serving, Llumnix [51] proposes a dynamic scheduling algorithm for load balancing and isolation across multiple instances, while FastServe focuses on preemptive scheduling within a instance. Sarathi-Serve [5] proposes chunked-prefills to mitigate the interference between the prefill and decoding phases. However, as shown in Figure 18, FastServe can further reduce the head-of-line blocking caused by the prefill chunks and the decoding phase. QLM [43] uses queue management to optimize SLO attainment for LLM inference, whose goal is different from FastServe. Besides, FastServe further proposes proactive swapping to mitigate the memory overhead of preemption.

Inference serving. Many traditional model serving systems [15, 16, 24, 39, 60] only focus on serving relatively small models in a cluster without awareness of characteristics of LLMs. Recently, several serving systems are proposed to optimize Transformer-based LLMs [19, 32, 35, 49, 59]. Orca [59] and vLLM [31] considers the autoregressive generation pattern of LLMs. However, due to their FCFS policy, they suffer from severe head-of-line blocking problem. VTC [49] focus on the fairness of LLM serving but does not consider the

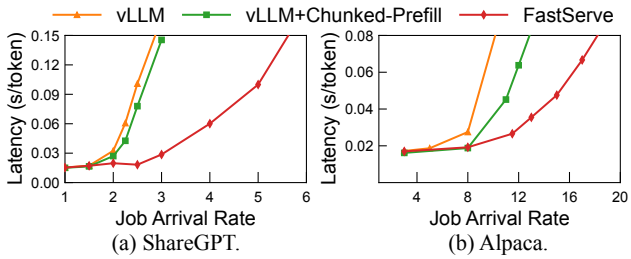


Figure 18: Comparison between FastServe and chunked prefill.

preemption scenario. Splitwise [42] and DistServe [63] disaggregates the prefill and decoding phase to eliminate the interference between them and thus optimize execution latency. LoongServe [57] uses elastic sequence parallelism to dynamically set degree of parallelism for different requests at different phases. These systems are orthogonal to FastServe.

Memory optimization for LLMs. Some work [10,55] targets training, which is orthogonal to the serving scenario. Quantization [18, 20, 34, 58] compresses the model weights into lower precision after training to reduce the memory footprint during inference. SparTA [62] exploits model sparsity to accelerate computation. However, these approaches sacrifice the model accuracy. vLLM [31] proposes PagedAttention to reduce the GPU memory fragmentation. This is orthogonal to this paper and FastServe implements PagedAttention as well.

8 Conclusion

We present FastServe, a distributed inference serving system for LLMs. We exploit the autoregressive pattern of LLM inference to enable iteration-level preemption and design a novel skip-join MLFQ scheduler to address head-of-line blocking problem. We propose a proactive key-value cache management mechanism to handle the memory overhead of the key-value cache and hide the data transmission latency with computing. Based on these, we build a prototype of FastServe. Experiments show that FastServe improves the throughput by up to 6.1× under the same latency SLO compared to vLLM.

Acknowledgments. We sincerely thank our shepherd, Anand Iyer, and the anonymous reviewers for their valuable feedback. This work was supported by the National Key Research and Development Program of China under Grant 2022YFB4500700, the Scientific Research Innovation Capability Support Project for Young Faculty under Grant ZYGXQNJSKYCXNLZCXM-I1, the Fundamental Research Funds for the Central Universities, Peking University, and the National Natural Science Foundation of China under the grant numbers 62172008, 62325201 and 624B2003. Xin Jin is the corresponding author. Bingyang Wu, Yinmin Zhong, Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022.

[2] Introducing Qwen. https://qwenlm.github.io/blog/qwen/, 2023.

[3] Reinventing search with a new ai-powered bing and edge, your copilot for the web. https://news.microsoft.com/the-new-Bing/, 2023.

[4] Sharegpt teams. https://sharegpt.com/, 2023.

[5] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *USENIX OSDI*, 2024.

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv*, 2023.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *SIGCOMM CCR*, 2013.

[8] Anthropic. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family, 2024.

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *USENIX OSDI*, 2015.

[10] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel dnn training. In *ACM SOSP*, 2021.

[11] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *USENIX OSDI*, 2020.

[12] 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. *arXiv*, 2020.

[13] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *SIGCOMM CCR*, 2015.

[14] NVIDIA Corporation. Fastertransformer, 2019.

[15] NVIDIA Corporation. Triton inference server: An optimized cloud and edge inferencing solution., 2019.

[16] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *USENIX NSDI*, 2017.

[17] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying

Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[18] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv*, 2022.

[19] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *ACM PPoPP*, 2021.

[20] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv*, 2022.

[21] Google. Our next-generation model: Gemini 1.5. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/, 2024.

[22] 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, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt

Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024.

[23] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *USENIX NSDI*, 2019.

[24] 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 *USENIX OSDI*, 2020.

[25] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *USENIX OSDI*, 2022.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[27] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, 2012.

[28] 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. *Neural Information Processing Systems*, 2019.

[29] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX NSDI*, 2019.

[30] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv*, 2020.

[31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *ACM SOSP*, 2023.

[32] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P. Xing, and Hao Zhang. Mpcformer: fast, performant and private transformer inference with mpc. *arXiv*, 2023.

[33] Dacheng Li*, Rulin Shao*, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. How long can open-source llms truly promise on context length?, 2023.

[34] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning (ICML)*, 2020.

[35] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *USENIX OSDI*, 2023.

[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 *USENIX OSDI*, 2018.

[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 *ACM SOSP*, 2019.

[38] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. *arXiv*, 2021.

[39] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv*, 2017.

[40] OpenAI, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael

Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024.

[41] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv*, 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 *ACM/IEEE ISCA*, 2024.

[43] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving. In *ACM Symposium on Cloud Computing*, 2024.

[44] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *ACM SOSP*, 2017.

[45] Linus Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 1968.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[47] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[48] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv*, 2019.

[49] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *USENIX OSDI*, 2024.

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv*, 2020.

[51] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *USENIX OSDI*, 2024.

[52] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[53] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv*, 2023.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Neural Information Processing Systems*, 2017.

[55] Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. Fine-tuning language models over slow networks using activation quantization with guarantees. *Neural Information Processing Systems*, 2022.

[56] 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. Huggingface's transformers: State-of-the-art natural language processing. *arXiv*, 2020.

[57] 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. *ACM SOSP*, 2024.

[58] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *International Conference on Machine Learning*, 2022.

[59] 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 *USENIX OSDI*, 2022.

[60] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. In *USENIX NSDI*, 2023.

[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. *arXiv*, 2022.

[62] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. SparTA: Deep-Learning model sparsity via Tensor-with-Sparsity-Attribute. In *USENIX OSDI*, 2022.

[63] 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 *USENIX OSDI*, 2024.