Multiplexed Heterogeneous LLM Serving via Stage-Aligned Parallelism

Tao Luo
University of Pennsylvania
Philadelphia, PA, USA
taoluo71@seas.upenn.edu

Kelvin K. W. Ng
University of Pennsylvania
Philadelphia, PA, USA
kelvinng@seas.upenn.edu

Zhen Ping Khor
University of Pennsylvania
Philadelphia, PA, USA
zpkhor@seas.upenn.edu

Sidharth Sankhe
University of Pennsylvania
Philadelphia, PA, USA
sankhe@seas.upenn.edu

Boon Thau Loo
University of Pennsylvania
Philadelphia, PA, USA
boonloo@seas.upenn.edu

Vincent Liu
University of Pennsylvania
Philadelphia, PA, USA
liuv@seas.upenn.edu

Abstract

Modern LLM serving workloads are increasingly heterogeneous, involving a growing portfolio of models with vastly different compute and memory requirements. Existing approaches to model serving—ranging from static GPU partitioning to dynamic reconfiguration and GPU multiplexing—fail to effectively support heterogeneity.

In this paper, we introduce ParaFlex, a new system for serving heterogeneous LLMs that breaks from the conventional assumption of a single shared pipeline structure across models. Rather than aligning the layout of model pipelines, ParaFlex aligns the execution times of their stages, allowing models of different sizes and parallelism configurations to coexist efficiently on shared GPUs. Our design achieves high GPU utilization, high throughput, and low tail latency under bursty, mixed-model workloads.

CCS Concepts

• **Computer systems organization → Neural networks**; • **Software and its engineering → Massively parallel systems**.

Keywords

Heterogeneous multi-model LLM serving, Pipeline parallelism, Head-of-Line blocking

ACM Reference Format:
Tao Luo, Kelvin K. W. Ng, Zhen Ping Khor, Sidharth Sankhe, Boon Thau Loo, and Vincent Liu. 2025. Multiplexed Heterogeneous LLM Serving via Stage-Aligned Parallelism. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772207

1 Introduction

As demand for Large Language Models (LLMs) grows, so too does the cost of serving them in production. LLM inference is compute- and memory-intensive, often requiring multiple high-end GPUs per request arranged in a parallelized 'execution group.' At the same time, organizations and cloud providers must now support a growing portfolio of models—ranging from lightweight, low-cost variants to massive foundation models—to satisfy different functionality, latency, cost, and performance requirements.

In this paper, we argue that future LLM serving systems must treat heterogeneity as a first-class concern. The industry trend toward multi-model deployment is clear: LLM API providers already provide standard APIs around various heterogeneous LLMs, e.g., Azure's offering spans from GPT-4o to GPT-4o-mini [19], and AWS supports Llama 3.1 405B as well as Llama 3.2 1B [18]. GPT-5's architecture now bakes in automated routing between differently sized models [4].

Today, strategies for hosting multiple models can broadly be classified into three approaches. By far, the simplest method is to take the set of available GPUs and partition them so that each model is hosted on its own dedicated set of GPUs. However, the burstiness of today's inference workloads means that any static provisioning is necessarily inefficient, leading to the underutilization of costly hardware resources [7, 14].

A second approach is to continue to partition the cluster into distinct execution groups but to remove and recreate execution groups for models as demand changes over time [9, 28]. This approach affords some flexibility in resource allocation; however, reinitializing an execution engine involves multiple operations that can take tens of seconds in total [9]. Under dynamic and bursty workloads, this approach can incur a big context switch overhead. Additional complexity occurs if the old and new execution groups need different numbers of GPUs, as they may be used to host different-sized models, necessitating complex coordination across execution groups.

A recent line of work explores a promising alternative: *multiplexing GPUs* among multiple models, with either all available models sharing all available GPUs in a single execution group or with some degree of partitioning (i.e., n execution groups, with all models and GPUs cleanly divided among the n groups) [7, 14, 29]. Multiplexing GPUs improves utilization, as it enables a single model's bursts to leverage a larger pool of resources while other models are idle.

Unfortunately, existing multiplexed LLM serving frameworks all assume models are identical or similarly sized—if they allow multi-model serving at all. For instance, these systems all enforce a single shared pipeline structure and parallelism configuration. To see why this might be problematic, consider a case where an

SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772207

API provider hosts two LLMs: one small and one large. Assume the provider wants to split the small model's layers into a small number of pipeline stages, p. The only way to share those resources with the large model is to stuff the large model into the same number of stages on the same set of GPUs.

In the resulting configuration, the large model will inevitably occupy the majority of each GPU's computation and memory resources. Most likely, it will either exceed the available GPU memory capacity or leave an inordinately small amount of space for KV cache. Going the other direction and aligning the small model's GPU-level footprint to that of the large model results in parallelism configurations with a very small amount of computation per GPU and, thus, large overheads (as a fraction of total execution time). Regardless of the alignment strategy, the small model suffers from Head-of-Line (HoL) blocking when its requests are delayed behind those of the large model. A naïve solution is to preempt the large model; however, this can lead to starvation, as the small model's autoregressive decoding may monopolize resources and prevent the large model from making progress.

In this paper, we present a novel strategy for sharding and placing heterogeneous models on shared GPUs that avoids the above effects. The key design principle is that, rather than forcing entire models to align their GPU-level footprints, we instead focus on aligning the approximate execution times of their pipeline stages. In the above example, this would mean splitting the large model into $p' > p$ stages, such that all stage execution times and memory footprints are approximately equal. Doing so eliminates HoL blocking and balances memory and computation usage among models.

While conceptually simple, breaking the requirement of a single shared pipeline structure and supporting heterogeneous multi-model serving requires fundamental changes to current systems' execution models and a rethinking of every step in the model-serving workflow. To that end, we present ParaFlex, the first system to allow for truly flexible model sharding and GPU multiplexing. ParaFlex is built around the abstraction of *autonomous execution engines* that can host stages of any model (regardless of whether their parallelism configurations align) and have full autonomy to schedule, delay, and swap requests of those models.

Building ParaFlex required overcoming several challenges: How do we enable autonomous engine execution? How do we support efficient coordination between engines? How do we place LLMs with different GPU-level footprints on a shared set of GPUs? How do we schedule, dispatch, and manage those heterogeneous models?

ParaFlex achieves flexibility by implementing stage-level buffers and a decentralized, non-blocking tensor forwarding channel between engines. It partitions the models into stages using their execution time and places them in a ring to ease coordination issues, all while balancing usable KV cache size on each GPU. Finally, when scheduling requests and managing KV cache occupancy, it leverages a multi-model batching-aware strategy that executes/holds requests to minimize latency while maximizing throughput, fairness, and compute efficiency. In our evaluation, ParaFlex consistently outperforms prior systems across a range of workloads and model configurations. It reduces median latency by up to 63% and increases throughput by up to 4.7× compared to tensor-parallel multiplexing. Compared to shared pipeline-parallel systems, it improves throughput by up to 1.6× and avoids HoL-induced latency coupling across models. Our contributions include:

- We propose a novel strategy for sharding and placing heterogeneous LLMs centered around the alignment of pipeline-parallel stage lengths.
- We design a novel LLM serving architecture built on autonomous execution engines that enables the flexibility necessary for the above strategy.
- We introduce a set of runtime policies—scheduling, dispatch, KV cache management—that complement the above and optimize specifically for heterogeneous serving.
- We implement and evaluate ParaFlex on both a hardware testbed and cluster-scale simulator, demonstrating 1.6× throughput over state-of-the-art systems AlpaServe/MuxServe, while maintaining comparable median latency.

2 Background

At a high level, today's LLMs consist of a stack of transformer blocks [26], each containing attention heads and feedforward components. An input layer takes in a tokenized input sequence, while an output layer turns the output from the last block into a probability distribution and samples the next output token from it.

Inference has two phases: prefill that examines all prompt tokens to generate the first output token, compute-intensive, and decode that autoregressively generates remaining tokens. Decode time dominates end-to-end latency. An optimization is to store and reuse the model's activations from the prefill and previous token generation steps using a data structure called a KV cache. With this cache, each step in the decode phase only needs to compute the KV cache for the last generated token. For the remaining decode work, systems rely heavily on batching to amortize the cost of loading the model weight and increase compute utilization. The amount of GPU memory remaining after model weights determines the KV cache capacity and, thus, is a major bottleneck to batching and inference throughput [12].

2.1 Today's Multi-LLM Workloads

While the LLM era started with a handful of organizations each training their own monolithic 'foundation model,' today's LLM API providers typically need to host many models of different sizes to satisfy different performance and functionality requirements.

For example, most LLM API providers supply a range of model offerings of different sizes and generations (e.g., OpenAI currently serves from GPT-3.5 Turbo to GPT-4.1, 4.1 mini, and 4.1 nano, with GPT-5 incorporating automated routing between legacy models and newer ones [4]).

Real-world workloads are long-tailed across all dimensions. Model demand is skewed—few popular models handle most requests. OpenRouter data [1] (collected on March-31-2025) [20] shows model popularity follows a Zipf distribution $s = 1.01$.

Within a single model, the arrival pattern of requests and their request/response lengths also follow long-tailed distributions. For example, the Azure ChatGPT conversation trace [24] exhibits a request inter-arrival time that follows a fairly bursty log-normal distribution with shape parameter $s = 2.24$. Looking at the token

[1] OpenRouter is a model aggregator that provides a unified interface for 300+ models across 50+ providers, handling ~5.6 trillion tokens per month.

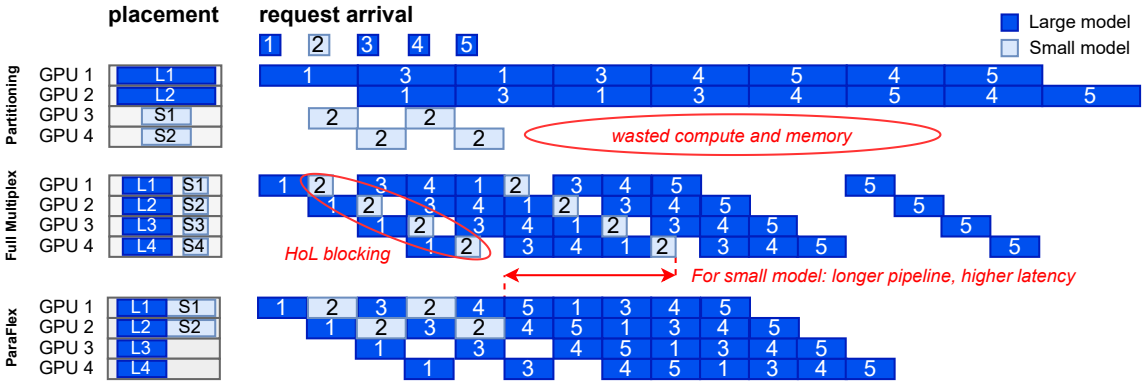

Figure 1: Five requests arrive sequentially, each request has one prefill and one decode iteration. The resource partition approach (top) leaves GPU idle. The shared pipeline approach (middle) causes HoL blocking and inflated latencies for small models. ParaFlex's approach (bottom) achieves good balance between latency and resource utilization.

length of the same traces, we find that in terms of response length—typically the dominant factor in latency—90pct have 59 tokens on average, while the remaining 10pct have 478; request length follows a similar distribution.

2.2 Distributing LLM Inference

Data parallelism simply replicates model instances. However, model size and memory capacity limits often require model parallelism.

Intra-layer parallelism. Intra-layer parallelism encompasses techniques like tensor parallelism (which splits transformer blocks symmetrically among multiple GPUs and has each GPU working on the same block simultaneously), as well as expert parallelism (which distributes the experts of an MoE model across GPUs), among others. These approaches improve the latency and per-GPU memory footprint of the model but involve costly collective communication to distribute and aggregate results, limiting their maximum scalability [17]. For instance, tensor parallelism is typically constrained to the extent of the scale-up GPU interconnect (e.g., NVLink) within a single server [8, 15].

Inter-layer parallelism. Inter-layer, or pipeline-parallel execution, divides the model across multiple stages, each containing a consecutive set of transformer blocks [2, 5]. Here, a sequence of pipeline stages processes incoming requests (i.e., for prefill, decode, or a chunked prefill), with each stage handling a single request at a time but different stages potentially executing in parallel.

The communication overhead of pipeline parallelism is relatively low, making it suitable for cross-machine GPUs where the bandwidth is lower than intra-node; for large models, it can achieve almost linear scalability [31]. Because the performance is less constrained by hardware and the configuration is more flexible, the optimal pipeline parallelism degree depends on the model and workloads involved [14, 30]. In fact, prior work [7, 14] showed that parallelizing a single model beyond its own resource demands can facilitate resource sharing and improve the utilization. The choice afforded by pipeline parallelism is the primary challenge tackled by ParaFlex's design; we leave an exploration of multiplex heterogeneous serving for other parallelism methods to future work.

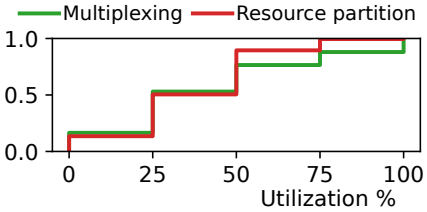

Figure 2: Partitioned model placement has lower GPU utilization than multiplexing (request rate=2).

2.3 Multiplexing Heterogeneous LLMs

While parallelism is relatively well understood in the context of single-model serving, applying it to multiplexed model serving is challenging, especially when the multiplexed models are heterogeneous. In particular, the straightforward extension of the above parallelism methods means that current systems force multiplexed models to precisely align their pipeline stage counts and placements so that they can treat all requests/stages of different models as if they were all from a single model.

Consider a small and large model on four GPUs with pipeline parallelism (illustrated Figure 1). Full partitioning causes underutilization under bursty workloads, while forced sharing creates HoL blocking and latency inflation.

HoL blocking effects can potentially be incurred during prefill and in every decode iteration.

We can quantify these costs empirically. Specifically, we deploy a large model (CodeLlama-34b) alongside two small models (Phi-2) on four GPUs, using the experimental methodology and realistic workload of §8.1. The resource partitioning approach assigns one GPU to each Phi-2 model and two GPUs to CodeLlama-34b with two pipeline stages. The multiplexing approach shards all models into four pipeline stages deployed to each of the four GPUs. Figure 2 shows the limits of partitioning in fully utilizing the cluster. Figure 3 shows that, as load increases, multiplexing achieves higher throughput but at the cost of median latency (tail latency is impacted primarily by the peak throughput of the system).

While hybrid solutions (that combine partitioning and sharing) are feasible, in the end, sacrifices inevitably need to be made in the service of model parallelism alignment.

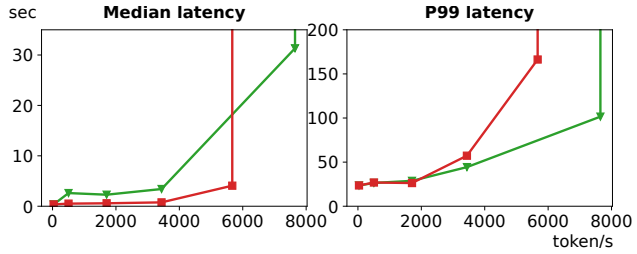

Figure 3: As the load increases, fully multiplexing has higher median latency and throughput than resource partitioning.

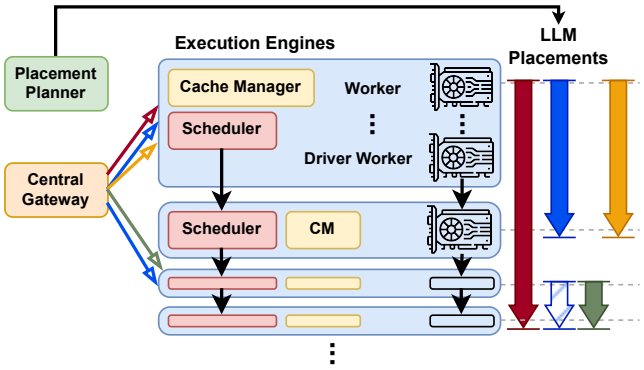

Figure 4: The system architecture of ParaFlex. As an example, we show four execution engines (top to bottom) and the placement for four LLMs, as annotated by arrows indicating the flow of pipeline parallelism. The LLMs are sharded into 4, 2, 2, and 2 stages (left to right), with the blue model replicated once as striped blue.

3 Design Overview

Multiplexed LLM serving requires rethinking inter-layer parallelism. Rather than forcing models to align pipeline configurations or eschewing multiplexing benefits, ParaFlex equalizes execution times and resource requirements across stages.

Our thesis is that doing so presents outsized benefits to efficiency, utilization, and thus, throughput and latency.

System architecture. Figure 4 illustrates the overall architecture of ParaFlex, encompassing four types of components:

(1) A *placement planner* that shards, replicates, and places LLMs onto execution engines based on workload distributions.
(2) A *central gateway* routing requests and streaming responses.
(3) A collection of *execution engines* managing per-stage execution and inter-stage communication.
(4) A set of *workers* per-GPU executing model stage.

ParaFlex's support for unaligned pipelines allows the planner to (1) partition LLMs into approximately evenly sized stages, i.e., smaller models into fewer stages and larger models into more stages, and (2) place the resulting stages across execution engines in a way that balances both expected load and GPU memory utilization.

During execution, requests flow through the system much like they would flow through a microservice workflow. New requests are forwarded to the execution engine responsible for the target instance's first stage. Each execution engine contains a local scheduler that determines the next model stage to execute based on the queued requests. The scheduler can also manage which KV cache to swap in or out and opportunistically batch requests.

The engine submits the scheduled task to the local workers for stage execution. Once the workers finish executing, the engine sends the intermediate tensor and metadata to the model's downstream execution engine asynchronously. The send and receive kernels execute concurrently to the actual stage executions.

The benefits of flexibility and independence. Additional model configuration flexibility and execution engine independence enable a markedly broader set of policies for everything from sharding and placement to scheduling and batching. ParaFlex, exercising this flexibility judiciously, addresses the HoL problem while maintaining the multiplexing effectiveness, achieving a better resource sharing and interference tradeoff.

4 The ParaFlex Planner

We begin with a description of how ParaFlex balances per-stage LLM execution time through partitioning and placement before covering how it enables and uses that placement in subsequent sections (§6 and §5). This planning phase occurs once at cluster-initialization time, but in principle, it can be combined with model-swapping to reconfigure for longer-term shifts in workload distribution.

4.1 Per-Model Sharding

ParaFlex targets mainstream production LLMs with a uniform layer structure, such as decoder-only models, and assumes uniform intra-layer parallelism across all execution engines, as described in §2.2. Specifically, we expect users to continue to use existing methods of sharding within layers, e.g., setting TP size based on the connectivity of the scale-up interconnect. Note that this assumption of uniform intra-layer parallelism is not fundamental; it is made for implementation simplicity and is orthogonal to the core problem.

ParaFlex instead focuses on inter-layer sharding, where, as mentioned in §3, its goal is to align the execution time of the final partitioned stages. To that end, ParaFlex includes a T_s = *target stage size* parameter that defines the approximate quantum into which models should be split. By default, T_s is equal to the execution time of the smallest model (accounting for intra-layer parallelism), but multiples or sub-multiples are also allowed (e.g., ×2, ÷2, ÷4, etc.). Lower T_s trades off per-stage overhead for multiplexing efficiency. The sweet spot depends on the hardware setup and workload. As a rule of thumb, operators can tune T_s by profiling all models with a set of T_s and choose the lowest T_s that satisfies SLOs.

A model's total number of stages, S, is calculated by dividing the model's execution time by T_s and rounding to the nearest integer. The model's L layers are then divided as evenly as possible among the S stages by assigning $\lfloor \frac{L}{S} \rfloor$ to each stage and allocating any remaining layers one at a time, beginning from the first stage and moving forward. This is based on the observation that per-token latency remains largely constant across typical context lengths observed in existing traces [2]. For extremely long, outlier context lengths, ParaFlex follows the common industry practice of dispatching these requests to separate, dedicated pools [3].

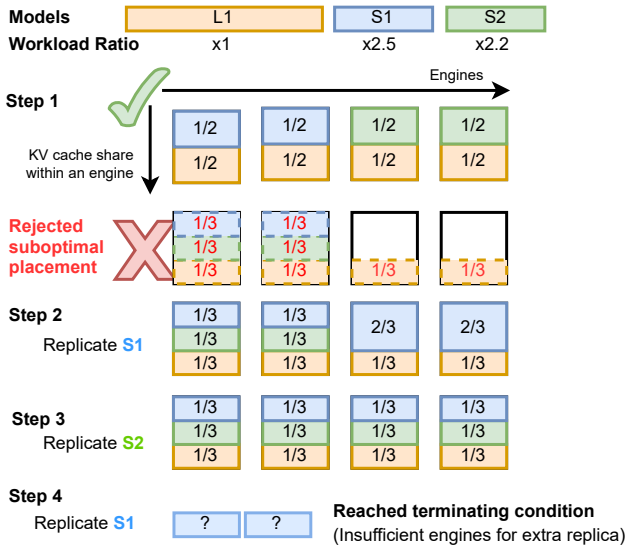

Figure 5: Replication and placement procedure for 1 large model and 2 small models with workload ratio of 1 : 2.5 : 2.2.

4.2 Replication and Placement

Once the models are sharded into similarly sized stages, ParaFlex needs to decide how to replicate and place those models. We formalize the joint replication/placement problem as an optimization that extends classical DRF by incorporating discrete placement decisions and joint co-optimization of replication, placement, and fairness.

Sets and parameters.

- $\mathcal{M} = \{1, \ldots, N_m\}$: set of models
- $\mathcal{E} = \{1, \ldots, N_e\}$: set of execution engines
- $s_i \in \mathbb{Z}^+$: number of pipeline stages for model i
- $R_i \in \mathbb{R}^+$: workload ratio (normalized arrival rate) for model i, where $\sum_{i \in \mathcal{M}} R_i = 1$
- $M \in \mathbb{R}^+$: total memory capacity per engine (assumed uniform)
- $w_i \in \mathbb{R}^+$: total weight size of model i
- $\kappa \in \mathbb{R}^+$: minimum KV cache size per stage

Decision variables.

- $r_i \in \mathbb{Z}^+$: number of replicas for model i
- $p_{i,j} \in \{1, \ldots, N_e - s_i + 1\}$: starting engine ID for replica j of model i, $j \in \{1, \ldots, r_i\}$

Derived quantities from placement.

- Engine assignment: Stage k of replica j of model i is placed on engine $e_{i,j,k} = p_{i,j} + k - 1$, where stages are placed sequentially on consecutive engines in ascending order
- Weight per stage: $w_{i,k} = w_i / s_i$ (approximately, due to even sharding)
- Available shared KV capacity on engine e:

$$C_e = M - \sum_{\substack{i,j,k: \\ e_{i,j,k}=e}} w_{i,k} \tag{1}$$

- Given a placement $(r_i, p_{i,j})$, KV allocations $\text{KV-alloc}_{i,j,k}$ are computed via DRF, treating each engine's available KV capacity C_e as a distinct resource type. We define each model's dominant share as its minimum per-stage share:

$$\phi_i = \min_{j \in \{1,\ldots,r_i\}, k \in \{1,\ldots,s_i\}} \frac{\text{KV-alloc}_{i,j,k}}{C_{e_{i,j,k}}} \tag{2}$$

Optimization objective and constraints.

$$\max_{(r_i, p_{i,j})} \Phi^*(r_i, p_{i,j}) \tag{3}$$

where $\Phi^*(r_i, p_{i,j}) = \min_{i \in \mathcal{M}} \phi_i$ is the minimum dominant share after applying DRF on the given placement.

- *Non-overlapping replicas:*

$$|p_{i,j} - p_{i,j'}| \geq s_i \quad \forall i \in \mathcal{M}, j \neq j' \tag{4}$$

- *Memory feasibility:*

$$\sum_{\substack{i,j,k: \\ e_{i,j,k}=e}} w_{i,k} \leq M \quad \forall e \in \mathcal{E} \tag{5}$$

- *Sufficient per-stage KV cache:*

$$\text{KV-alloc}_{i,j,k} \geq \kappa \quad \forall i \in \mathcal{M}, j \in \{1, \ldots, r_i\}, k \in \{1, \ldots, s_i\} \tag{6}$$

- *Resource-demand proportionality:*

$$\frac{r_i \cdot s_i}{\sum_{i' \in \mathcal{M}} r_{i'} \cdot s_{i'}} \approx \frac{R_i \cdot s_i}{\sum_{i' \in \mathcal{M}} R_{i'} \cdot s_{i'}} \quad \forall i \in \mathcal{M} \tag{7}$$

Note the formulation assumes the average arrival rate of each model is known in advance, either from historical estimation or a manual forecast. If the arrival rate changes over time, our algorithm can also be extended to adapt the existing placement plan by incrementally adding or removing model instances, leveraging its iterative nature. The affected execution engines can then reload the corresponding model stage weights as needed.

Algorithm overview. Given the optimization problem, ParaFlex employs a unified algorithm for both placement and replication decisions, called iteratively.

The algorithm contains two components: a *replication planner* and a *placement optimizer*. In each step, the replication planner iteratively updates the current replication plan with an additional replica of a chosen model. The placement optimizer then takes this updated replication plan and optimizes its placement across the available execution engines. As illustrated in Figure 5, by iteratively calling two components, models are incrementally replicated and placed until either the replication planner or placement optimizer violates constraints. The complexity of this algorithm is $O(N_m^2 \cdot N_e^2)$, where N_m is the number of models and N_e is the number of engines.

Replication planner (Algorithm 1). The replication planner iteratively adds model replicas while ensuring that allocated resources align proportionally with demand among models. We quantify a model's relative demand using a *resource demand ratio*, calculated by multiplying a model's normalized arrival rate by its relative size. This is the target resource ratio for the replication planner.

Conversely, as the placement optimizer §4.2 allocates an equal share of resources to each stage, the actual *allocated resource ratio* is determined by the number of replicas multiplied by each model's PP size. ParaFlex's goal is to provision resources for the models in proportion to this resource demand ratio. In each iteration, the replication planner minimizes the difference between the allocated and target resource demand ratios by duplicating the model with

the least relative resources. The algorithm terminates when the chosen model runs out of available engines.

Placement optimizer (Algorithm 2). In the same step, the placement optimizer receives the new replication plan and generates an optimized placement. Traditional distributed systems often rely on random placement. In contrast, we optimize placement based on two design principles:

P1 Keep model execution flowing in the same direction.
P2 Focus on maximizing the total usable KV cache share while enforcing fairness.

(P1) Maintaining pipeline directionality between models: If any two models have overlapping execution engines, this principle requires that the models' overlapping parts should follow the same engine execution order. This heuristic borrows from classic work on flow/jobshop scheduling that has shown that same-order placement of sequential computation graphs has better latency than purely random placement, regardless of the scheduling algorithm [23].

Our benchmarks showed similar results for execution time and memory use. Intuitively, one iteration's pipelined execution acts like a request moving through multiple engines. Under high contention, random placement causes collisions and delays, while same-order placement limits collisions to request ingress.

(P2) Maximizing fair share of the KV Cache: ParaFlex's other objective is to maximize total usable KV cache share while ensuring fairness among models. Because KV cache capacity is a major bottleneck of LLM serving throughput [12], it is critical to ensure a fair and efficient allocation of KV caches among all deployed models across all execution engines.

The placement algorithm leverages dominant resource fairness (DRF) [10] as a primitive in a non-traditional way. It also generalizes DRF by incorporating memory fungibility across engines as an additional dimension for optimization.

DRF is typically applied to heterogeneous resource allocation, e.g., allocating CPUs and memories. However, we can apply DRF if every engine's KV cache capacity is treated as *unique types of resources*. Assume the placement of model stages on execution engines is known, and all models have infinite demand. Because of the even inter-layer parallelism sharding of ParaFlex, the KV cache demand remains balanced across all resident stages, making all resident engines' cache space the dominant resource. Note that this fixed KV cache share is used only conceptually for offline planning; ParaFlex's KV allocation at runtime is dynamic and flexible.

The DRF takes a placement plan as input to determine each model's estimated fair share. Intuitively, the fair share is both a measure of the fairness of the model as well as the relative balance of post-weight KV cache capacity, with more even allocations better on both axes (see Figure 6 for an example). Specifically, we define the target metric as the minimum stage-wise KV cache share across all models based on their memory share across all engines. We enumerate all possible placement plans and choose the placement that has the highest target metric.

Placement workflow: Model replicas are placed incrementally in length order, beginning with the longest. For each replica, we enumerate all possible engine placement options and choose the one that maximizes the target metric under DRF policy. In the case of ties, we choose the one with the lowest first-stage engine ID.

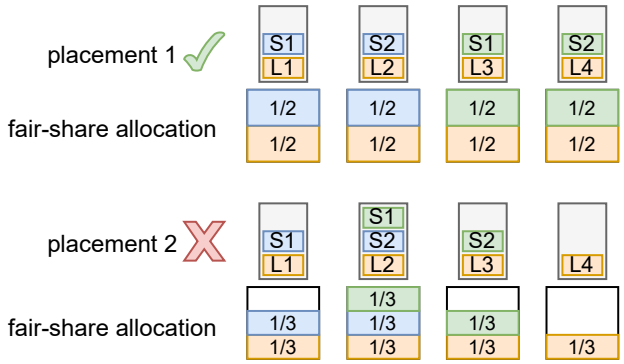

Figure 6: Different placements lead to different KV cache share by DRF; some placements are better than others.

Algorithm 1 Choose next model for replication

Input:
N: total number of engines to place
$s = \langle s_1, s_2, ..., s_n \rangle$: number of stages for each model
$r = \langle r_1, r_2, ..., r_n \rangle$: current number of replicas of each model
$R = \langle R_1, R_2, ..., R_n \rangle$: workload ratio of each model
Output: i: model chosen to be replicated
$U \leftarrow \{i | r_i = 0\}$ ▷ unallocated models
$\hat{R} \leftarrow \langle R_1 s_1, R_2 s_2, ..., R_n s_n \rangle$ ▷ target resource ratio
if $|U| > 0$ **then** ▷ allocated most resource heavy model first
  return $\arg\max_{m \in U} \hat{R}_m$
$\hat{r} \leftarrow \langle r_1 s_1, r_2 s_2, ..., r_n s_n \rangle$ ▷ actual resource ratio
$\tilde{r} = \langle \frac{\hat{r}_1}{R_1}, \frac{\hat{r}_2}{R_2}, ..., \frac{\hat{r}_n}{R_n} \rangle$ ▷ actual:target relative resource ratio
a $\leftarrow \min \tilde{r}$ ▷ least relative resource ratio
$i \leftarrow \arg\max_{i \in [1,n]} \{\hat{R}_i | \tilde{r}_i = a\}$
if $r_i s_i > N$ **then** ▷ abort when exceed total engines
  return -1
return i

Algorithm 2 Placement algorithm

Input:
N: total number of engines to place
M: total memory capacity per engine
$w = \langle w_1, w_2, ..., w_n \rangle$: model weight size of each model
$s = \langle s_1, s_2, ..., s_n \rangle$: number of stages for each model
$R = \langle R_1, R_2, ..., R_n \rangle$: workload ratio of each model
k: minimum KV cache size per stage per replica
Output: $p = \langle p_{1,1}, p_{1,2}, ..., p_{n,m} \rangle$: starting engine IDs for each model and replica
$r \leftarrow \langle 1, 1, ... \rangle$ ▷ number of replicas of each model
while true do
  $p \leftarrow \langle -1, -1, ... \rangle$ ▷ -1 means placement still unknown
  for each model i in descending order of number of stages **do**
    for each replica j **do**
      $e \leftarrow -1$ ▷ track starting engine
      $m \leftarrow -1$ ▷ track minimum per-stage KV allocation among all models
      for each engine k **do** ▷ enumerate all possible engines
        $p_{i,j} \leftarrow k$
        ▷ DRF_on_KV returns stagewise KV allocation for each model by DRF algorithm after deducting the model weight size
        drf_mem_min_alloc $\leftarrow$ min(DRF_on_KV(p, w))
        if drf_mem_min_alloc $> m$ **then** ▷ update when improves
          $m \leftarrow$ drf_mem_min_alloc
          $e \leftarrow k$
      if drf_mem_min_alloc $< c$ **then** ▷ exceed threshold
        return p' ▷ return last iteration placement
      $p_{i,j} \leftarrow e$
  $i \leftarrow$ choose_model(s, r, R) ▷ Algorithm 1
  if $i = -1$ **then**
    return p
  $r_i \leftarrow r_i + 1$
  $p' \leftarrow p$

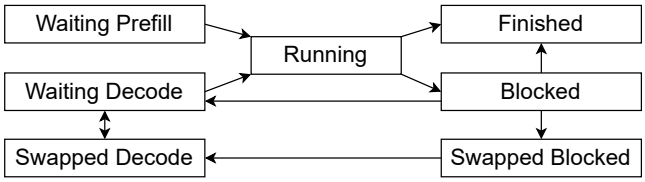

Figure 7: Request scheduling state diagram.

5 Request Dispatch and Scheduling

The central gateway load balances by dispatching incoming requests to the replica with the fewest outstanding requests. Once dispatched, each stage decides locally what to execute next.

Request processing state diagram (Figure 7). In contrast to single-model systems where simple scheduling policies like 'prefill first' are sufficient for maintaining high compute efficiency, ParaFlex needs to be more careful with which requests it chooses to start processing and when. To that end, ParaFlex's execution engines bucket requests into seven possible states based on its current status within the stage:

- *Waiting prefill:* the request is ready to execute the stage of its current prefill iteration on that engine
- *Waiting decode:* the request is ready to execute the stage of its current decode iteration on that engine
- *Swapped decode:* same as the waiting decode state, but its KV cache has been swapped out.
- *Blocked:* the request has finished executing its local stage and is waiting for the upstream stage's output.
- *Swapped blocked:* same as the blocked state, but its KV cache has been swapped out.
- *Finished:* the request has finished generating its last token, and its KV cache is freed.

Swapping policy. Vanilla vLLM has two options when the KV cache runs out of memory: recompute or swap to host memory. In line with the key design principles of the system, ParaFlex extends this to distributed execution by allowing each engine to independently determine when to swap in and out the KV cache between GPU and host memory. ParaFlex disables recomputation and purposely avoids coordinated global caching decisions as these incur inefficient synchronization and, depending on the policies, may trigger preemptive swapping or thrashing for other stages.

The swapping policy tries to keep the fair allocation of KV cache among all running models on the engine by picking the model stages with the largest KV cache occupancy when eviction is necessary. Within that model stage's requests, it prioritizes swapping out blocked requests over waiting decode requests and newer requests over old requests.

Engine-local scheduling policy. Engine schedulers first try to schedule max-sized decode batches, followed by prefill requests. If neither exists or is schedulable (e.g., due to KV cache limitations), ParaFlex will proceed to schedule an undersized decode batch.

In all three cases, the scheduler will try to schedule requests in ascending order of arrival time. It will try to either continuously batch requests if the engine is the first stage of the model or otherwise schedule request batches as batched by the first engine. If a request/batch cannot be allocated with enough KV cache, it will evict victims by swapping, starting from the models with the highest KV cache allocation.

6 Autonomous Execution Engine Design

ParaFlex's distributed architecture is composed of multiple independent execution engines. Unlike existing pipeline implementations that couple the engine to a single pipeline [2], ParaFlex enables model-specific pipelines composed from multiple engines.

6.1 Execution Engine Architecture

ParaFlex's execution engine is built on top of the vLLM [12] single-model serving system and inherits much of its per-stage execution procedures. Like in vLLM, ParaFlex execution engines manage metadata for request scheduling and oversee the KV cache. For the latter, the engines are responsible for tracking per-request block usage, identifying available memory, and dynamically managing GPU-CPU KV cache swapping. Meanwhile, one or more workers handle the actual model execution and memory management on each GPU through PyTorch or customized CUDA kernels.

During initialization, workers preallocate the KV cache, which is then split into fixed-size blocks that contain a sequential set of KV vector entries. Additional kernels facilitate operations such as block swapping and copying. ParaFlex operates on different token sizes per block by extending the original PagedAttention kernel [12]. Workers within an execution engine communicate via GPU-based process groups, e.g., AllReduce for tensor parallel intra-layer operations.

6.2 Distributed Multi-model Cache Management

A key challenge is enabling KV cache management across distributed engines and multiple models. Single-model KV cache management is a relatively straightforward affair—every token generated by the system takes a fixed amount of KV cache per transformer layer, and every layer shares the same memory management state. In distributed, multi-model KV cache management, the system must balance cache efficiency and overhead for models of varying sizes, and engines must be able to manage KV cache for their own stages.

Uniform KV cache block. While, in principle, different models may have different ideal cache block sizes, ParaFlex allocates a single physical KV cache partition per worker with a uniform physical block size across the entire cache—regardless of the model, the cache manager will assign additional memory to requests at that block granularity.

Block sizing is fundamentally a tradeoff between having a large enough block size to support GPU parallelism and keeping the size small enough not to waste memory [12].

There are a couple of reasons for this choice of uniformity. First, a uniform block size enhances the sharing of blocks among different models. Second, we note that the requirement of a large enough block size affects all models equally and in a similar fashion, whereas over-provisioned block sizes only affect extremely small models or those with short overall sequence lengths, implying that, given the choice, prioritizing a large enough block size will benefit the greatest number of models. Finally, ParaFlex's sharding strategy prevents pathological cases as it tries to balance stage sizes

across models. In practice, we can sweep the parameter space to find the optimal global block size for a given configuration. Optimizations like Jenga[29] are complementary. To accommodate different numbers of tokens in different blocks (e.g., based on each model's attention head size), the blocks are logically reshaped using `torch.Tensor.view` and supplied to the unmodified PagedAttention CUDA kernel [12]. No extra overhead is added to the GPU execution.

Multi-engine coordination. A single request's KV cache is distributed across multiple execution engines during execution. For each request, each execution engine maintains a local block table for the deployed stage. During the prefill phase, the request metadata is forwarded from upstream stages one after another; the block table is created based on the input tokens in the metadata. During decode, the last stage appends the newly generated token to the request metadata, and the new token update is then propagated into all subsequent engines, after which new blocks are allocated on each engine. If there is not enough KV cache space, engines swap in/out blocks locally between GPU/CPU memory.

When the request's last token is generated, the last stage informs other engines to free the KV cache blocks via a sequence of remote procedure calls. Similarly, the last stage can initiate KV cache recompute as well.

6.3 Cross-Engine Communication

For each model instance, we establish unique process groups among the workers where the model is deployed. These groups facilitate communication in distributed multi-GPU serving. In inter-layer parallelism, the stage engine must transfer intermediary data to the subsequent engine, including tensors and request metadata. This involves both engines calling `send()` and `receive()`, with each knowing which engine to connect with.

To that end, we note that ParaFlex's placement strategy (described in §4.2) never places two stages of the same model on the same execution engine. While allowing more flexibility might open further classes of sharding/scheduling algorithms, it may also create coordination complexities or even possible deadlocks under certain GPU resource management schemes. Thus, for each model on each ParaFlex execution engine, there is a single predecessor and successor (though an execution engine as a whole may communicate with many other engines).

For a given transfer, the upstream execution engine coordinates the data transfer in two phases to transfer metadata and intermediary tensor (see §7 for details).

Because different execution engines, particularly those managing intermediate stages of models, may execute requests in a different order than their predecessors, intermediate outputs must be queued in a buffer dictionary.

7 Implementation

ParaFlex is implemented on vLLM with 35,711 lines of Python, C++, and CUDA code, maintaining the original API while modifying distributed execution, scheduling, and memory management.

On top of the design outlined in §6.1, we present more details.

Single engine logical-to-physical block mapping. Each engine is responsible for managing the KV cache of its local transformer layers. We maintain a logical to physical address mapping via a layer-wise block table, which helps to accommodate multiple model stages with different numbers of layers. The logical blocks are addressed by both their layer index and layer-local block index. The physical blocks are addressed by the position in the physical partition, which is used as input to the paged attention kernel.

Two-phase cross-engine data transfer. The first phase is the transfer of the metadata, which includes the request ID, tensor shape, adapter parameters, etc. This transfer is between CPUs using remote procedure calls and Ray primitives. To ensure maximum concurrency and minimize blocking, all the network operations run on separate CPU threads from the main engine thread. We note that sending all the metadata for a batch of requests incurs a large serialization overhead. Thus, ParaFlex leverages incremental updates on request metadata during the decode phase.

The above remote procedure calls also serve to trigger the downstream engine's `receive()` calls for both hidden and residual state. Concurrently, the upstream engine calls `send()`. Here as well, each resident model has its own pair of CUDA streams dedicated to sending and receiving its tensors from the downstream and upstream neighbors, respectively. Only one such pair is necessary because, as mentioned above, each model's stage has a single predecessor/successor.

8 Evaluation

We evaluate ParaFlex's performance in serving heterogeneous models, its response to varying workload characteristics, its efficacy in addressing the issues in §2 and Figure 1, and the effectiveness of its constituent techniques.

8.1 Methodology

We evaluate ParaFlex using both a high-fidelity cluster-scale simulator for large-scale, cross-node results, along with a smaller-scale hardware testbed.

Our simulator uses Vidur's cost model [1], trained on real GPU profiling data. The simulated cluster has 8 hosts with 8 A100 GPUs per host (NVSwitch), connected via 200 Gbps RDMA NICs.

Following best practices, each server is configured with a single engine of TP size = 8. The simulator implementation follows the same architecture of ParaFlex as shown in Figure 4. We set the default max batch size to 64 and disabled batching for prefill.

We validate our approach at smaller scale with a single node containing 8× A100 80GB GPUs connected via NVSwitch. To emulate the simulated multi-server cluster, we assign an execution engine to each GPU with TP size = 1 and scale down the number of NVLink channels to match the RDMA bandwidth between servers.

Baselines. Our evaluation uses the following baselines[2].

[2]Note that no prior work has explicitly considered multi-model and pipeline-parallel serving (generally opting for one or the other). Thus, when implementing our baselines, we often need to fill in gaps in their design; where possible, we try our best to be faithful to the original design/implementation.

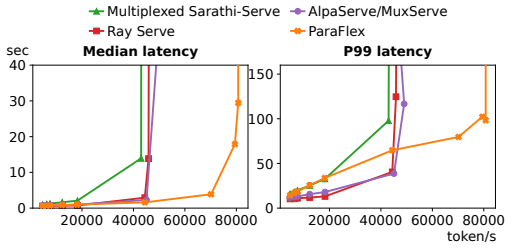

Figure 8: Base case end-to-end latency versus throughput.

Ray Serve: This baseline deploys models to dedicated GPUs, typical of traditional serving frameworks like Ray Serve and TensorRT-LLM. Because the models cannot share resources, each model's pipeline is short, we assume each model has the same PP size (default of 2) and no replication.

Multiplexed Sarathi: This pipelining baseline presents the other extreme of aligning sharding configurations by sharing all GPUs among all models (PP size 8). As in the original system, the scheduler limits the number of concurrent decode batches to the stage count.

AlpaServe [14]/MuxServe [7]:

This baseline aligns parallelism configurations but groups models by size. It creates large and small model groups, assigning resources proportional to demand rate (model size × request rate).

Multiplexed vLLM: This baseline tests the alternative of handling large models with only intra-layer parallelism. Due to GPU access limitations, this baseline is evaluated solely in the testbed[3]. There, the TP size is 8, the total GPU count.

Workload. We use the following models and request patterns by default in all experiments. We chose 1x llama-2-70b-hf, 1x codellama-34b-instruct-hf, and 2x internllm2_5-20b-chat for deployment, representing models of varying sizes. Two internllm2_5-20b-chat models represent two separate small models, named internllm2_5-20b-chat-1 and internllm2_5-20b-chat-2, respectively.

Unless otherwise specified, we assume a skewed and bursty workload based on prior work [7, 9, 14], as well as our analysis of OpenRouter's model popularity rank (§2.1). The popularity of the models adheres to a Zipf distribution with alpha=1.01, inversely proportional to their size.

As a result of this distribution, the top model accounts for 49% of the total request volume. We assume each model's requests arrive via an independent process, replayed from the trace from [24] at different rates. To extend this trace to a multi-model setting, we apply random offset on the trace for each model. We vary the seeds across various arrival rates to ensure our conclusions are robust against randomness.

8.2 ParaFlex's Performance Impact

Aggregated latency and throughput. From Figure 8, ParaFlex displays the highest throughput, whereas Multiplexed Sarathi is the lowest. ParaFlex achieves low median and P99 latency even under high load. Under low load, Ray Serve exhibits the best latency due to the absence of cross-model interference. Compared to Multiplexed Sarathi, AlpaServe/MuxServe enhances median and P99 latency by reducing cross-model interference, ParaFlex's median latency matches AlpaServe/MuxServe, while ParaFlex's P99 latency aligns with Multiplexed Sarathi. ParaFlex shows a greater median latency advantage than P99, as P99 latency is affected by the large model.

Per-model decomposition. Comparing latency across models, Figure 9 reveals a coupling effect in the shared pipeline baselines Multiplexed Sarathi and AlpaServe/MuxServe. Request latency remains consistent within a shared pipeline, regardless of model size, particularly under high load or tail latency, which is precisely the HoL blocking effect in action. ParaFlex breaks this coupling, reducing overall latency, albeit with a slight penalty for large models at medium load (due to increased probability of interference).

Effect of max batch size. We vary the scheduler's max batch size (16, 32, 64) to study its effect. For readability, we keep only the Multiplexed Sarathi baseline (others exhibit similar effects). In Figure 10, larger batch sizes enhance throughput and lower median latency for both systems by facilitating more concurrent decode requests and minimizing queuing delay. However, larger batch size also leads to slightly higher P99 latency in ParaFlex. This is because our scheduler deprioritizes partial batches in favor of full batches. Under a high batch size, requests have more chance to join a partial batch and get slowed down.

Overall, higher batches yield better performance improvements from ParaFlex over Multiplexed Sarathi.

8.3 Effect of Workload Characteristics

Model skewness. We set the Zipf distribution's $s = 2.1$ to increase the popularity skewness. This results in the most popular model receiving 74% of all requests. We also include a perfectly balanced workload where each model represents 25% of total requests. Thanks to ParaFlex's flexible replication capability, Figure 11 highlight ParaFlex's advantages from increased workload skewness compared to other baselines.

Workload with varying burstiness. The trace shows the coefficient of variance (CV) to be 12.3. We adjust workload burstiness by setting the CV of the lognormal interarrival distribution to 0.1, 3.5, and 24. Figure 12 shows high burstiness improves latency under low load but may have less benefits or even degrade it under high load. This occurs because while high burstiness facilitates batching, it results in more requests needing to be queued when the batch is full at high load. ParaFlex's median latency shows lower sensitivity to burstiness than P99. while Multiplexed Sarathi's median latency and throughput are also more sensitive. ParaFlex's insensitive median latency shows it can batch effectively without the help of workload burstiness.

Workload with varying context and output length. We adjust the context and output length by scaling the trace's original length by 0.25 and 4 times.

Figure 13 shows that longer context length results in higher ParaFlex throughput with comparable latency. Processing all input tokens in one forward pass renders the time for extra tokens negligible relative to end-to-end latency.

Varying the output length from Figure 14 shows that longer lengths result in lower throughput and higher latency due to additional slower decode iterations. As reasoning models and longer

[3]The Vidur simulator requires stage-level multi-node performance profiling, which would necessitate more GPUs than we are able to obtain.

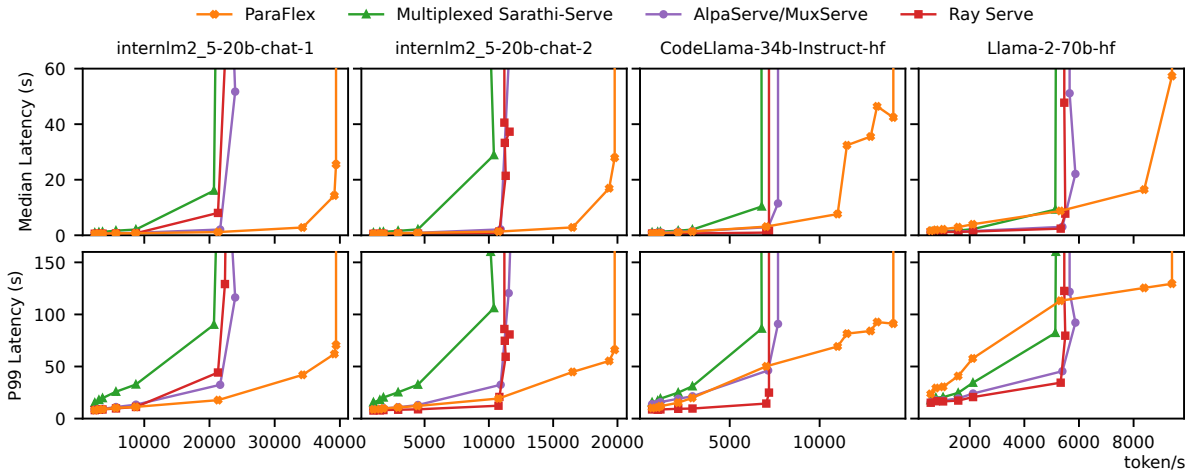

Figure 9: Per-model breakdown of end-to-end latency versus throughput in the base case.

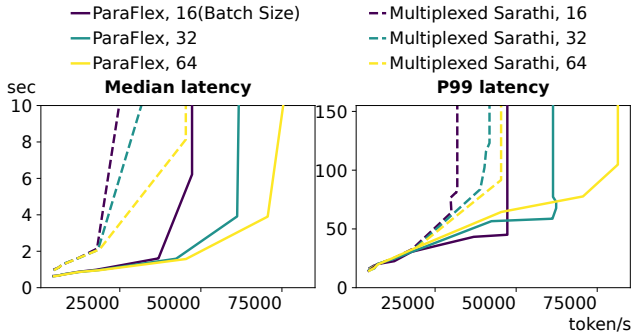

Figure 10: End-to-end latency versus throughput with varying max decode batch size.

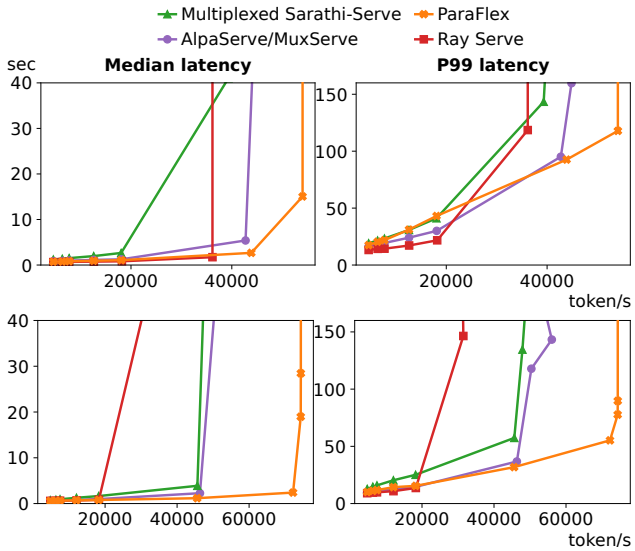

Figure 11: End-to-end latency versus throughput with balanced (top) and imbalanced (bottom) model workload.

outputs become more prevalent, ParaFlex proves increasingly effective in serving such workloads.

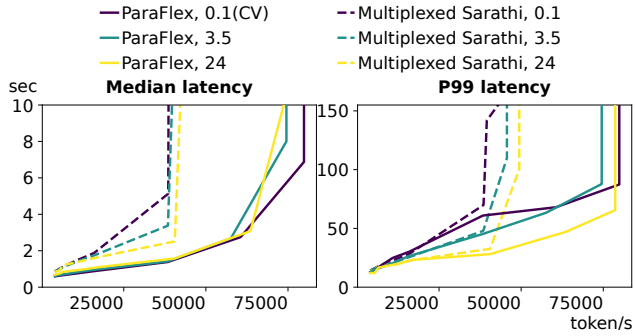

Figure 12: End-to-end latency versus throughput with varying burstiness.

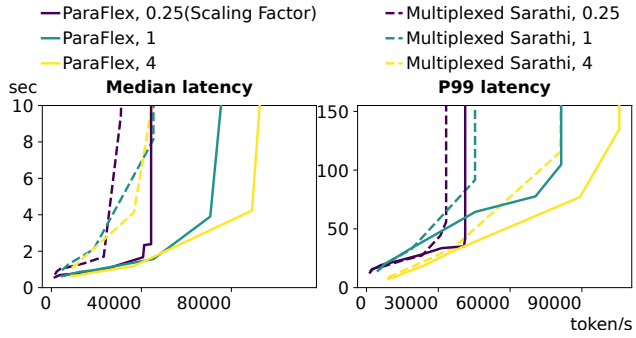

Figure 13: End-to-end latency versus throughput with scaled context length.

8.4 Microbenchmark Analysis

We benchmark heterogeneous models to assess ParaFlex in addressing head-of-line blocking and GPU utilization, in comparison to the Multiplexed Sarathi baseline.

Real models exhibit distinct characteristics (e.g., ratio of depth to width and attention architecture). To isolate model size as the only variable, we created synthetic models based on llama-7B, with 60, 120, and 240 transformer layers. We name these as llama-20B, -40B, and -80B, respectively.

We evaluate two settings with varying model size dispersions. In the first low-dispersion setting (2x dispersion), there is 1 large model

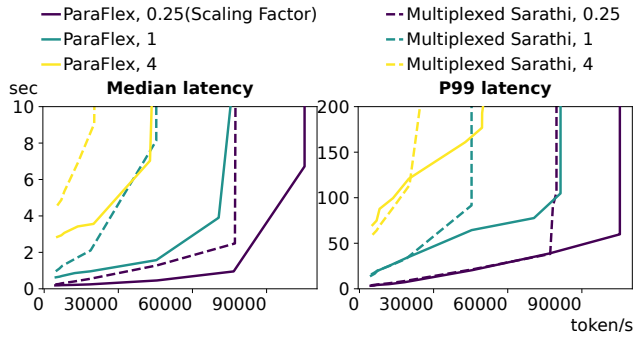

Figure 14: End-to-end latency versus throughput with scaled output length.

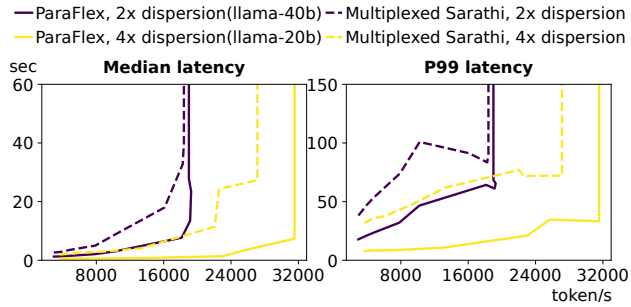

Figure 15: Llama-20B and Llama-40B end-to-end latency versus throughput with varying model size dispersion.

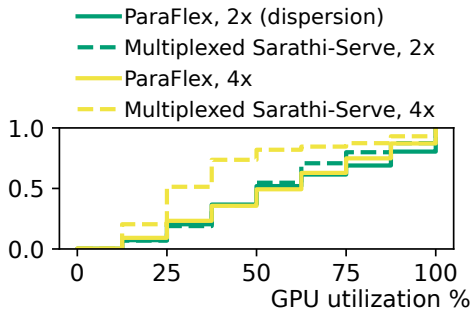

Figure 16: CDF of active GPU percentage.

(llama-80B) and 2 medium models (llama-40B). In the second high-dispersion setting (4x dispersion), there is 1 large (llama-80B) and 4 small (llama-20B) models. Identical model instances are treated as separate models, each with a uniform arrival rate.

Mitigating HoL blocking. In Figure 15 we aggregate the throughput and latency of smaller models in both settings. ParaFlex consistently demonstrates higher throughput and lower latency for smaller models. Greater model size dispersion leads to improved latency and throughput with ParaFlex. This indicates ParaFlex mitigates HoL blocking.

Increased GPU activity. We calculated the percentage of active GPUs over time. A higher percentage indicates better utilization. Figure 16 shows the active GPU percentage at system saturation—ParaFlex significantly improves GPU utilization compared to Multiplexed Sarathi. This effect increases with higher model dispersion.

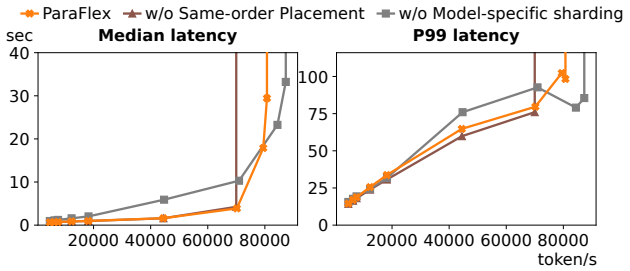

Figure 17: End-to-end latency vs. throughput with/without ParaFlex's placement and sharding optimizations.

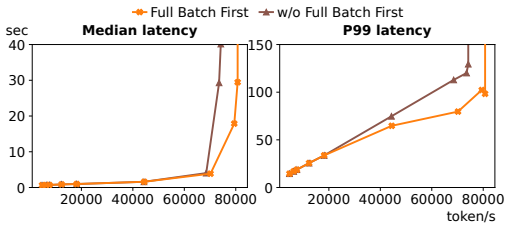

Figure 18: End-to-end latency vs. throughput with/without ParaFlex's 'full batch first' scheduling policy.

8.5 Ablation Study

We maintain the same workload as §8.2 while disabling specific techniques to showcase their effectiveness.

Sharding. A new baseline is created by disabling ParaFlex's sharding and replication mechanisms. This baseline uses a shared sharding plan and pipeline across all models while keeping the local scheduler of the execution engine. Figure 17 shows that this baseline has significantly worse median and P99 latency than ParaFlex, although with higher throughput.

Placement strategy. We compare our same-order DRF-based placement strategy to random placement of model stages. Both strategies maintain identical sharding, replication, and they avoid overlapping between replicas. The random placement ensures a uniform number of model stages across execution engines for load balancing. We run multiple times with different seeds. Figure 17 shows ParaFlex's same-order placement enhances throughput while maintaining similar median and P99 latency.

Scheduling. We compare our maximum-sized decode batch-first scheduling policy with the standard prefill-first approach.

Figure 18 shows that the full-batch-first policy effectively improves the peak throughput and reduces the tail latency. When the system is saturated, our policy executes fewer batches, demonstrating its superior batching effectiveness.

8.6 System Evaluation

We validated simulation results on a small-scale hardware testbed. The execution engine's TP size is reduced to 1. As a result, the maximum batch size is lowered to 32 to reduce swapping. The block size is set at 32 KB. Since system evaluation is slower than simulation, we also decreased the burstiness level and set CV=10 for stable performance metrics with fewer requests. As shown in §8.3, performance metrics show minimal sensitivity to burstiness. The Multiplexed vLLM was introduced as an additional baseline,

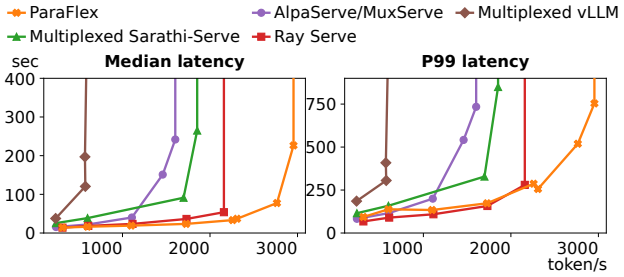

Figure 19: Median (left) and P99 (right) end-to-end latency versus throughput in system.

where all models share the same tensor-parallel group of 8 GPUs.

From Figure 19, ParaFlex achieves the highest throughput, followed by Ray Serve and shared pipeline variants. Multiplexed vLLM exhibits the lowest throughput and latency due to limited tensor-parallel scalability with low-bandwidth interconnects. Compared to the simulation, Ray Serve outperforms other baselines in throughput. This is due to the simulator's underestimation of stage overheads beyond computation and communication. Non-multiplexing, with the shortest pipeline, experiences the least overhead.

Ray Serve and ParaFlex show low median latency, while Multiplexed vLLM has the highest. Similar to the simulation, AlpaServe/-MuxServe improves median latency compared to Multiplexed Sarathi. Notably, ParaFlex's P99 latency is lower than some baselines under low load. The testbed largely aligns with the simulation results and highlights Multiplexed vLLM's limitations for multiplexing with restricted cross-GPU bandwidth.

9 Discussion

Autoscaling. While autoscaling falls outside the primary scope of our contribution, it presents distinct challenges that merit dedicated future investigation. Reconfiguration would be triggered by major workload shifts or hardware updates. Extending ParaFlex to dynamic reconfiguration requires detecting workload shifts, planning and executing new configurations, and migrating existing requests. A detailed analysis of this process is orthogonal to the primary contributions of this paper.

MoE model support. ParaFlex assumes dense models and tensor parallelism, MoE models and expert parallelism can be handled similarly after considering model sparsity.

For MoE, we expect ParaFlex's planning algorithm can treat MoE layer as *mostly* black-box and account for sparse utilization .

In particular, load imbalance between experts adds complexity, but ParaFlex can profile average active-expert loads, use conservative or worst-case stage-time estimates during planning, and potentially apply shortest-job-first (SJF) or preemptive scheduling policies at runtime to mitigate HoL at runtime. Expert balancing is an active area of research and better balance improves ParaFlex 's fit.

10 Related Work

Multiplexed LLM serving. The most relevant prior work is systems that support multiplexed LLM serving. Using the terminology of [7], we can divide proposed techniques into spatial multiplexing and temporal multiplexing.

ParaFlex uses temporal multiplexing. It could combine with spatial multiplexing approaches like GSLICE [6] and MuxServe [7], but a full exploration of the fusion is beyond our scope.

The most relevant work AlpaServe [14], forces identical parallelism configurations. ParaFlex demonstrates substantial advantages through flexible, unaligned pipelines with finer-grained sharding. Also relevant are Serverless LLM [9], Medusa [28] QLM [22]. They assume a different execution model where GPUs only load one model at a time, necessitating model weight loading from main memory or storage before serving requests for another model. In contrast, our approach maintains multiple models in GPU memory, preventing context switches in the critical path.

HoL blocking in LLM serving. We note that issues like HoL blocking also exist in some single-model contexts. Prefill iterations take longer than decode. Sarathi-serve [2] incrementally prefills in chunks. Splitwise [21], DistServe [31], DéjàVu [25], and TetriInfer [11] separate prefill and decode into different instances.

Limited KV cache space can also cause long decode requests to block subsequent ones. FastServe [27] mitigates this by preempting long-running requests after each decode iteration.

Interference occurs intermittently in both settings; prefill happens once per request, while long decode requests are less frequent due to the tail distribution of output lengths. We are addressing the orthogonal HoL blocking problem in serving heterogeneous models, a persistent issue affecting performance in each iteration.

Pipeline merging in training systems. Pipeline merging has been explored in training systems such as Chimera [13] DeepSeek-V3 [16] and RLHFuse [32] to improve GPU utilization. While these systems share ParaFlex's pipeline-optimization goals, they address fundamentally different challenges.

Training workloads are predictable, allowing carefully designed fixed schedules. Serving workloads are highly dynamic and bursty, making fixed pipeline schedules impractical. Likewise, training's primary goal is throughput, while inference is latency-sensitive. Further, training includes both forward and backward passes; inference has only forward passes. These fundamental differences lead to different placement strategies, e.g., training systems use bidirectional pipelines to reduce bubbles, while ParaFlex uses consistently ordered pipeline placement to avoid contention and minimize delays, optimized specifically for forward-only passes.

11 Conclusion

To address limitations in current pipeline-parallel serving systems, we propose ParaFlex, a serving system that allows for autonomous sharding and independent scheduling. This architecture reflects a shift toward decentralized system methodologies, challenging conventional assumptions in LLM serving and paving the way for more resource-efficient multi-model environments. The new architecture introduces a rich design space. We leave the exploration of model dependency such as agentic workflows and multimodal LLMs for future work.

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310* (2024).

[3] Amey Agrawal, Haoran Qiu, Junda Chen, Íñigo Goiri, Chaojie Zhang, Rayyan Shahid, Ramachandran Ramjee, Alexey Tumanov, and Esha Choukse. 2024. Medha: Efficiently Serving Multi-Million Context Length LLM Inference Requests Without Approximations. *arXiv preprint arXiv:2409.17264* (2024).

[4] Sam Altman. 2025. OPENAI ROADMAP UPDATE FOR GPT-4.5 and GPT-5. https://x.com/sama/status/1889755723078443244

[5] Murali Andoorveedu. 2024. [RFC]: Initial support for Pipeline Paralleism. https://github.com/vllm-project/vllm/issues/4461

[6] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506.

[7] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024. MuxServe: Flexible Multiplexing for Efficient Multiple LLM Serving. *arXiv preprint arXiv:2404.02015* (2024).

[8] Hugging Face. 2024. Efficient Training on Multiple GPUs. https://huggingface.co/docs/transformers/main/en/perf_train_gpu_many. Accessed: 2024-12-08.

[9] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[10] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA.

[11] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181* (2024).

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626.

[13] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[14] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679.

[15] Phillip Lippe. 2024. Tensor Parallelism Tutorial. https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/scaling/JAX/tensor_parallel_simple.html. Accessed: 2024-12-08.

[16] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[17] NVIDIA. 2024. Demystifying AI Inference Deployments for Trillion-Parameter Large Language Models. https://developer.nvidia.com/blog/demystifying-ai-inference-deployments-for-trillion-parameter-large-language-models. Accessed: 2024-12-08.

[18] OpenRouter. 2025. Amazon Bedrock Models | OpenRouter. https://openrouter.ai/models?fmt=cards&providers=Amazon%20Bedrock. Accessed: April 17, 2025.

[19] OpenRouter. 2025. Azure | OpenRouter: Browse models provided by Azure. https://openrouter.ai/provider/azure. Accessed: April 17, 2025.

[20] OpenRouter. 2025. LLM Rankings | OpenRouter. https://openrouter.ai/rankings?view=month. Accessed: April 17, 2025.

[21] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[22] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Shengkun Cui, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2024. One Queue Is All You Need: Resolving Head-of-Line Blocking in Large Language Model Serving. arXiv:2407.00047 [cs.DC] https://arxiv.org/abs/2407.00047

[23] Chandrasekharan Rajendran and Oliver Holthaus. 1999. A comparative study of dispatching rules in dynamic flowshops and jobshops. *European journal of operational research* 116, 1 (1999), 156–170.

[24] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1348–1362. doi:10.1109/HPCA61900.2025.00102

[25] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 46745–46771. https://proceedings.mlr.press/v235/strati24a.html

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[27] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Distributed Inference Serving for Large Language Models. arXiv:2305.05920 [cs.LG] https://arxiv.org/abs/2305.05920

[28] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating Serverless LLM Inference with Materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 653–668.

[29] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, Jidong Zhai, Joseph Gonzalez, and Ion Stoica. 2025. Jenga: Effective Memory Management for Serving LLM with Heterogeneity. arXiv:2503.18292 [cs.DC] https://arxiv.org/abs/2503.18292

[30] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578.

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

[32] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. 2024. Optimizing RLHF Training for Large Language Models with Stage Fusion. *arXiv preprint arXiv:2409.13221* (2024).