FLEXPIPE: Adapting Dynamic LLM Serving Through Inflight Pipeline Refactoring in Fragmented Serverless Clusters

Yanying Lin
Shenzhen Institutes of Advanced
Technology, CAS; UCAS
UC San Diego

Shijie Peng
Shenzhen Institutes of Advanced
Technology, CAS
UCAS

Chengzhi Lu
Nanyang Technological University

Chengzhong Xu
University of Macau

Kejiang Ye[*]
Shenzhen Institutes of Advanced
Technology, CAS

Abstract

Serving Large Language Models (LLMs) in production faces significant challenges from highly variable request patterns and severe resource fragmentation in serverless clusters. Current systems rely on static pipeline configurations that struggle to adapt to dynamic workload conditions, leading to substantial inefficiencies.

We present FLEXPIPE, a novel system that dynamically reconfigures pipeline architectures during runtime to address these fundamental limitations. FLEXPIPE decomposes models into fine-grained stages and intelligently adjusts pipeline granularity based on real-time request pattern analysis, implementing three key innovations: fine-grained model partitioning with preserved computational graph constraints, inflight pipeline refactoring with consistent cache transitions, and topology-aware resource allocation that navigates GPU fragmentation. Comprehensive evaluation on an 82-GPU cluster demonstrates that FLEXPIPE achieves up to 8.5× better resource efficiency while maintaining 38.3% lower latency compared to state-of-the-art systems, reducing GPU reservation requirements from 75% to 30% of peak capacity.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; *Cloud computing*; • **Computing methodologies** → *Distributed artificial intelligence*.

Keywords: Large Language Models, Serverless Computing, Resource Efficiency, Pipeline Parallelism, Dynamic Scaling

*Corresponding Author

1 Introduction

The exponential growth in Large Language Model (LLM) parameters [6, 10, 12, 46, 47] has created significant challenges for serving these models in production environments. With models scaling to hundreds of billions of parameters, deploying efficient inference systems requires distributed computing approaches as single-device execution becomes infeasible due to memory constraints [5, 30, 60]. Current serving systems primarily rely on two distributed paradigms: tensor parallelism, which distributes matrix operations across devices with high-bandwidth interconnects, and pipeline parallelism, which segments models into sequential stages with lower communication requirements [21, 26]. However, these approaches face substantial challenges when deployed in real-world serverless environments [27, 53], particularly in balancing computational efficiency with resource adaptability under dynamic request patterns [57] and fragmented cluster resources [51].

Production analysis of Alibaba clusters [1] reveals *two fundamental challenges* that current LLM serving systems cannot address. **i) workload volatility**: request patterns exhibit extreme variability with coefficient of variation (CV) fluctuating up to 7× across timeframes (Fig. 1), causing static pipelines to misalign with shifting workload characteristics [13, 57] and resulting in 17% average GPU utilization. **ii) resource fragmentation**: serverless environments scatter GPUs across heterogeneous workloads [25, 51], preventing

---

the high-bandwidth interconnects essential for tensor parallelism [5, 60]. Our measurements show only 0.02% probability of co-locating 4 GPUs on the same server (§3), forcing suboptimal execution patterns. This architectural mismatch between model requirements and fragmented resource availability fundamentally undermines serving efficiency in elastic environments [27, 53].

Existing systems [1, 14, 20, 22, 24, 26, 57] employ sophisticated pipeline optimization techniques but fundamentally rely on static configurations that cannot adapt to dynamic environments. While these systems achieve impressive performance under stable conditions, they struggle with the dual challenges of workload volatility and resource fragmentation. For instance, AlpaServe [26] optimizes pipeline architectures based on historical request patterns, focusing on long-term performance rather than adapting to short-term variability. Such static approaches inevitably create bottlenecks when faced with rapid workload fluctuations or when fragmented GPU resources prevent optimal tensor-parallel execution (§3). This mismatch between fixed pipeline designs and the dynamic reality of serverless environments results in significant inefficiencies during deployment.

To address these fundamental limitations, we introduce FlexPipe, a dynamically adaptive LLM serving system that performs inflight pipeline refactoring without service interruption. FlexPipe challenges the conventional wisdom that fixed pipeline configurations are necessary for consistent performance. Instead, it exploits a critical insight: pipeline granularity requirements fundamentally shift with workload characteristics. Fine-grained pipelines excel under bursty workloads by distributing buffering capacity across stages, while coarse-grained pipelines minimize communication overhead during stable periods. FlexPipe continuously transitions between these configurations based on real-time coefficient of variation metrics, achieving superior resource efficiency across the full spectrum of serverless request patterns.

However, implementing this approach presents three significant technical challenges: (1) determining optimal partition boundaries that balance computation and communication overhead while preserving computational graph constraints for future reconfiguration, (2) maintaining cache consistency during dynamic pipeline topology transitions without service interruption, and (3) navigating GPU resource fragmentation in highly dynamic serverless environments while minimizing cold-start initialization latency.

To address these challenges, FlexPipe introduces three core innovations: (1) *Fine-grained model partitioning* that decomposes LLMs through a constrained optimization algorithm, balancing computation and communication overhead while preserving computational graph constraints for efficient future reconfiguration; (2) *Inflight pipeline refactoring* that dynamically transitions between pipeline granularities without service interruption, using real-time monitoring
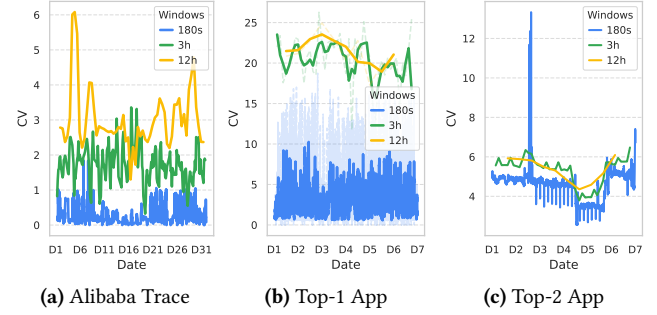

(a) Alibaba Trace **(b)** Top-1 App **(c)** Top-2 App

Figure 1. Request distribution CV (coefficient of variation) variations across different periods. Significant mismatches exist in CV calculated with different window sizes (180s, 3h, 12h), 7× variation exists. (a) Request distribution CV of Alibaba Trace, (b) Request distribution CV of Top-1 App, and (c) Top-2 App from Azure [58].

and CV metrics to seamlessly reconfigure pipeline topologies while maintaining cache consistency; and (3) *Topology-aware resource allocation* that employs hierarchical resource coordination and memory-aware scheduling strategies to navigate GPU fragmentation, minimizing contention during parallel scaling operations while transforming cold starts into efficient warm starts through parameter locality preservation.

We evaluated our approach on a production-grade Kubernetes cluster with 42 servers and 82 GPUs using realistic workloads. Results demonstrate significant performance advantages: 38.3% lower latency under stable workloads and 66.1% improvement under variable conditions. For large models like OPT-66B, FlexPipe achieves 24.38% lower prefill latency compared to alternatives. Most importantly, our system maintains consistent performance as workload variability increases—recovering from pipeline stalls in just 9ms under high-CV conditions (82% faster than competitors) while achieving up to 8.5× better resource efficiency. In a production deployment, our dynamic resource allocation strategy reduced always-on GPU reservation from 75% to just 30% of peak capacity without compromising service quality, while decreasing resource allocation wait time by 85%. These results confirm that FlexPipe effectively addresses the fundamental challenges of resource fragmentation and request variability in serverless environments.

Contributions. Our key contributions include:

- A novel approach for dynamically reconfiguring pipeline architectures in response to changing request distributions without service interruption.
- A method for fine-grained model partitioning that enables efficient pipeline refactoring while preserving computational efficiency.
- A system for enhancing LLM inference elasticity through dynamic resource allocation and pipeline topology adaptation.
- Empirical evidence demonstrating FlexPipe's effectiveness through extensive evaluation on production-grade infrastructure with real-world workloads.

2 Background

2.1 Distributed LLM Inference Paradigms

The exponential growth in LLM parameters has created fundamental conflicts between memory capacity and computational demands. With models scaling to hundreds of billions of parameters, single-device approaches face severe memory constraints, while inference scenarios encounter compute limitations under concurrent requests [32, 37, 50, 58]. Distributed parallel computing has emerged as the essential solution for large-scale model deployment [18, 47].

Tensor Parallelism: Tensor parallelism distributes tensor operations across multiple devices by partitioning matrix operations row-wise or column-wise [5, 60]. This approach effectively parallelizes multi-head attention mechanisms in Transformers and achieves optimal computational efficiency when GPUs are tightly coupled with high-bandwidth interconnects. However, tensor parallelism fundamentally depends on low-latency, high-bandwidth network communication (NVLink, InfiniBand) for frequent synchronization operations [30]. This network dependency becomes a critical bottleneck in fragmented environments where GPUs are distributed across different physical nodes with limited inter-node bandwidth. In such scenarios, the frequent all-reduce operations required for tensor synchronization can dominate execution time, making tensor parallelism impractical for distributed deployments with commodity network infrastructure.

Pipeline Parallelism: Pipeline parallelism implements an inter-layer decoupling strategy, dividing models into sequential stages based on layer dependencies [5, 8, 15, 21, 34]. This approach employs asynchronous scheduling to achieve spatiotemporal overlap between computation and communication [26, 30]. The fundamental advantage of pipeline parallelism lies in its communication pattern: while tensor parallelism requires $O(n^2)$ all-reduce communications per layer with n devices, pipeline parallelism reduces inter-stage communication to point-to-point transfers with $O(1)$ complexity per stage. This dramatic reduction in communication overhead—from dense synchronization matrices to sparse sequential dependencies—enables pipeline parallelism to maintain performance even when network bandwidth is constrained or latency is high. The asynchronous nature of pipeline execution also enables natural load balancing across heterogeneous hardware configurations, as slower devices can process smaller pipeline stages without blocking faster ones. However, this communication efficiency comes with the inherent challenge of pipeline bubble overhead, where stages remain idle during pipeline fill and drain phases. Sophisticated micro-batching strategies and overlapping techniques are essential to amortize these bubbles and maintain high computational efficiency [21, 24].

2.2 Serverless Challenges for LLM Serving

Serverless computing architectures promise enhanced hardware utilization through dynamic resource provisioning [35, 36, 56, 59]. However, the fundamental design philosophy of serverless platforms creates a fundamental tension with distributed LLM inference requirements.

Modern serverless schedulers [17, 59] implement anti-affinity policies that deliberately scatter service replicas across diverse physical nodes to prvent cascading failures. This spatial distribution conflicts directly with distributed LLM inference, which requires tightly coupled GPU clusters with high-bandwidth interconnects (e.g., NVLink, InfiniBand) for efficient tensor parallelism. The result is a paradoxical scenario: individual GPUs are abundant, but cohesive GPU clusters are scarce, fundamentally undermining communication-intensive parallelism strategies.

The serverless resource allocation model operates through dual-tier provisioning: *always-on resources* (60-75% of peak capacity) guarantee baseline service levels, while *elastic resources* handle demand spikes. This conservative approach, designed to prevent service outages, creates chronic under-utilization during normal operations while still introducing multi-second scaling delays that violate sub-second response requirements for interactive LLM applications.

Beyond serverless, similar fragmentation challenges emerge in multi-tenant clusters enforcing strict isolation, edge computing deployments with heterogeneous hardware, and dedicated clusters supporting diverse workloads. The common thread across these environments is the fundamental tension between resource isolation policies designed for predictable performance and the collaborative access patterns required for efficient distributed inference.

This analysis reveals that resource fragmentation represents a fundamental systems challenge requiring adaptive pipeline architectures that maintain efficiency across diverse and changing resource landscapes, rather than static optimization approaches that assume stable resource topologies.

3 Motivation

Optimizing parallel computing strategies is essential for efficient LLM inference, particularly in serverless environments with fragmented resources and fluctuating request patterns. Our systematic analysis reveals critical correlations between inference performance, pipeline architecture, and request distributions that fundamentally impact serving efficiency.

3.1 Resource Fragmentation in Cloud

To understand resource characteristics in cloud environments, we conducted a two-week analysis of GPU resources from a major cloud provider. Our findings revealed a striking 216% average GPU subscription rate (Fig. 2a), indicating that *two services typically share each GPU*. Memory utilization

Table 1. GPU cluster statistics showing resource utilization patterns.

Metric	Cluster C1	Cluster C2
Number of nodes	430	927
Number of GPUs	468	1,175
SM Utilization (%)		
Mean	16.91	23.74
Median (P50)	9.16	10.85
P95	80.53	85.37
10-30% utilization	31.26%	20.98%
GPU Memory Utilization (%)		
Mean	43.48	50.92
Median (P50)	28.78	53.69
P95	99.09	99.34
10-30% utilization	38.44%	17.78%

Note: C1 represents the inference-only cluster, while C2 is a hybrid training-inference cluster. Both employ dynamic resource scaling strategies.

presented significant variability (Table 1), with P50 servers showing modest 20.3% GPU memory utilization while P99 servers approached saturation at 99.3%.

Resource availability analysis demonstrates severe constraints: securing a single GPU with >85% free memory occurs with only 8.7% probability, while co-locating 4 GPUs on the same server drops to 0.02%. This fragmentation stems from heterogeneous model deployments creating unpredictable memory patterns, performance isolation mechanisms fragmenting resources, and aggressive oversubscription maximizing utilization at the cost of availability.

The practical impact of this fragmentation is substantial. In production clusters, 78% of tensor parallelism requests were forced to degrade to pipeline parallelism due to unavailability of adjacent GPUs with high-bandwidth interconnects. This degradation fundamentally challenges distributed inference paradigms, as tensor operations designed for tightly-coupled execution must be reorganized into pipeline stages with higher communication overhead and suboptimal memory access patterns. Communication-intensive approaches like tensor parallelism require high-bandwidth interconnections between GPUs, yet the *scattered distribution of available GPUs* (Fig. 2b) creates persistent misalignment between physical topology and logical requirements. The *ephemeral nature* of serverless GPU allocation further exacerbates this challenge, as optimal GPU configurations may remain available only briefly before reallocation—creating an inherent incompatibility between tensor-parallel computational models and the reality of fragmented cloud GPU environments.

This GPU fragmentation introduces *significant operational challenges*. Due to the immediate reallocation of released GPUs to competing workloads, production clusters typically adopts conservative scaling strategies—maintaining approximately 75% of historical peak GPU capacity as always-on resources, with the remaining 25% allocated through dynamic
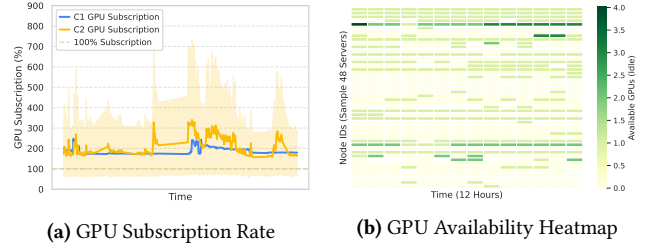

(a) GPU Subscription Rate   **(b)** GPU Availability Heatmap

Figure 2. Resource fragmentation in Alibaba. (a) GPU subscription rate averaging 216%, indicating significant resource overcommitment, and (b) Heatmap revealing spatially scattered GPU availability patterns that impede formation of high-bandwidth interconnected GPU groups needed for tensor parallelism.

Table 2. Performance metrics for different pipeline granularities.

Stages	Load(s)	Compute(ms)	Comm.(ms)	Max Batch
4	47.14	69.94	6.3	128
8	13.05	36.63	14.7	256
16	9.19	18.67	31.5	512
32	5.43	9.67	65.1	1024

*Note: OPT-66B (120GB) performance with sequence length 4096 on A100 GPUs. **Compute** indicates per-stage inference time, **Comm.** represents inter-stage communication overhead, and **Max Batch** shows maximum supported batch size per configuration.*

scaling. This approach produces a **problematic trade-off**: during normal operations, GPU utilization remains unnecessarily low (approximately 17% in our measurement study), yet during traffic spikes, the delayed provisioning of additional GPUs frequently causes SLO violations as scaling operations cannot keep pace with request bursts. The *fundamental disconnect* between idealized theoretical GPU allocation models and the reality of fragmented, ephemeral GPU availability creates a persistent efficiency gap in large-scale LLM serving.

Insight 1: *Resource fragmentation in cloud environments significantly impedes communication-intensive parallelism strategies that rely on high-bandwidth interconnects, necessitating alternative approaches for distributed LLM inference.*

3.2 Pipeline Granularity and Data Parallelism

Pipeline parallelism provides an effective solution for utilizing fragmented GPU resources in LLM inference. The granularity of pipeline stages—defined by operators or parameters per stage—fundamentally affects memory footprint, communication patterns, and computational characteristics. As shown in Table 2, finer-grained partitioning significantly reduces per-stage memory requirements, decreasing parameter loading time and per-stage inference latency, but introduces a critical trade-off: more stages cause proportionally increased inter-stage communication overhead.

In elastic serverless environments, fine-grained pipeline architectures offer significant advantages. Our experiments show 32-stage pipelines reduce parameter loading latency

to just 5.43s—an 8.7× improvement over 4-stage configurations—enabling rapid establishment of data-parallel replicas during demand spikes. This reduction incurs a 65.1ms communication penalty per inference iteration, creating a trade-off between initialization speed and runtime efficiency.

Fine-grained partitioning creates a fundamental trade-off in memory efficiency: 32-stage pipelines achieve 8× larger batch sizes (1024 vs 128) than 4-stage configurations for OPT-66B, dramatically improving GPU tensor core utilization. This increased batch capacity amortizes communication overhead across more requests, creating a counterintuitive effect where higher communication costs are offset by improved computational efficiency.

This granularity trade-off reveals a fundamental architectural insight that directly motivates dynamic adaptation in serverless environments. The dichotomy between fine-grained and coarse-grained pipeline configurations exposes an inherent temporal optimization problem: bursty serverless workloads demand fine-grained pipelines to achieve rapid horizontal scaling (8.7× faster initialization) and exploit large batch processing capacity (8× larger batches) during traffic spikes, while stable operational periods derive greater efficiency from coarser partitions that minimize per-request communication overhead through reduced inter-stage coordination. This creates a dynamic optimization landscape where the optimal pipeline configuration is fundamentally time-dependent and workload-sensitive. The strategic imperative becomes clear: systems must dynamically transition between these configurations—temporarily adopting fine-grained architectures during demand surges to maximize elasticity and batch throughput, then reverting to coarser-grained configurations during stable periods to minimize communication penalties and optimize per-request latency.

Given the inherently bursty nature of serverless workloads, dynamic pipeline granularity adjustment becomes essential. During traffic spikes, fine-grained pipelines provide *rapid scaling* and increased batch processing capacity. As traffic stabilizes, transitioning to coarser-grained configurations minimizes per-request latency through reduced communication overhead, creating an optimal balance between elasticity and efficiency.

Insight 2: *Fine-grained pipelines offer superior elasticity and batch processing capability during bursty workloads but incur communication overhead penalties. The optimal approach dynamically transitions between granularities—using fine-grained configurations temporarily during traffic spikes and reverting to coarser pipelines when workloads stabilize.*

3.3 Pipeline Overhead in Request Distributions

Advanced pipeline systems [1, 5, 20, 26] primarily optimize pipeline architectures offline using historical workload data for long-term performance. However, LLM inference in serverless environments demands short-term optimization: cloud
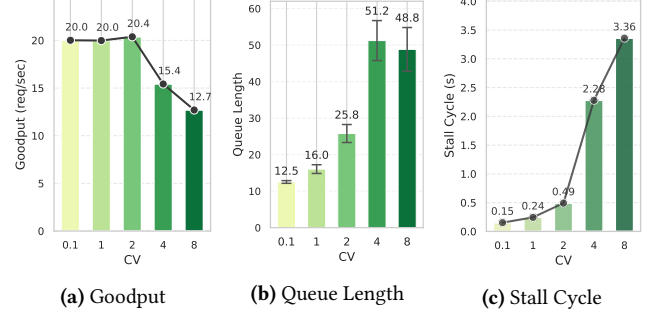

(a) Goodput **(b)** Queue Length **(c)** Stall Cycle

Figure 3. Impact of request distribution variability on pipeline performance. (a) *Goodput* decreases by 37% as CV increases from 0.1 to 8 due to resource contention; (b) Average *queue length* grows nearly 4× with increasing CV, indicating pipeline congestion; (c) *Stall cycle* ratio increases exponentially (22×) at high CV values, showing how static pipelines become inefficient under variable workloads.

providers need to quickly reclaim idle GPU resources, while multi-model services and multi-agent systems require locally optimized performance through concentrated resources. This fundamental mismatch between long-term optimization strategies and short-term load characteristics leads to significant performance degradation, manifested as pipeline bubbles and queuing delays.

As shown in Fig. 1, request distribution CV exhibits *substantial fluctuation*, with variation of up to **7×** across different time windows. To quantify how this volatility impacts performance, we evaluated a static 4-stage OPT-66B pipeline under varying request distributions with a baseline QPS of 20. Fig. 3 reveals the *severe performance degradation* caused by increasing workload variability across multiple dimensions. As CV increases from 0.1 to 8, goodput decreases by 37% (Fig. 3a), while average queue length grows nearly **4×** (Fig. 3b)—clear indicators of pipeline congestion and request backpressure. Most critically, the pipeline stall cycle ratio increases *exponentially*, reaching **22×** at high CV values (Fig. 3c), demonstrating how static pipeline configurations become *fundamentally inefficient* under variable workloads.

Our experiments demonstrate that with the same 4-stage pipeline architecture, a 4× difference in CV leads to a nearly 10× increase in pipeline stall overhead (comparing CV=1 and CV=4). This exponential relationship reveals the *critical importance of adapting pipeline structures to match request volatility patterns*, as static configurations optimized for one CV value perform poorly when the request distribution changes.

To explore the relationship between pipeline architecture and load characteristics, we evaluated three pipeline models (4, 8, and 16 stages) under constant total request volume but varying CV values. As shown in Fig. 4, 4-stage and 8-stage architectures maintain excellent response time stability (approximately 0.5 seconds) under low CV conditions, while the 16-stage architecture's processing time increases to 1.2 seconds (2.7× longer). However, in high-burst scenarios (CV=4), the 16-stage architecture achieves average latency of only

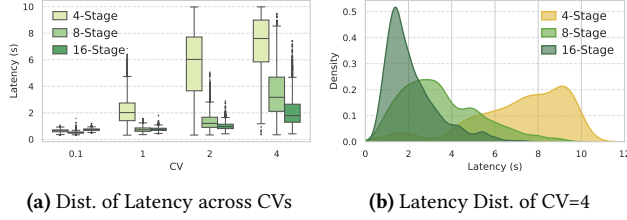**(a)** Dist. of Latency across CVs　　**(b)** Latency Dist. of CV=4

Figure 4. Latency distribution across different request patterns. (a) Box plot comparing pipeline granularities across varying CV values, showing fine-grained pipelines perform better with high-variability workloads; (b) Detailed latency distribution for CV=4 with 4-stage pipeline, revealing significant variance from pipeline stalls.

one-third that of the 4-stage architecture, comparable to the latter's performance at CV=1.

This behavior can be explained through a stochastic process model that reveals the dynamic coupling between pipeline depth (S) and load burstiness (CV). While the theoretical delay of an S-stage pipeline is $T_{pipe} = S \cdot \tau + (S-1) \cdot \delta$ (where τ is single-stage service time and δ is communication overhead), burst requests cause uneven workloads across stages. We established an extended G/G/S queuing model:

$$T_{total} = \underbrace{\frac{\rho^S}{S!(1-\rho)} \cdot \frac{CV_a^2 + CV_s^2}{2}}_{\text{Queue Latency}} + \underbrace{\sum_{i=1}^{S}\left(\frac{\lambda_i}{\mu_i - \lambda_i}\right)}_{\text{Stage Congestion Delay}} \quad (1)$$

where $\rho = \lambda/\mu$ represents system utilization, λ is arrival rate, μ is service rate, CV_a and CV_s denote coefficients of variation for arrival intervals and service times respectively, λ_i is the arrival rate at stage i, and μ_i is the service rate at stage i. As CV_a increases, increasing pipeline stages S produces two opposing effects: 1) fine-grained task segmentation reduces per-stage service time, alleviating congestion; 2) it increases cumulative pipeline register delays. Our experimental data indicates that when $CV_a > 3$, effect 1 dominates, and setting $S \propto \sqrt{CV_a}$ achieves optimal latency. This explains why the 16-stage pipeline achieves a 3× performance improvement over the 4-stage pipeline at CV=4—the deeper pipeline *effectively absorbs peak loads through distributed buffering*.

Insight 3: Request distribution variability causes pipeline stage imbalances, resulting in different optimal architectures for various workloads. In highly bursty environments, deeper pipeline architectures can effectively absorb peak loads through distributed buffering, significantly outperforming static configurations.

4　System Overview

Based on our analysis of resource fragmentation, pipeline granularity trade-offs, and dynamic request patterns in serverless environments, we present FlexPipe, a dynamically adaptive LLM serving system designed to overcome these challenges. As illustrated in Fig. 5, FlexPipe comprises three synergistic components:
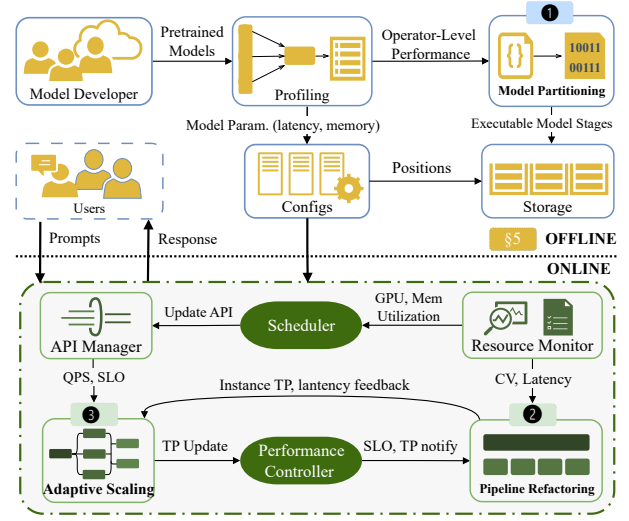

Figure 5. FlexPipe system architecture showing the three core components: ❶ Fine-Grained Pipeline Model Partitioning that decomposes LLMs at operator level for optimal adaptability, ❷ Inflight Pipeline Refactoring that dynamically adjusts pipeline granularity based on request patterns, and ❸ Adaptive Pipeline Scaling that enables efficient resource allocation during traffic fluctuations.

Fine-Grained Pipeline Model Partitioning. This component performs operator-level decomposition of LLMs to create balanced pipeline stages optimized for both computation and reconfiguration potential. By analyzing computation graphs and establishing natural partition boundaries, it creates pipeline architectures ranging from coarse stages with minimal communication overhead to fine-grained stages that enable rapid scaling during bursty workloads.

Inflight Pipeline Refactoring. At FlexPipe's core, this component monitors request distributions and dynamically restructures pipeline topology without interruption. Using coefficient of variation (CV) metrics and queue monitoring, it selects optimal configurations that minimize stalls while maximizing resource utilization. The system seamlessly transitions between fine-grained pipelines (for high-CV traffic) and coarse-grained architectures (for stable workloads) through consistent parameter migration.

Adaptive Pipeline Scaling. This component orchestrates GPU allocation during traffic fluctuations through topology-aware scheduling. It implements a Hierarchical Resource Graph to coordinate parallel scaling while avoiding resource contention, and employs affinity-based scheduling to leverage parameter locality across scaling events, transforming cold starts into efficient warm starts through intelligent cache management.

Key Implementation Challenges. Implementing FlexPipe presents three critical technical challenges: (1) determining optimal operator-level partition boundaries that balance computation-communication trade-offs while preserving refactoring potential, requiring specialized constrained optimization algorithms; (2) maintaining state consistency during topology changes without service interruption through

efficient KV cache synchronization and coordinated parameter migration; and (3) navigating resource fragmentation while minimizing initialization delays using topology-aware allocation strategies that preserve parameter locality across scaling operations. These challenges represent the fundamental tension between dynamic adaptation and efficient resource utilization in serverless environments.

5 Fine-Grained Model Partitioning

To enable adaptation to varying request patterns, FLEXPIPE decomposes models into fine-grained pipeline stages that can be reconfigured at runtime. The core challenge lies in balancing communication overhead and computational efficiency while facilitating dynamic refactoring. This requires solving three critical issues: determining the optimal granularity that satisfies bandwidth constraints while enabling communication-computation overlap, maintaining performance stability across varying micro-batch sizes, and supporting seamless runtime pipeline transitions with minimal synchronization overhead.

FLEXPIPE initiates pipeline optimization through computation graph analysis and operator-level profiling. For a given model M with L layers, we first construct its computation graph $G = (V, E)$ where vertices $v_i \in V$ represent operators and edges $e_{ij} \in E$ denote data dependencies. The Profiling module measures three critical metrics for each operator: computation time $t_c(v_i)$, parameter size $s_p(v_i)$, and activation size $s_a(v_i)$. To achieve optimal partitioning, we employ a dynamic programming algorithm that simultaneously considers communication-computation overlap and future refactoring needs.

The partitioning process solves a constrained optimization problem:

$$
\min_{\{S_k\}} \sum_{k=1}^{K} \left| t_c(S_k) + \frac{s_p(S_k)}{B} - C \right| + \lambda \cdot R(S_k)
$$

$$
\text{s.t.} \bigcup_{k=1}^{K} S_k = V, \quad S_i \cap S_j = \emptyset \ \forall i \neq j \tag{2}
$$

$$
\max_{S_k} s_p(S_k) \leq M_{\text{GPU}}
$$

where K denotes the number of stages, S_k represents the k-th stage containing a subset of operators, V is the complete set of operators in the computation graph, $t_c(S_k)$ is the computation time of stage k, $s_p(S_k)$ is the parameter size of stage k, B represents the inter-stage bandwidth, C is the target computation-communication overlap cycle, M_{GPU} is the GPU memory capacity, and λ is a regularization weight. The regularization term $R(S_k)$ encodes the refactoring potential of each partition, favoring cuts that preserve hierarchical structure boundaries (e.g., attention blocks in Transformers) to facilitate future merging. This formulation ensures

balanced stage execution times while creating natural breakpoints for potential pipeline reconfiguration.

For micro-batch adaptation, we introduce batch-aware transmission scaling:

$$
s_a(S_k, b) = s_a^{base}(S_k) \cdot \left(1 + \alpha \log \frac{b}{b_{\text{base}}} \right) \tag{3}
$$

where b is the micro-batch size, b_{base} is the profiling batch size, and α is the compression factor learned from historical data through linear regression. This allows the system to predict communication patterns for arbitrary batch sizes during online serving. Crucially, the partitioning algorithm preserves the parameter grouping structure to enable future replica alignment - parameters within the same logical group (e.g., attention heads or MLP blocks) are colocated in contiguous memory regions, allowing merged stages to reuse existing memory layouts.

6 Inflight Pipeline Refactoring

The dynamic nature of cloud environments and the resource fragmentation challenges identified earlier necessitate a flexible approach to pipeline management. To address this, we design an inflight pipeline refactoring mechanism (Fig. 6) that can dynamically adjust pipeline granularity during model serving without service interruption. The core challenge lies in dynamically adjusting pipeline granularity under time-varying request distributions while maintaining hardware efficiency and consistency.

We formulate this as a multi-objective optimization problem with temporal constraints. Let $\mathcal{G} = \{g_1, ..., g_K\}$ denote the set of candidate pipeline granularities, where each granularity $g_k = (\eta_k, b_k)$ corresponds to stage count η_k and batch size b_k. The temporal correlation of request patterns is captured by coefficient of variation (CV) $v_t = \frac{\sigma_t}{\mu_t}$, where σ_t and μ_t represent the standard deviation and mean of request arrival intervals at time t, respectively.

6.1 Granularity Adaptation

The optimal granularity g^* is determined through joint optimization of throughput and latency, balancing the trade-off between processing speed and response time:

$$
g^* = \arg\max_{g_k \in \mathcal{G}} \left[\alpha \cdot \frac{T_k}{T_{max}} + (1 - \alpha) \cdot \frac{L_{min}}{L_k} \right] \cdot \exp \left(-\frac{|v_t - v_k|}{\sigma} \right) \tag{4}
$$

where $g_k = (\eta_k, b_k)$ represents granularity configuration k with stage count η_k and batch size b_k, $\mathcal{G}$ is the set of candidate granularities, T_k and L_k denote throughput and latency for granularity g_k, T_{max} and L_{min} are normalization constants, v_t is the current CV value at time t, v_k represents the optimal CV threshold for granularity g_k, $\alpha \in [0, 1]$ is the throughput-latency trade-off weight, and σ controls adaptation sensitivity. Intuitively, this formula finds the sweet

spot between maximizing throughput (first term) and minimizing latency (second term), while the exponential term ensures the selected granularity aligns with the current request pattern. When request patterns are stable (low CV), the system favors coarser granularity to reduce communication overhead; when requests become bursty (high CV), it shifts toward finer granularity to enable rapid scaling.

For multi-granular data parallelism, we introduce a hierarchical scheduling framework that determines the optimal number of parallel instances for each granularity level:

$$\mathcal{M}(g_k) = \left\lfloor \frac{\mu_{total}}{\mu_k} \right\rfloor, \quad \mu_k = \frac{T_k}{\beta_1 + \beta_2 \cdot \eta_k} \tag{5}$$

where $\mathcal{M}(g_k)$ indicates the number of parallel instances for granularity g_k, μ_{total} is the total system processing capacity, μ_k is the effective processing capacity per instance of granularity g_k, T_k is the throughput of granularity g_k, β_1 and β_2 are coordination overhead coefficients that model performance degradation from pipeline coordination, and η_k is the number of stages in granularity g_k. This approach dynamically distributes computational resources across different granularity levels based on their efficiency—finer granularities enable quicker scaling but incur higher coordination overhead, while coarser granularities optimize steady-state performance but adapt more slowly to load changes.

6.2 Hardware Efficiency Optimization

In fragmented cloud environments, efficient GPU allocation becomes critical when multiple models with varying pipeline granularities compete for limited resources. The fundamental challenge lies in balancing resource sharing efficiency against performance isolation requirements. Our key insight is that heterogeneous models can achieve superior resource utilization when their computational patterns complement each other—reducing GPU idle periods while maintaining service quality.

To address this challenge, we formulate GPU resource allocation as a constrained optimization problem that maximizes system efficiency while respecting hardware and performance constraints:

$$\max_{\{x_{ij}\}} \sum_{i=1}^{N} \sum_{j=1}^{M} \left[\frac{T_{ij}}{m_j} - \gamma(CV_i) \cdot \mathbb{I}(\sum_{i'} x_{i'j} > 1) \right] \tag{6}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} x_{ij} \cdot m_j \leq M_j, \quad \forall j \in [1, J] \tag{7}$$

$$\left| \frac{T_{ij}}{T_{i'j'}} - 1 \right| \leq \epsilon, \quad \forall i, i' \in \mathcal{G}_k \tag{8}$$

Problem Formulation: The objective function maximizes throughput efficiency $\frac{T_{ij}}{m_j}$ while penalizing multiplexing overhead through $\gamma(CV_i)$. Here, $x_{ij} \in \{0, 1\}$ indicates whether pipeline stage i is assigned to GPU j, T_{ij} represents the throughput of stage i on GPU j, and m_j denotes memory
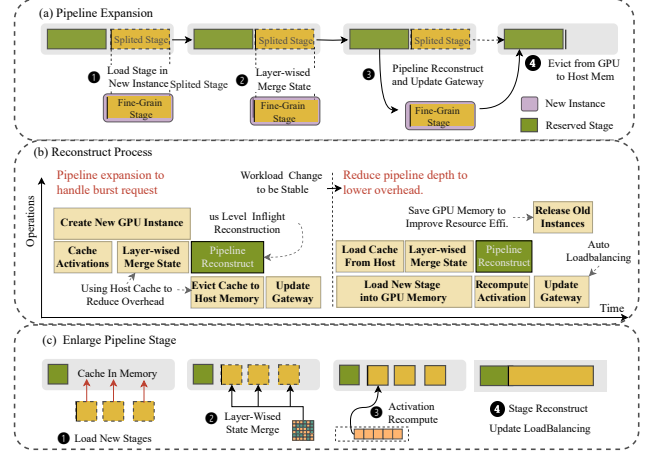

Figure 6. Inflight pipeline refactoring mechanism. (a) Stage refinement process where fine-grained partitioning occurs by evicting parameters and redistributing them onto additional GPUs; (b) Temporal sequence diagram showing synchronization protocol during refactoring; (c) Stage consolidation process where parameters from multiple stages are merged, utilizing host memory caching to minimize loading overhead from persistent storage.

consumption. The memory constraint (Eq. 7) ensures that total memory allocation across all stages assigned to GPU j does not exceed its capacity M_j. The load balancing constraint (Eq. 8) maintains balanced computation times across stages within the same granularity group $\mathcal{G}_k$, preventing pipeline stalls caused by stage imbalances.

FLEXPIPE also *strictly prohibits* multiple pipeline stages from the same model from being allocated to the same GPU. This is critical for preserving performance isolation and preventing resource contention between stages of the same model, as they typically exhibit similar computation patterns and would compete for the same GPU resources rather than complementing each other. By ensuring that different pipeline stages of the same model—regardless of their granularity—are always deployed on separate GPUs, FLEXPIPE *maximizes parallelism while minimizing interference*, creating an optimal balance between resource utilization and computational efficiency.

Multiplexing Penalty Function: The penalty function $\gamma(CV_i)$ models the performance degradation from resource multiplexing based on workload variability:

$$\gamma(CV_i) = \gamma_0 \cdot (1 + \alpha \cdot CV_i^2) \tag{9}$$

where γ_0 represents the base multiplexing penalty and α controls sensitivity to workload variability. The quadratic relationship CV_i^2 reflects our empirical observation that bursty workloads (high CV) create significantly more performance interference when multiplexed due to concurrent resource demand spikes. For stable workloads (low CV), the penalty approaches the minimal γ_0, enabling efficient resource sharing. The indicator function $\mathbb{I}(\sum_{i'} x_{i'j} > 1)$ applies this penalty only when multiple models share the same GPU, ensuring that single-model deployments incur no multiplexing overhead.

Algorithm 1: Inflight Pipeline Refactoring

1: Initialize granularity set $\mathcal{G} = \{g_1, ..., g_K\}$
2: **while** True **do**
3:     Monitor request intensity λ_t and compute characteristic velocity $v_t = \frac{\partial \lambda_t}{\partial t}$
4:     Update queue length $\hat{q}_j$
5:     **for** each granularity $g_k \in \mathcal{G}$ **do**
6:         Compute optimization score:
7:         $S_k = \left[\alpha \cdot \frac{T_k}{T_{max}} + (1 - \alpha) \cdot \frac{L_{min}}{L_k} \right] \cdot \exp\left(-\frac{|v_t - v_k|}{\sigma} \right)$
8:         Evaluate hardware efficiency using Eq. 9
9:     **end for**
10:     Select optimal granularity $g^* = \arg\max_{g_k \in \mathcal{G}} S_k$
11:     **if** $g^* \neq g_{current}$ **then**
12:         Determine required data parallelism using Eq. 5
13:         Perform parameter migration with consistency:
14:         $C(t) = \bigcup_{i \in \text{GPUs}} \text{KV}_i(t) \otimes M_{\text{valid}}$
15:         Update routing metadata and activate new pipeline configuration
16:     **end if**
17:     Wait until next optimization interval
18: **end while**

This formulation addresses the fundamental tension between resource efficiency and performance isolation in fragmented environments. By dynamically adjusting the multiplexing penalty based on workload characteristics, the system makes informed decisions about resource consolidation versus isolation—prioritizing consolidation for stable workloads while maintaining isolation for bursty patterns that would otherwise create performance interference.

6.3 Consistency Maintenance

During pipeline refactoring, maintaining KV cache consistency across distributed GPU instances represents a fundamental challenge that directly impacts inference quality. The key insight is that cache coherence can be preserved through selective synchronization rather than global state replication. We implement a consistency protocol that tracks cache validity at the token level:

$$C(t) = \bigcup_{i \in \text{GPUs}} \text{KV}_i(t) \otimes M_{\text{valid}} \tag{10}$$

where $C(t)$ represents the consistent KV cache state across all GPUs at time t, $\text{KV}_i(t)$ represents the KV cache state on GPU instance i at time t, M_{valid} is a validity mask identifying tokens that need synchronization (with 1 indicating valid tokens and 0 indicating invalid ones), and $\otimes$ denotes element-wise multiplication. During pipeline refactoring, the system performs asynchronous KV cache transfers (Fig. 6(b)) while the inference continues on the original pipeline configuration, minimizing service interruption.

The refactoring algorithm operates through continuous workload monitoring and predictive adaptation. FlexPipe tracks request intensity gradients to anticipate traffic shifts before they manifest as performance degradation, enabling proactive rather than reactive optimization. When workload characteristics deviate from the current pipeline's optimal operating range, the system evaluates alternative configurations using cached performance profiles. This predictive approach transforms pipeline adaptation from a costly reactive process into an efficient proactive mechanism. The refinement process (Fig. 6(a)) partitions computational stages when burst capacity is needed, while consolidation (Fig. 6(c)) merges stages during stable periods to minimize communication overhead. Decision latency remains under 5ms across configurations spanning 2-32 pipeline stages, ensuring that adaptation benefits consistently exceed transition costs even under rapidly changing workloads.

7 Adaptive Pipeline Scaling

Serving LLMs in serverless, dynamic request patterns require responsive resource allocation strategies to maintain service quality during both peak and idle periods. To address service demands during traffic bursts, we propose a dynamic-aware pipeline scaling mechanism. This mechanism leverages dynamic batching [13] to monitor request queue status, combined with stage-level elastic scaling to balance performance and overhead. As shown in Fig. 7, the system performs distributed stage scaling during traffic peaks and automatically reclaims resources when requests subside.

Elastic Scaling Granularity Decision. Selecting the scaling granularity requires balancing three key parameters: ① fine-grained scaling (stage-level) reduces cold-start time but increases communication overhead, while coarse-grained scaling (pipeline-level) does the opposite; ② the coefficient of variation (CV) of traffic reflects request volatility, with high CV scenarios requiring rapid response; ③ queue length (Q) characterizes the urgency of system load. We establish a scaling granularity decision function:

$$m_j = \left\lfloor \frac{G_{\text{max}}}{1 + \beta \cdot e^{-\gamma(cv_j \cdot \hat{q}_j)}} \right\rfloor \tag{11}$$

where m_j is the selected scaling granularity for workload j, G_{max} is the maximum scaling granularity (corresponding to the finest granularity), cv_j is the coefficient of variation for workload j, $\hat{q}_j = \min(q_j/Q_{\text{max}}, 1)$ is the normalized queue length with q_j being the current queue length and Q_{max} being the maximum queue capacity, and β, γ are calibration parameters controlling the sigmoid transition. As the product $cv_j \cdot \hat{q}_j$ increases, the exponential term decays more rapidly, pushing m_j closer to G_{max} to select a finer granularity. This function smoothly adjusts the scaling granularity through its Sigmoid characteristics, avoiding decision oscillation. While
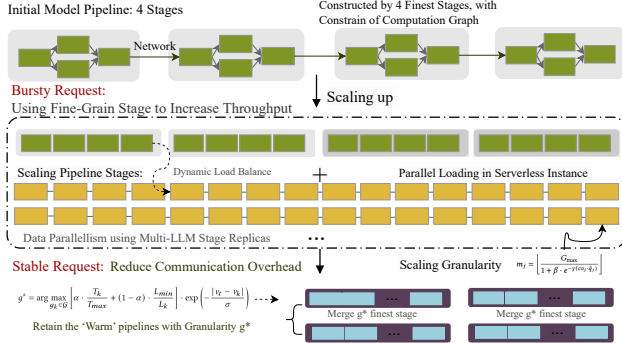
Figure 7. The process of model scaling using fine-grained pipeline stages. FlexPipe conference satisfies the minimum granularity pipeline stage for loading and executing inference. Then, after traffic changes, it modifies to a coarser granularity pipeline stage with fewer additional overheads.

integrating SLO constraints to ensure service quality:

$$\frac{(T_j - S_j) \cdot \sum_{k=1}^{m_j} \mu_{jk}}{Q_j} \geq r_j \quad (12)$$

where T_j is the SLO deadline for workload j, S_j is the initialization time for scaling operations, μ_{jk} represents the expected throughput of the k-th expanded stage, m_j is the number of scaling stages, Q_j is the current queue length for workload j, and r_j is the number of requests to be processed within the deadline. This constraint ensures that the selected granularity m_j can process r_j requests within the time limit T_j.

To address resource contention during multi-model scaling that arises from multiple concurrent requests for GPU memory, PCIe bandwidth, and network resources during rapid parallel deployment, we design a three-level coordination control strategy:

Topology-Aware Resource Coordination. During rapid scaling operations, resource contention emerges as multiple models simultaneously compete for GPU memory, network bandwidth, and storage I/O. To address this challenge, we implement a *Hierarchical Resource Graph* (HRG) that orchestrates resources across three critical levels: server (GPU memory, PCIe bandwidth), rack (network bandwidth), and cluster (storage I/O).

The HRG maintains annotated paths with scaling event markers to identify contention patterns and track resource dependencies. This enables the system to **proactively predict** bottlenecks and intelligently distribute workloads. Rather than treating scaling operations as independent events, HRG directs new instances toward available resources while avoiding paths with recent scaling activities, effectively transforming a *resource contention problem* into a *resource coordination opportunity*.

This topology-aware approach ensures concurrent scaling operations distribute optimally across the physical infrastructure. By respecting the hierarchical nature of datacenter resources, the system makes informed placement decisions

that significantly reduce initialization latency during traffic bursts while maintaining performance isolation between competing workloads.

Memory-Aware Elastic Scaling. In serverless environments, scaled-down model instances have their resources immediately reallocated to competing workloads. This causes cache invalidation, forcing subsequent scale-up operations to incur significant cold-start penalties as parameters reload from slower storage. To address this challenge, FlexPipe implements a two-pronged memory-aware approach preserving locality across scaling operations. First, the system maintains parameter copies in host memory even after GPU eviction, creating a middle-tier cache that survives instance termination and prevents costly reloads from persistent storage. Second, FlexPipe implements an affinity-based scheduling policy prioritizing servers with historical model placement:

$$s^* = \arg\max_{s \in \mathcal{H}_i} \left(w_t \cdot e^{-\lambda(t_{now}-t_s)} + w_g \cdot |g_s \cap G_{avail}| \right) \quad (13)$$

where s^* is the selected server, $\mathcal{H}_i$ tracks servers that previously hosted model i, w_t and w_g are temporal and GPU affinity weights respectively, t_{now} is the current time, t_s is the last time server s hosted model i, λ is the temporal decay rate, g_s is the set of GPUs on server s, G_{avail} represents available GPUs, and $|g_s \cap G_{avail}|$ denotes the number of available GPUs on server s. The temporal decay factor $e^{-\lambda(t_{now}-t_s)}$ prioritizes recently used hosts whose caches are likely still warm. This approach *significantly reduces* initialization time by leveraging cached parameters in host memory, effectively transforming cold starts into warm starts.

8 Implementation

We have implemented FlexPipe with approximately 7K lines of code, including a 3.2K LoC tool for dynamic operator-level model partitioning and merging. After pipeline refactoring, KV cache data must migrate between GPU devices. Using NCCL would introduce significant connection establishment overhead of several seconds and potential bandwidth contention. To address this challenge, we implemented a hierarchical data transfer mechanism that prioritizes RDMA for high-bandwidth, low-latency transfers between GPU devices. For machines without RDMA support, we fall back to the sendfile system call, which enables efficient kernel-space data transfers without redundant copying between user and kernel buffers. This hybrid approach eliminates connection initialization overhead while achieving near-line-rate data transmission speeds.

9 Evaluation

We evaluated FlexPipe on a Kubernetes (v1.23.7) cluster with 42 servers and 82 GPUs, each server having at least

256GB memory and connected via 100Gbps network. For realistic workload patterns, we utilized Microsoft Azure Functions traces [58] supplemented with the Splitwise corpus for prompt generation, enabling rigorous assessment under production-grade request distributions.

Baseline. We compare FlexPipe against two categories of systems. First, serverless-based systems: ServerlessLLM [17], which uses DeepSpeed's parallelism for distributed inference, and Tetris [25], which provides memory-efficient hosting without specialized pipeline parallelism. Second, offline-optimized systems: AlpaServe [26], which configures pipelines based on historical request patterns, and MuxServe [14], which employs statistical multiplexing for multi-tenant serving. We also include recent advances in throughput-latency optimization [1] and interference mitigation [20] to provide comprehensive comparison coverage. These systems lack the dynamic adaptation capabilities of FlexPipe, allowing us to evaluate both serverless efficiency and adaptability against established approaches.

Metric and Model. We evaluate performance using goodput (throughput under quality constraints) and end-to-end latency across varying workload distributions. Experiments use representative models spanning different scales: WHISPER-9B [33], LLAMA2-7B [47], BERT-21B [12], and OPT-66B [10]. We measure initialization latency to validate our pipeline scaling approach, quantify pipeline stall cycles to assess inflight refactoring effectiveness, and analyze GPU memory efficiency to evaluate resource utilization. These metrics provide comprehensive insight into system performance under diverse conditions.

9.1 End-to-End Performance

Latency Breakdown. We analyzed end-to-end latency across systems under varying request distributions (Fig. 8) while maintaining consistent goodput. Under stable workloads (CV=1), FlexPipe achieves 38.3% lower overall latency than AlpaServe and ServerlessLLM while delivering identical goodput (12,000 requests), primarily by reducing queue time by 54.8%. As request variability increases (CV=2), FlexPipe's advantage grows to 46.9% lower latency than MuxServe through a strategic trade-off: accepting higher communication time (105ms vs. 45ms) to achieve 72.6% reduction in queue wait time, all while maintaining maximum goodput. The most significant improvements appear under highly variable workloads (CV=4), where FlexPipe delivers 66.1% lower total latency than AlpaServe and 80.6% lower than MuxServe, while preserving 98.3% of maximum throughput (compared to MuxServe's 33.3% reduction and ServerlessLLM's 40.4% decline). These gains stem from FlexPipe's pipeline reconfiguration that increases communication overhead (225ms vs. 45ms) to dramatically reduce queue times—transforming exponentially growing wait times into manageable communication overhead. This demonstrates a key insight: under bursty
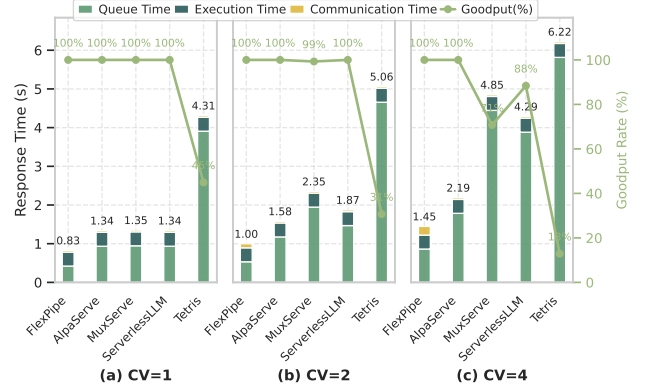

Figure 8. End-to-End Latency Breakdown across varying request distributions. FlexPipe maintains lower overall latency despite higher communication overhead by significantly reducing queue wait times: (a) CV=1 (stable workload), (b) CV=2 (moderate variability), and (c) CV=4 (highly variable workload).
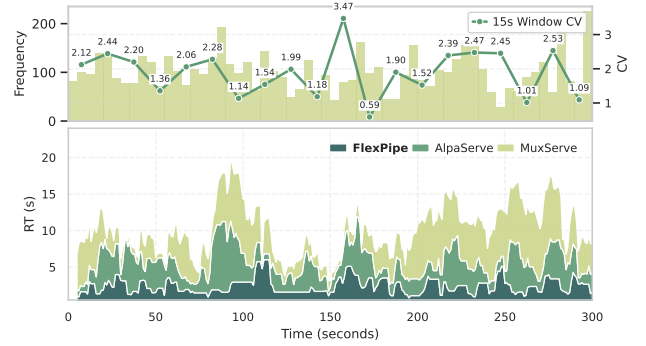

Figure 9. Latency under highly variable workload (CV=8, first 300s). (a) Request distribution CV variability measured in 15s windows, (b) Response latency comparison across systems.

workloads, communication-intensive fine-grained pipelines significantly outperform static architectures trapped in queue buildup cycles.

All experiments used a baseline of 20 QPS across the complete 2-hour lifecycle, with different CV values creating varying peak loads. While static approaches typically provision GPU resources to match peak volumes, FlexPipe maintains only 30% of peak capacity as always-ready resources, with remaining capacity allocated through dynamic scaling. This resource allocation strategy enables FlexPipe to maintain stable performance and consistent goodput across all workload variability levels while static systems suffer progressively degrading performance as request patterns become more erratic.

Burst Absorption. Fig. 9 demonstrates system performance under extreme workload variability (CV=8). Fig. 9(a) reveals substantial fluctuations in 15-second measurement windows, with CV ranging from 0.59 to 3.47—highlighting the challenging dynamics of bursty serverless environments. As shown in Fig. 9(b), while MuxServe experiences sustained high latencies (frequently exceeding 10 seconds) and AlpaServe exhibits periodic performance spikes, FlexPipe
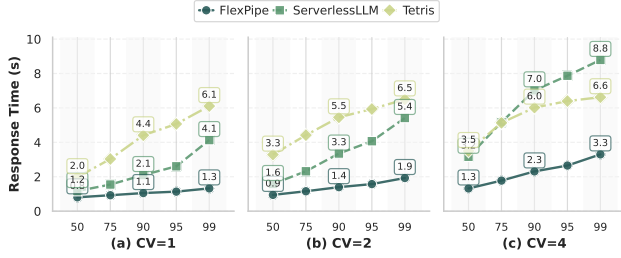
Figure 10. Performance stability analysis across varying request distributions (CV=1, 2, 4), showing FLEXPIPE maintains consistently lower latency percentiles even as workload variability increases in serverless.

maintains significantly lower and more consistent response times throughout the evaluation period, even during intense traffic surges at 75s and 165s intervals.

This comparison specifically includes non-serverless GPU multiplexing systems because they represent the SOTA in handling variable workloads through GPU sharing [1, 20], yet fundamentally differ from FLEXPIPE in their adaptation approach. Unlike serverless-oriented systems that focus on rapid resource provisioning, these multiplexing systems optimize for maximum GPU utilization through sophisticated sharing strategies—providing the most challenging performance baseline for FLEXPIPE's dynamic adaptation mechanism. The results demonstrate that even compared to systems specifically designed for workload variability, FLEXPIPE's dynamic pipeline refactoring capability more effectively absorbs request bursts by transitioning between pipeline granularities based on real-time traffic patterns. By using finer-grained pipelines during traffic spikes and coarser configurations during stable periods, FLEXPIPE avoids the queuing delays that plague static architectures, resulting in more predictable performance even under highly variable workloads.

9.2 Performance Stability

To evaluate system stability under dynamic serverless workloads, we analyzed latency percentiles across varying request distributions (Fig. 10). We focus specifically on Serverless-LLM and Tetris as representative serverless LLM deployment approaches facing unique resource elasticity and fragmentation challenges.

Under stable traffic patterns (CV=1), FLEXPIPE maintains a tight latency distribution with significantly lower P99 latency compared to ServerlessLLM and Tetris. This advantage becomes increasingly pronounced as request variability intensifies. At moderate variability (CV=2), FLEXPIPE's P99 latency remains well-controlled while serverless competitors show substantial degradation. The stability gap widens further under highly variable workloads (CV=4), where FLEXPIPE consistently maintains much lower P99 latency compared to both ServerlessLLM and Tetris, demonstrating superior performance stability even in challenging conditions.

FLEXPIPE's dynamic pipeline refactoring prevents the exponential latency degradation observed in static serverless
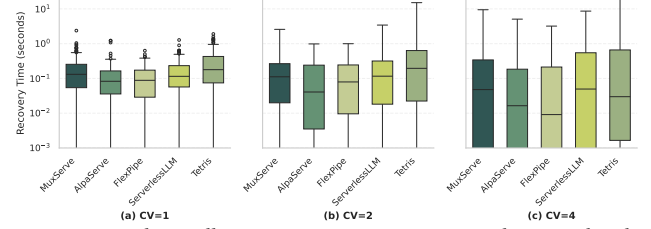

Figure 11. Pipeline stall recovery time across systems and request distribution variability (CV). FLEXPIPE achieves substantially faster recovery under high-variability workloads (9ms at CV=4), demonstrating the effectiveness of dynamic pipeline refactoring in addressing structural stall causes.

architectures. The latency percentile analysis shows FLEX-PIPE maintains consistently lower latency across all percentiles—particularly at P90-P99 where traditional systems exhibit dramatic increases. This advantage becomes more pronounced as CV increases from 1 to 4, with FLEXPIPE maintaining well-controlled P99 latency while competitors experience 2-3× degradation. Such predictability is crucial in serverless environments where resource fragmentation introduces natural variability. By dynamically adjusting pipeline granularity based on workload characteristics, FLEXPIPE effectively transforms queueing delays into manageable communication overhead, delivering significantly more stable performance under challenging conditions.

9.3 Pipeline Stall Recovery

To quantify each system's resilience to pipeline stalls, we established an objective measurement methodology. We define a stall as occurring when response latency exceeds 1.5× the baseline latency (P25 latency under normal operation), and recovery as the point when latency returns to within 1.2× of this baseline. The elapsed time between these events constitutes the recovery duration.

Fig. 11 illustrates recovery performance across systems under varying request distributions. Under stable workloads (CV=1), FLEXPIPE's median recovery time (88ms) is comparable to AlpaServe (83ms), while significantly outperforming MuxServe (131ms), ServerlessLLM (115ms), and Tetris (179ms). As variability increases to moderate levels (CV=2), FLEXPIPE maintains consistent recovery performance (79ms) while other systems show mixed behavior.

The most significant advantage emerges under highly variable workloads, where FLEXPIPE achieves remarkably rapid recovery (9ms)—44% faster than AlpaServe (16ms) and 82% faster than both MuxServe (48ms) and ServerlessLLM (50ms). This exceptional performance stems from FLEXPIPE's ability to dynamically reconfigure pipeline topology in response to detected imbalances, rather than merely adjusting batch sizes or waiting for queue drain as static systems must do.

When a stall is detected, FLEXPIPE's pipeline monitoring system identifies congested stages through distributed queue-length analysis, then initiates targeted refactoring to redistribute computational bottlenecks. By maintaining KV cache consistency during transitions (as described in §6),
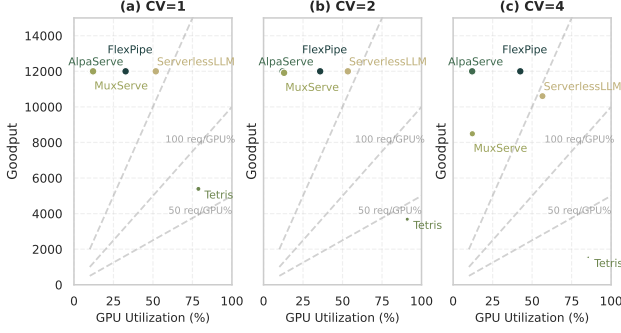
Figure 12. Resource efficiency analysis across varying request distributions: (a) CV=1 (stable workload), (b) CV=2 (moderate variability), (c) CV=4 (high variability).

Figure 13. Performance comparison with production workloads: (a) Average prefill latency across model scales showing FlexPipe's consistent advantage (6.43%-24.38% improvement), (b) Latency distribution showing tighter performance bounds with fewer outliers.

FlexPipe achieves architectural adaptation during live inference without service interruption—transforming potential performance-degrading stalls into brief transitional states and enhancing system resilience under variable workloads.

9.4 Resource Efficiency

We evaluate system efficiency by examining the relationship between GPU utilization and achieved goodput under varying request distributions. Fig. 12 demonstrates FlexPipe's optimal balance between resource utilization and throughput across workload variability—critical for serverless environments with resource fragmentation.

Under stable workloads (Fig. 12(a), CV=1), FlexPipe achieves maximum goodput at 33% GPU utilization—3× more efficient than AlpaServe with identical throughput through fine-grained pipeline partitioning that distributes load across fragmented resources. As variability increases (Fig. 12(b), CV=2), FlexPipe maintains near-maximum goodput with modest utilization increases (36%), while competitors experience throughput degradation despite higher resource consumption. Under highly variable workloads (Fig. 12(c), CV=4), this efficiency gap becomes pronounced: Tetris achieves only 1,543 requests/second despite 85% GPU utilization, while FlexPipe maintains 12,000 requests/second at 43% utilization—8.5× better resource efficiency. This disparity reveals that high GPU utilization in static systems often indicates resource contention rather than productive computation, whereas FlexPipe's dynamic topology optimization prevents such inefficiencies by continuously rebalancing bottlenecks, ensuring GPU cycles translate to inference throughput rather than synchronization overhead.

This analysis reveals FlexPipe's fundamental advantage: while static pipeline architectures show inverse relationships between GPU utilization and throughput under variable workloads, FlexPipe maintains proportional resource scaling. Dynamic pipeline refactoring enables FlexPipe to require only 30% of peak capacity as always-ready resources—a 70% reduction compared to static approaches that provision for peak demand. The remaining capacity is allocated through
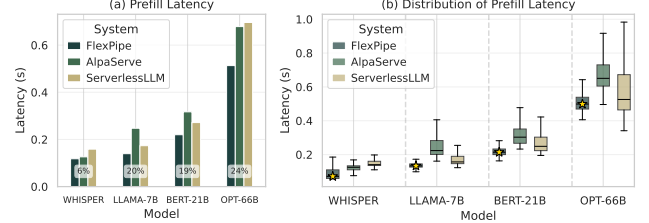
elastic scaling with 5-minute reclamation windows, directly addressing the resource fragmentation challenges.

9.5 Performance in Production Workloads

To evaluate real-world effectiveness, we deployed FlexPipe on our cluster using production workload traces across four representative models. As shown in Fig. 13(a), FlexPipe consistently achieves lower prefill latency across all model scales—from lightweight WHISPER-9B (6.43% improvement over AlpaServe) to large-scale OPT-66B (24.38% reduction compared to AlpaServe).

The performance differential increases with model complexity: 19.52% improvement for LLAMA-7B and 18.97% for BERT-21B over ServerlessLLM. This scalability advantage stems from FlexPipe's dynamic pipeline refactoring capability, which becomes increasingly valuable as computational profiles grow more complex. The latency distribution in Fig. 13(b) further demonstrates that FlexPipe delivers not only lower average latency but significantly tighter performance distributions with fewer outliers—critical for maintaining consistent service-level objectives in production. This aligns with recent work on fairness in serving [39] and end-to-end optimization [43] that emphasizes consistent performance delivery across diverse workloads.

Notably, FlexPipe achieves its greatest advantage (24.38%) on the largest OPT-66B model, where effective pipeline stage management becomes most critical. By dynamically redistributing computational bottlenecks and minimizing pipeline stalls based on real-time workload characteristics, FlexPipe achieves an average 17.32% latency improvement across all models, confirming the effectiveness of adaptive pipeline granularity optimization in dynamic serving environments.

9.6 Case Study in Production Cluster

To validate our approach in production environments, we conducted a phased rollout of FlexPipe in production serverless GPU cluster, gradually migrating LLM serving traffic. Results demonstrate substantial operational improvements: resource allocation wait times decreased by 85% compared to the static baseline, while instance initialization latency was reduced by 72% on average across all model sizes.

Most significantly, FLEXPIPE's dynamic resource allocation strategy enabled a dramatic reduction in the always-on GPU reservation—from 75% to just 30% of historical peak capacity—without compromising service quality. This efficiency gain translates to substantial operational cost savings while maintaining consistent performance under varying workloads. The reduced startup latency and improved resource allocation directly address the fundamental challenges of resource fragmentation and request variability identified in our motivation analysis (§3), confirming the practical benefits of inflight pipeline refactoring in production serverless environments.

10 Related Work

The challenges of LLM serving in fragmented serverless environments have driven extensive research across multiple dimensions. We organize related work into three categories based on their approach to addressing resource fragmentation and workload volatility.

Static Pipeline Optimization Traditional pipeline optimization approaches use offline graph partitioning for static workloads. Representative systems like Megatron [31] and Alpa [60] employ sophisticated algorithms to optimize communication patterns and stage placement [15, 28, 30, 44, 45]. These approaches fundamentally assume predictable resource availability and stable request patterns, making them inadequate for fragmented serverless environments where resource topology changes dynamically. Unlike FLEXPIPE's runtime adaptation capability, they lack the ability to respond to real-time resource fragmentation or workload volatility.

Resource Multiplexing and Sharing Resource multiplexing approaches address multi-tenant efficiency through three primary strategies. Memory optimization systems like vLLM [22] and FlexGen [40] enhance throughput through page-based memory management, while multi-tenant frameworks such as Punica [11], SLoRA [38], and BlockLLM [19] enable parameter sharing across models. Statistical multiplexing systems including MuxServe [14], Llumnix [42], Orca [55], Varuna [9], FaaST [16], and USHER [41] maximize GPU utilization through intelligent scheduling, elastic scaling, and holistic interference avoidance. However, these approaches optimize for predictable resource patterns and struggle when resource availability fluctuates rapidly—a fundamental characteristic of serverless environments. Unlike static multiplexing strategies, FLEXPIPE employs inflight refactoring to dynamically reconfigure pipeline topology, maintaining efficiency across variable workload distributions without relying on stable resource assumptions.

Serverless and Elastic Serving Serverless adaptations for LLM serving have emerged to address cloud-native deployment challenges through three primary strategies. Resource provisioning systems like ServerlessLLM [17] and Tetris [25]

optimize for rapid resource provisioning and cold-start minimization, while FaaS-optimized solutions such as SAND [2], FaaSnap [7], FaaSNet [48], BATCH [3], and Groundhog [4] handle resource elasticity through function-level optimization. Computation disaggregation approaches exemplified by DistServe [61] and InfiniGen [23] separate prefill and decoding phases to enable independent scaling, while context optimization systems like CacheGen [29] and CacheBlend [54] focus on efficient KV cache management. Additional approaches include Tabi [49] for multi-level optimization and FlashLLM [52] for cost-effective inference. However, these approaches fundamentally optimize individual components independently, addressing either resource efficiency or latency in isolation without considering the dynamic tension between resource fragmentation and workload volatility. Unlike these static optimization strategies, FLEXPIPE provides coordinated pipeline adaptation that continuously reconfigures execution topology based on real-time patterns, representing a paradigm shift from component-level optimization toward holistic system adaptation.

11 Conclusion

We presented FLEXPIPE, a fundamentally new approach to LLM serving that challenges the conventional wisdom of static pipeline optimization. Through inflight pipeline refactoring, FLEXPIPE transforms how distributed inference systems adapt to resource fragmentation and workload volatility—two pervasive challenges in modern serverless environments. By enabling continuous topology reconfiguration without service interruption, FLEXPIPE demonstrates that fine-grained adaptability, rather than static optimization, is the key to achieving both resource efficiency and performance consistency in dynamic serving environments.

12 Acknowledgments

References

[1] 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. In *18th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara CA USA July 10-12 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 117–134.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing.. In *Proc. 2018 USENIX Annu. Tech. Conf. USENIX ATC 2018 Boston MA USA July 11-13 2018 (USENIX ATC 2018)*. 923–935.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020-11. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20 Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC 2020)*. IEEE, 69. https://doi.org/10.1109/sc41405.2020.00073

[4] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. 2023-05-08. Groundhog: Efficient Request Isolation in FaaS. In *Proc. Eighteenth Eur. Conf. Comput. Syst. (EuroSys '23)*. ACM, 398–415. https://doi.org/10.1145/3552326.3567503

[5] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, et al. 2022-11. DeepSpeed- Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *SC22 Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC 2022)*. IEEE, 46:1–46:15. https://doi.org/10.1109/sc41404.2022.00051

[6] Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, et al. 2023-09-13. PaLM 2 Technical Report. https://doi.org/10.48550/arXiv.2305.10403

[7] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022-03-28. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proc. Seventeenth Eur. Conf. Comput. Syst. (EuroSys '22)*. ACM, 730–746. https://doi.org/10.1145/3492321.3524270

[8] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022-03-28. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In *Proc. Seventeenth Eur. Conf. Comput. Syst. (EuroSys '22)*. ACM, 472–487. https://doi.org/10.1145/3492321.3519584

[9] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022-03-28. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In *Proc. Seventeenth Eur. Conf. Comput. Syst. (EuroSys '22)*. ACM, 472–487. https://doi.org/10.1145/3492321.3519584

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, et al. 2020. Language Models Are Few-Shot Learners. 33 (2020), 1877–1901.

[11] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2023-10-28. Punica: Multi-Tenant LoRA Serving. https://doi.org/10.48550/arXiv.2310.18547

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019-05-24. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/arXiv.1810.04805

[13] Khaled Diab, Parham Yassini, and Mohamed Hefeeda. 2022. Orca: Server-assisted Multicast for Datacenter Networks.. In *19th USENIX Symp. Networked Syst. Des. Implement. NSDI 2022 Renton WA USA April 4-6 2022 (NSDI 2022)*. 1075–1091.

[14] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. 2024-06-13. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. https://doi.org/10.48550/arXiv.2404.02015

[15] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, et al. 2021-02-17. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP '21)*. ACM, 431–445. https://doi.org/10.1145/3437801.3441593

[16] Mohammadbagher Fotouhi, Derek Chen, and Wes J. Lloyd. 2019-12-09. Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application. In *Proc. 5th Int. Workshop Serverless Comput. (Middleware '19)*. ACM, 49–54. https://doi.org/10.1145/3366623.3368141

[17] 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 Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara CA USA July 10-12 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 135–153.

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, et al. 2024-11-23. The Llama 3 Herd of Models. https://doi.org/10.48550/arXiv.2407.21783

[19] Bodun Hu, Jiamin Li, Le Xu, Myungjin Lee, Akshay Jajoo, Geon-Woo Kim, Hong Xu, and Aditya Akella. 2024-09-23. BlockLLM: Multi-tenant Finer-grained Serving for Large Language Models. https://doi.org/10.48550/arXiv.2404.18322

[20] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, et al. 2024-01-20. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. https://doi.org/10.48550/arXiv.2401.11181

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, et al. 2019-12-08. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Number 10. Curran Associates Inc., 103–112.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, et al. 2023-10-23. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. 29th Symp. Oper. Syst. Princ. (SOSP '23)*. ACM, 611–626. https://doi.org/10.1145/3600006.3613165

[23] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara CA USA July 10-12 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 155–172.

[24] Conglong Li, Zhewei Yao, Xiaoxia Wu, Minjia Zhang, Connor Holmes, Cheng Li, and Yuxiong He. 2023-02-07. DeepSpeed Data Efficiency: Improving Deep Learning Model Quality and Training Efficiency via Efficient Data Sampling and Routing. https://doi.org/10.48550/arXiv.2212.03597

[25] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing.. In *Proc. 2022 USENIX Annu. Tech. Conf. USENIX ATC 2022 Carlsbad CA USA July 11-13 2022 (USENIX ATC 2022)*. USENIX Association.

[26] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving.. In *17th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2023 Boston MA USA July 10-12 2023 (OSDI 2023)*. USENIX Association, 663–679.

[27] Yanying Lin, Yanbo Li, Shijie Peng, Yingfei Tang, Shutian Luo, Haiying Shen, Chengzhong Xu, and Kejiang Ye. 2024-07. QUART: Latency-Aware FaaS System for Pipelining Large Model Inference. In *2024 IEEE 44th Int. Conf. Distrib. Comput. Syst. ICDCS*. 1–12. https://doi.org/10.1109/ICDCS60910.2024.00010

[28] Zhiqi Lin, Youshan Miao, Guodong Liu, Xiaoxiang Shi, Quanlu Zhang, Fan Yang, Saeed Maleki, Yi Zhu, et al. 2023-01-21. SuperScaler: Supporting Flexible DNN Parallelization via a Unified Abstraction. https://doi.org/10.48550/arXiv.2301.08984

[29] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, et al. 2024-08-04. CacheGen: Fast Context Loading for Language Model Applications via KV Cache Streaming.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019-10-27. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proc. 27th ACM Symp. Oper. Syst. Princ. (SOSP '19)*. ACM, 1–15. https://doi.org/10.1145/3341301.3359646

[31] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, et al. 2021-11-14. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC '21)*. ACM, 58. https://doi.org/10.1145/3458817.3476209

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, 'I nigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *51st ACMIEEE Annu. Int. Symp. Comput. Archit. ISCA 2024 B. Aires Argent. June 29 - July 3 2024*. IEEE, 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[33] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022-12-06. Robust Speech Recognition via Large-Scale Weak Supervision. https://doi.org/10.48550/arXiv.2212.04356

[34] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020-11. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC20 Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC 2020)*. IEEE, 20. https://doi.org/10.1109/sc41405.2020.00024

[35] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, et al. 2023-10-23. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proc. 29th Symp. Oper. Syst. Princ. (SOSP '23)*. ACM, 231–246. https://doi.org/10.1145/3600006.3613155

[36] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Laurin Brandner, Anne Koziolek, and Torsten Hoefler. 2025. SeBS-Flow: Benchmarking Serverless Cloud Function Workflows. In *Proc. Twent. Eur. Conf. Comput. Syst. EuroSys 2025 Rotterdam Neth. 30 March 2025 - 3 April 2025*. ACM, 902–920. https://doi.org/10.1145/3689031.3717465

[37] Mohammad Shahrad, Rodrigo Fonseca, 'I nigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, et al. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider.. In *Proc. 2020 USENIX Annu. Tech. Conf. USENIX ATC 2020 July 15-17 2020 (USENIX ATC 2020)*. 205–218.

[38] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, et al. 2024-06-05. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. https://doi.org/10.48550/arXiv.2311.03285

[39] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara CA USA July 10-12 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 965–988.

[40] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, et al. 2023-06-15. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU.. In *Int. Conf. Mach. Learn. ICML 2023 23-29 July 2023 Honol. Hawaii USA (ICML 2023)*. 31094–31116.

[41] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. 2024. USHER: Holistic Interference Avoidance for Resource Optimized ML Inference. In *18th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara

[42] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara CA USA July 10-12 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 173–191.

[43] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2025. Towards End-to-End Optimization of LLM-based Applications with Ayo. In *Proc. 30th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. Vol. 2 ASPLOS 2025 Rotterdam Neth. 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 1302–1316. https://doi.org/10.1145/3676641.3716278

[44] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. 2021-05. Automatic Graph Partitioning for Very Large-scale Deep Learning. In *2021 IEEE Int. Parallel Distrib. Process. Symp. IPDPS (IPDPS 2021)*. IEEE, 1004–1013. https://doi.org/10.1109/ipdps49936.2021.00109

[45] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization.. In *Adv. Neural Inf. Process. Syst. 34 Annu. Conf. Neural Inf. Process. Syst. 2021 NeurIPS 2021 Dec. 6-14 2021 Virtual (NeurIPS 2021, Vol. 34)*. Curran Associates, Inc., 24829–24840.

[46] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timoth'ee Lacroix, Baptiste Rozi'ere, Naman Goyal, et al. 2023-02-27. LLaMA: Open and Efficient Foundation Language Models. https://doi.org/10.48550/arXiv.2302.13971

[47] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, et al. 2023-07-19. Llama 2: Open Foundation and Fine-Tuned Chat Models. https://doi.org/10.48550/arXiv.2307.09288

[48] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute.. In *Proc. 2021 USENIX Annu. Tech. Conf. USENIX ATC 2021 July 14-16 2021 (USENIX ATC 2021)*. USENIX Association, 443–457.

[49] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023-05-08. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proc. Eighteenth Eur. Conf. Comput. Syst. (EuroSys '23)*. ACM, 233–248. https://doi.org/10.1145/3552326.3587438

[50] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, et al. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters.. In *19th USENIX Symp. Networked Syst. Des. Implement. NSDI 2022 Renton WA USA April 4-6 2022 (NSDI 2022)*. USENIX Association, 945–960.

[51] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent.. In *Proc. 2023 USENIX Annu. Tech. Conf. USENIX ATC 2023 Boston MA USA July 10-12 2023 (USENIX ATC 2023)*. USENIX Association, 995–1008.

[52] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, et al. 2023-10-01. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. 17, 2 (2023-10-01), 211–224. https://doi.org/10.14778/3626292.3626303

[53] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022-02-28. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS '22)*. ACM, 768–781. https://doi.org/10.1145/3503222.3507709

[54] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, et al. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proc. Twent. Eur. Conf. Comput. Syst. EuroSys 2025 Rotterdam Neth. 30 March 2025 - 3 April 2025*. ACM, 94–109. https://doi.org/10.1145/3689031.3696098

[55] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models.. In *16th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2022 Carlsbad CA USA July 11-13 2022 (OSDI 2022)*. USENIX Association, 521–538.

[56] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating Serverless LLM Inference with Materialization. In *Proc. 30th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. Vol. 1 ASPLOS 2025 Rotterdam Neth. 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 653–668. https://doi.org/10.1145/3669940.3707285

[57] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild.. In *20th USENIX Symp. Networked Syst. Des. Implement. NSDI 2023 Boston MA April 17-19 2023 (NSDI 2023)*. USENIX Association, 787–808.

[58] Yanqi Zhang, 'I nigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021-10-26. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ. (SOSP '21)*. ACM, 724–739. https://doi.org/10.1145/3477132.3483580

[59] Zili Zhang, Chao Jin, and Xin Jin. 2024. Jolteon: Unleashing the Promise of Serverless for Serverless Workflows.. In *21st USENIX Symp. Networked Syst. Des. Implement. NSDI 2024 St. Clara CA April 15-17 2024 (NSDI 2024)*. USENIX Association, 167–183.

[60] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, et al. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning.. In *16th USENIX Symp. Oper. Syst. Des. Implement. OSDI 2022 Carlsbad CA USA July 11-13 2022 (OSDI 2022)*. 559–578.

[61] 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 Symp. Oper. Syst. Des. Implement. OSDI 2024 St. Clara CA USA July 10-12 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 193–210.