Check for updates

Software: Practice and Experience

WILEY

RESEARCH ARTICLE

BanaServe: Unified KV Cache and Dynamic Module Migration for Balancing Disaggregated LLM Serving in AI Infrastructure

Yiyuan He[1,2,3] | Minxian Xu[2] | Jingfeng Wu[2,4] | Jianmin Hu[1,2,3] | Chong Ma[3] | Min Shen[3] | Le Chen[3] | Chengzhong Xu[5] | Lin Qu[3] | Kejiang Ye[2]

[1]Southern University of Science and Technology, Shenzhen, China | [2]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China | [3]AIOS Team, Alibaba Group Inc., HangZhou, China | [4]University of Chinese Academy of Sciences, China | [5]State Key Lab of IOTSC, Faculty of Science and Technology, University of Macau, Macau SAR, China

Correspondence: Minxian Xu (mx.xu@siat.ac.cn)

Received: 25 September 2025 | **Revised:** 5 December 2025 | **Accepted:** 2 January 2026

Keywords: AI infrastructure | cloud computing | disaggregated LLM serving | large language models | resource management

ABSTRACT

Objective: Large Language Models (LLMs) are increasingly deployed in modern AI infrastructure, creating a strong demand for high-throughput and resource-efficient serving systems. Disaggregated LLM serving, which decouples prompt prefill from auto-regressive decode to accommodate their heterogeneous compute and memory characteristics, has emerged as a promising architecture. However, existing disaggregated serving systems suffer from three fundamental limitations: static resource allocation that fails to adapt to highly dynamic workloads, severe load imbalance between compute-bound prefill and memory-bound decode stages, and prefix-cache-aware routing that skews load distribution and creates performance hotspots. These issues collectively limit resource utilization, scalability, and the ability to meet service level objectives (SLOs) under real-world workloads.

Methods: To address these challenges, we propose BanaServe, a dynamic orchestration framework for disaggregated LLM serving that continuously rebalances both computational and memory resources across prefill and decode instances. BanaServe introduces three key mechanisms: (i) layer-level weight migration to enable coarse-grained redistribution of computation, (ii) attention-level Key–Value (KV) cache migration for fine-grained memory load balancing, and (iii) a Global KV Cache Store with layer-wise overlapped transmission to decouple routing decisions from cache placement. Together, these mechanisms eliminate cache-induced hotspots and allow routers to perform purely load-aware scheduling with minimal latency overhead. BanaServe is implemented on top of state-of-the-art LLM serving frameworks, including vLLM and DistServe.

Results: We evaluate BanaServe under diverse and challenging workloads, including long-context inference, bursty request arrivals, and mixed prompt–generation patterns. Experimental results show that, compared to vLLM, BanaServe improves throughput by 1.2–3.9× and reduces total processing time by 3.9%–78.4%. In comparison with DistServe, BanaServe achieves 1.1–2.8× higher throughput while reducing latency by 1.4%–70.1%. These gains are consistent across workload variations, demonstrating BanaServe's robustness under highly dynamic serving conditions.

Conclusion: BanaServe demonstrates that dynamic, multi-granularity resource rebalancing and cache-decoupled routing are essential for efficient disaggregated LLM serving. By jointly addressing resource elasticity, stage imbalance, and cache-induced

© 2026 John Wiley & Sons Ltd.

load skew, BanaServe substantially improves throughput, latency, and resource utilization in real-world deployments. This work provides a practical and scalable foundation for next-generation LLM serving systems operating under dynamic and heterogeneous workloads.

1 | Introduction

The rapid advancement of Large Language Models (LLMs) such as GPT-4 [1], LLaMA [2], and Claude [3] has revolutionized applications in conversational AI, code generation, knowledge retrieval, and content creation [4, 5]. With models containing billions of parameters [6], serving LLMs efficiently at scale presents significant challenges: maintaining low latency for interactive applications, achieving high throughput for cost-effectiveness, and maximizing hardware utilization under dynamic workloads. These challenges are particularly acute as LLMs transition from research prototypes to production systems serving millions of users.

Most state-of-the-art LLMs adopt the decoder-only Transformer architecture [7], whose forward computation during inference can be conceptually divided into two sequential stages: the prompt prefill stage and the auto-regressive decoding stage [8]. In the prompt prefill stage, the model processes the entire input sequence in one pass, allowing all Transformer layers to operate over all tokens in the prompt simultaneously. In the auto-regressive decoding stage, the model generates output tokens one at a time, with each new token conditioned on all previously generated tokens. This computational pattern is dictated by the self-attention mechanism, which requires every token to attend to all tokens before it. To mitigate redundant computation in the decoding stage, modern serving systems employ the KV Cache [9, 10] to store intermediate attention states of past tokens, enabling efficient reuse across steps. These two stages lead naturally to several key performance metrics for evaluating LLM serving. The Time to First Token (TTFT) measures the latency from request arrival to the generation of the first token. TTFT is mainly determined by the prompt prefill stage. The Time Per Output Token (TPOT) [11] reflects the average time required to generate each subsequent token during decoding. The end-to-end latency captures the total time from request submission to the completion of the entire output sequence.

To accelerate LLM serving, recent research has proposed a number of system level optimizations, among which two representative techniques are prefill/decoding (PD) disaggregation and prefix caching. PD disaggregation, as adopted by systems such as DistServe [11] and Splitwise [12], assigns the prompt prefill stage and the auto-regressive decoding stage to different sets of GPUs. This design eliminates interference between prefill and decode by preventing the two stages, whose resource demands differ significantly, from competing on the same device.

On the other hand, prefix caching, as implemented in systems like SGLang [13] and vLLM [14], stores the KV Cache corresponding to common prompt prefixes and reuses it across requests. Combined with a prefix cache aware router that dispatches requests to prefill instances with the highest cache hit rates, this approach can substantially reduce redundant computation in the prefill stage and improve throughput in high load scenarios.

However, both strategies exhibit fundamental limitations under realistic, dynamic workloads:

1. **Static configurations lead to resource underutilization.** Modern LLM serving systems frequently exhibit sub-optimal resource usage under static allocation policies, driven by two primary factors. First, at low request rates (RPS $\leq$ 10), substantial portions of computational and memory capacity remain idle. As shown in Figure 1, empirical measurements reveal that Hugging Face Transformers (HFT) [15] and vLLM [14] leave approximately 20%–40% of GPU resources unused under these conditions. Second, a fundamental mismatch between fixed resource allocations and the dynamic workload characteristics of LLM serving further exacerbates inefficiency. In traditional designs, meeting storage requirements often constrains available compute capacity, and vice versa, preventing balanced utilization across resource types. This persistent allocation–demand disparity motivates our design of a dynamic resource allocation mechanism to enhance the cost-effectiveness of deployment.
2. **Prefix cache aware router induces severe load imbalance.** As shown in Figure 2a, a representative scenario illustrates how a prefix cache aware router allocates incoming requests across three serving instances based on both cache hit rate and current load. Although this policy seeks to balance computation with cache locality, it systematically

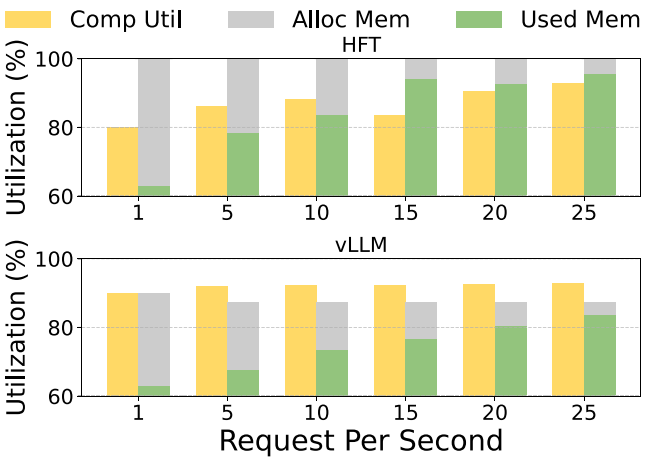


FIGURE 1 | GPU resource utilization comparison between HFT and vLLM serving frameworks across different request rates, conducted with a single LLaMA-13B instance deployed on an A100 GPU, with each test repeated five times.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

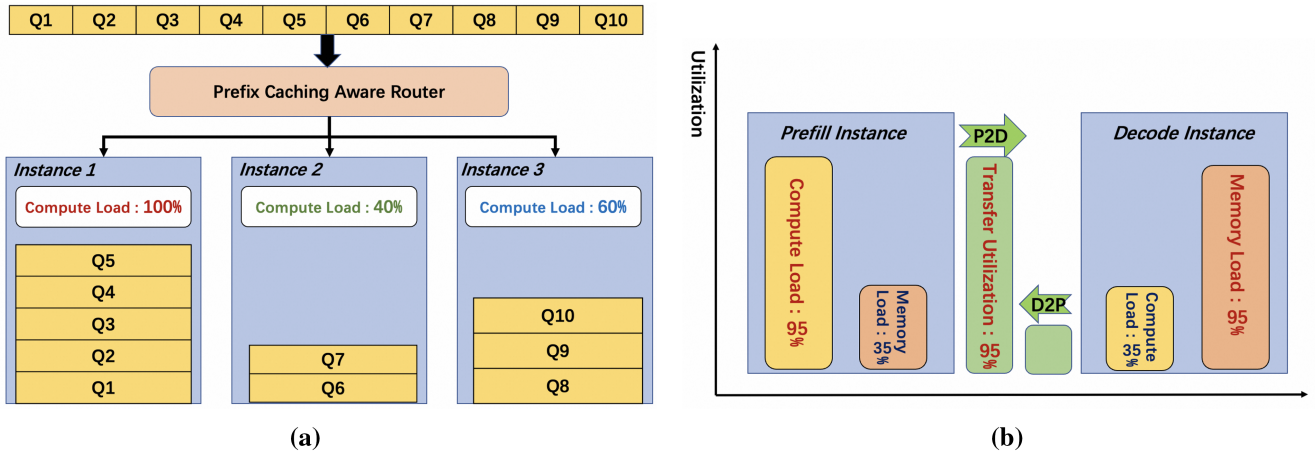


FIGURE 2 | Two empirically validated limitations in current LLM serving architectures. (a) Load imbalance caused by a prefix cache aware router. Requests associated with popular cached prefixes are preferentially directed to high-hit-rate instances, such as Instance 1, which operates at full compute load (100%) and stores prefixes Q1–Q5. Instance 2 remains at lower utilization (40%) while holding redundant entries Q6–Q7 that appear in other caches. Instance 3 sustains 60% load but recomputes uncached prefixes Q8–Q10. This allocation pattern leads to persistent skew, redundant storage, and repeated computation. (b) Resource utilization asymmetry in PD disaggregated architectures measured using the DistServe framework with a LLaMA-13B model on the Alpaca dataset. Prefill instances processing long prompts sustain high compute utilization (approximately 95%) but low memory usage (approximately 35%), whereas decode instances exhibit the opposite pattern with lower compute utilization (approximately 35%) and high memory usage. Communication is dominated by one-way KV Cache transfers from prefill to decode, leaving other channels underutilized.

favors instances with high prefix cache hit rates, resulting in persistent load skew and inefficient resource usage.

In this example, Instance 1 achieves the highest cache hit rate and receives the majority of incoming queries. It operates at full compute capacity (100%) while holding a large set of cached prefixes (Q1–Q5), reinforcing its scheduling priority. Instance 2 receives fewer queries and maintains low compute utilization (40%), its cached prefixes (Q6–Q7) also reside in other instances, creating redundant KV Cache storage across the cluster. Instance 3 sustains moderate compute utilization (60%) but primarily serves uncached prefixes (Q8–Q10), which must be recomputed due to their absence in high-priority caches, thereby incurring additional latency and forfeiting cache benefits.

This allocation pattern highlights a positive-feedback dynamic: instances with high hit rates attract more requests and expand their cache content, while lower-hit-rate instances process fewer queries yet store duplicate data or perform repeated computations. Over time, this imbalance increases queuing delays and degrades cache efficiency on overloaded instances, while under-utilized devices fail to contribute fully to overall system throughput.

3. **PD disaggregation introduces intrinsic load imbalance.** As shown in Figure 2b, this imbalance is derived from empirical measurements conducted in our experimental environment. Specifically, we deployed the DistServe [11] with a LLaMA-13B [2] model, using the publicly available Alpaca [16] dataset to generate inference workloads. We instrumented both the prefill and decode instances to record compute and memory utilization during execution.

   The results show that the prefill stage typically processes long prompt sequences, making it highly compute intensive (often sustaining >95% compute utilization) but relatively light on memory utilization (around 35%). In contrast, the decode stage executes iterative auto-regressive generation and leverages the KV Cache generated during prefill to reduce computation, leading to lower compute utilization (around 35%) but significantly higher memory utilization. This asymmetry is further reflected in inter-stage communication: prefill instances must transmit KV Cache data and next-token outputs to decode instances, resulting in predominantly one-way bandwidth usage from prefill to decode. Such empirically observed load imbalance in PD disaggregated architectures limits effective resource utilization across devices, leaving one resource dimension underused while the other becomes a performance bottleneck.

These limitations share a common root cause: the tight coupling between resource allocation and state (cache) placement in current architectures. In PD disaggregation, computational and memory resources are statically partitioned across prefill and decode instances, but the state required to execute a stage, such as weights for computation or KV Cache for memory, cannot be easily relocated without incurring significant overhead. As a result, adjusting resources dynamically in response to workload shifts becomes impractical, because moving computation inevitably entails moving large volumes of associated state. Similarly, cache aware routing is constrained by locality: to maximize cache hit rates, it must direct requests to instances holding the relevant prefixes, even if those instances are overloaded. This perpetuates load imbalance, as high hit rate nodes continuously attract more traffic while low hit rate nodes remain underutilized. Together, these constraints prevent current systems from simultaneously achieving high resource utilization and balanced load, particularly under dynamic and unpredictable workloads. Although prior PD-disaggregated systems such as Splitwise [12] and NVIDIA Dynamo [17] adjust prefill–decode resources at

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the granularity of GPU pools or instances, their designs preserve the coupling between computation and KV state placement. Consequently, routing remains constrained by cache locality, and coarse-grained resizing cannot address the fine-grained compute–memory asymmetry that arises under dynamic workloads. By contrast, BanaServe decouples state management from computation through layer-level and attention-level migration together with a Global KV Cache Store, enabling sub-layer online rebalancing and more responsive adaptation to workload fluctuations.

To address these challenges, we present **BanaServe**, a dynamic orchestration framework that decouples resource allocation from state management through three key innovations. First, BanaServe introduces *fine-grained resource migration* mechanisms that include layer-level weight migration, attention-level KV Cache migration, and selective computation offloading, enabling rapid rebalancing between prefill and decode instances without service disruption. Second, BanaServe employs a *Global KV Cache Store* accessible to all prefill instances, eliminating cache locality constraints on routing decisions. Third, BanaServe implements *layer-wise overlapped transmission* that hides communication latency by pipelining KV Cache transfers with layer computation, making cache sharing practical at scale.

These mechanisms work synergistically to enable truly adaptive serving: the orchestrator continuously monitors system load and dynamically redistributes resources at fine granularity, while the router makes purely load-aware scheduling decisions unconstrained by cache placement. This design philosophy of separating mechanism (resource allocation) from policy (routing) allows BanaServe to achieve both high efficiency and robust performance under diverse workload conditions.

Our main **contributions** are summarized as follows:

1. We reveal two intrinsic limitations of current LLM serving systems: (i) static resource configurations fail to adapt to non-stationary workloads; (ii) inherent load imbalance between prefill and decode stages, where prefill is compute bound and decode is memory bound, causing under-utilization in one tier while the other becomes a bottleneck; and (iii) prefix cache aware routing inherently skews load distribution.
2. We design and develop **BanaServe**, a dynamic orchestration system for LLM serving that decouples resource allocation from state management. The system architecture enables adaptive rebalancing between prefill and decode instances under highly dynamic workloads.
3. Within BanaServe, we present several key mechanisms: dynamic weight migration, KV Cache offloading, and attention computation offloading for fine-grained resource balancing, and a Global KV Cache Store with layer-wise transmission to support cache sharing and communication–computation overlap.
4. We implement BanaServe and evaluate it with 13B-parameter models on production workload traces. Compared to vLLM [14], BanaServe achieves up to 3.9× higher throughput with 78.4% lower latency, and compared to DistServe [11], it delivers 2.8× throughput improvement with 70.1% latency reduction, while maintaining robust performance across both short-context (Alpaca [16]) and long-context (LongBench [18]) scenarios.

2 | Background

To understand the challenges addressed by BanaServe, we first outline the computational characteristics of prefill and decode in LLM inference, then evaluate limitations of both co-located and disaggregated deployment architectures.

2.1 | Characteristics of Prefill and Decode Operations

Prefill and decode phases in LLM inference have fundamentally different execution characteristics and resource requirements. During the *prefill* phase, the model processes the entire input prompt in parallel across the sequence dimension [19, 20], enabling high degrees of parallelism in matrix multiplications and attention computation. This stage is thus *compute-bound*, saturating GPU cores and delivering predictable runtimes primarily determined by input length and model size.

In contrast, the *decode* phase generates new tokens sequentially in an autoregressive fashion, making each step dependent on the previous output. This inherently limits parallelism, shifting the bottleneck from compute throughput to *memory-bound*. The decode stage involves frequent accesses and updates to the KV Cache, resulting in irregular and memory-intensive access patterns. Furthermore, output lengths vary across requests, introducing uncertainty and making decode workloads harder to predict and balance.

2.2 | Inefficiencies of Co-Located Prefill and Decode

The co-location of prefill and decode on the same GPU instances causes fundamental inefficiencies. First, the mismatch in computational characteristics leads to resource contention: compute-bound prefill interferes with memory-bound decode, inflating latency for both. When prefill saturates GPU compute resources, decode requests are delayed, while decode-heavy intervals leave computational throughput underutilized.

Second, co-location architectures [21] often maintain a one-to-one mapping between prefill and decode threads for pipeline simplicity. However, because TTFT is often orders of magnitude larger than TPOT, the slower prefill phase throttles overall throughput and prevents decode from reaching full utilization.

Finally, resource utilization patterns exhibit substantial volatility throughout the inference lifecycle. During the prefill phase, both memory consumption and compute utilization peak due to attention computation and KV Cache initialization. However, once the system transitions to decode, memory usage patterns shift dramatically and compute requirements decrease

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

significantly. These opposed patterns make it difficult to provide a fixed resource allocation that avoids both under-utilization and bottlenecks.

2.3 | Imbalanced Utilization in PD Disaggregation

While PD disaggregation addresses the interference issues of co-located serving, it introduces new challenges related to resource imbalance between prefill and decode instances. Previous work, exemplified by Taming [22] and Blitzscale [23], has documented substantial differences in resource utilization across dedicated prefill and decode clusters. In particular, GPU utilization monitoring in Sarathi-Serve [22] shows that prefill instances often operate at high capacity during peak periods but experience substantial idle time during low-traffic intervals. Conversely, decode instances may become bottlenecks when handling long-running generation tasks, even as prefill instances remain underutilized.

Our own measurements (Figure 2b) on representative LLM workloads corroborate these findings, revealing similar asymmetries in both compute and memory utilization. Prefill instances require substantial memory for processing large input sequences and initializing KV Cache, but this memory becomes available once requests are transferred to decode instances. Decode instances, meanwhile, accumulate KV Cache memory over time as generation proceeds, creating asymmetric memory pressure across the system. This imbalance manifests in scenarios where prefill instances may be memory-constrained while decode instances have abundant free memory, or vice versa.

2.4 | Limitations of Static PD Disaggregated Configuration Under Dynamic Workloads

Current PD disaggregation architectures rely on predetermined, static configurations that fail to adapt to the dynamic nature of real-world workloads. During periods of low request rates, the system suffers from resource over-provisioning, with dedicated prefill and decode instances remaining underutilized. In contrast, during high-traffic periods, the static allocation becomes a throughput bottleneck, unable to scale beyond the predetermined capacity limits.

Sudden traffic spikes present particularly challenging scenarios for static configurations. Bursty request patterns can overwhelm individual prefill or decode nodes, potentially leading to system failures or severe performance degradation. The lack of elasticity in static systems means that temporary traffic surges cannot be absorbed by redistributing load across available resources, resulting in poor fault tolerance and service reliability.

The operational overhead of adjusting prefill and decode instance configurations is prohibitively expensive in static systems. Scaling operations typically require service restarts, model reloading, and careful coordination to maintain consistency. This rigidity prevents systems from responding quickly to changing workload patterns and limits the ability to optimize resource allocation based on real-time performance metrics.

3 | Related Work

In this section, we review prior work on phase-disaggregated system architectures, with a focus on their application to LLM serving systems. We also survey complementary research directions, including request scheduling strategies and dynamic load balancing techniques, which collectively address key challenges in distributed inference.

3.1 | LLM Serving

Recent advances in LLMs serving aim to reduce inference latency and improve hardware utilization for models with billions of parameters, which must process long-context prompts and perform autoregressive decoding under strict SLOs. In this setting, the dual-phase execution pipeline, comprising the compute bound prefill phase and the memory bound decode phase, creates challenges for sustaining high throughput, minimizing tail latency, and efficiently managing KV Cache memory.

Several representative systems have been proposed to improve inference efficiency within monolithic serving architectures. vLLM [14] introduces PagedAttention, a block-level KV Cache management mechanism that reduces memory fragmentation and enables efficient batched decoding [21] for dynamic, variable-length sequences. SGLang [13] focuses on optimizing prompt processing through prefix caching and a cache-aware scheduling policy, which routes requests to maximize KV Cache hit rates and thereby reduce redundant computation in the prefill phase. DeepSpeed-FastGen [24] builds on DeepSpeed-Inference by incorporating optimizations for high-throughput, low-latency text generation, including enhanced kernel fusion, improved batching strategies, and adaptive communication scheduling to maximize performance across diverse workload patterns. TensorRT-LLM [25] incorporates graph-level compilation, fused attention kernels, and quantization-aware execution to deliver low-latency, high-throughput Transformer inference on NVIDIA GPUs.

While these systems significantly improve kernel-level efficiency, they adopt a monolithic execution model and do not explicitly address the compute–memory divergence present in modern LLM inference. Their optimization objectives focus on kernel fusion, graph compilation, or memory reuse, rather than on redistributing heterogeneous workload between prefill and decode stages. Such limitations motivate the need for disaggregated architectures that can address phase-level heterogeneity directly.

3.2 | Disaggregated Serving Systems

Traditional phase-wise unified systems typically co-locate both the prefill and decode phases within monolithic architectures, often leading to significant interference due to the divergent characteristics of these stages. Recent research has revealed fundamental differences between the prefill and decode phases, prompting the development of novel phase-disaggregated system designs. DistServe [11] proposes this approach through GPU-level disaggregation, dynamically allocating separate GPU

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

devices for each phase while introducing parallelism adjustments to optimize resource utilization. Building on this, LoongServe [26] further refines the paradigm through sequence parallelism and flash decoding [27], significantly improving throughput for long-context scenarios. The trade-off for these compute-centric approaches lies in their operational complexity, as they require careful synchronization between disaggregated components and may incur overhead during phase transitions. TetriInfer [28] introduces a two-level scheduling mechanism enhanced with predictive resource profiling, effectively preventing decode-phase scheduling bottlenecks through anticipatory workload distribution.

Beyond compute-centric disaggregation, cluster-level systems explore orchestration mechanisms for large-scale heterogeneous deployments. NVIDIA Dynamo represents a cluster-level PD-serving orchestration framework whose SLA Planner dynamically adjusts the ratio of prefill to decode resources. However, its reconfiguration granularity remains at the GPU-pool level, assuming co-location of computation and KV state. This design leaves routing implicitly constrained by cache locality and provides limited support for sub-stage rebalancing under bursty or heterogeneous traffic.

As cluster sizes expand, disaggregated architectures are becoming increasingly popular in distributed scenarios, with recent work contributing to this area. Taking a more radical approach, Splitwise [12] advances system disaggregation by partitioning and distributing distinct processing phases across dedicated machines, rather than merely separating GPU devices. This architectural strategy facilitates phase-specific resource allocation, enabling each computation stage to leverage hardware optimally suited to its particular requirements.

Although Splitwise and NVIDIA Dynamo both address coarse-grained resource configuration between prefill and decode stages, their adjustments occur only at the instance or pool granularity. Splitwise explores static or periodically optimized P:D ratios but lacks mechanisms for fine-grained, intra-stage rebalancing. NVIDIA Dynamo performs SLA-driven online adjustments, but its scheduling remains tightly coupled with KV locality, and its reconfiguration latency prevents real-time correction of load skew caused by asymmetric workloads. In contrast, BanaServe introduces layer-level and attention-level module migration and a Global KV Cache Store that decouples routing from cache placement, enabling real-time sub-layer rebalancing that neither Splitwise nor Dynamo can achieve.

Mooncake [29] and MemServe [30] explore distributed memory management approaches, with Mooncake building a KVCache-centric disaggregated architecture through a distributed KV Cache pool and scheduling based on different SLOs for clusters at various stages. Meanwhile, MemServe completes an elastic memory pool for managing memory across instances. Moreover, KVDirect [31] proposes a tensor-centric communication mechanism and library for different machines, utilizing Remote Direct Memory Access (RDMA) technology to facilitate KV transmission between distributed machines. These memory-centric approaches highlight that scalable KV management is essential for PD serving. However, distributed KV designs inevitably introduce remote lookup cost, metadata management, and consistency coordination, especially under highly dynamic workloads. These challenges motivate architectural choices that reduce the scheduling constraints imposed by KV placement.

Considering that stream-based disaggregated architectures are an emerging system design, WindServe [32] proposes separating prefill and decode tasks onto different streams, utilizing fine-grained dynamic scheduling based on streams to enhance resource utilization and performance. In contrast, semi-PD [33] attempts to introduce Streaming Multiprocessor (SM)-level phase separation and further designs a dynamic partitioning algorithm based on SLO awareness to maximize the inherent parallelism of GPUs.

The existing disaggregated architectures face several drawbacks. One significant issue is the imbalance in resource utilization, particularly with the prefill phase, which often leads to wasted GPU memory. This inefficiency arises because the resources allocated for prefill may not be fully utilized, resulting in suboptimal performance. Additionally, the migration of requests between the prefill and decode phases can introduce stalls, further degrading overall system throughput and responsiveness. While distributed approaches emphasize the transmission of KV Cache data between machines or instances, they frequently overlook the impact of model reconfiguration. This oversight makes it challenging to handle sudden surges in requests, as the system may not adapt quickly to changing workload demands. Moreover, stream-based methods impose specific hardware requirements that may limit their applicability across different environments, potentially restricting their scalability and flexibility.

While Mooncake adopts a distributed KV Cache pool to support cross-node memory disaggregation, its design inherently requires the scheduler to consider cache locality and remote lookup overhead. As a result, nodes storing popular prefixes may still become hotspots, and prefix-aware routing cannot be fully decoupled from state placement. In contrast, BanaServe employs a centralized Global KV Cache Store together with layer-wise overlapped KV transmission, making per-layer KV access latency negligible relative to prefill computation. This enables routing decisions to depend solely on real-time load rather than cache placement, effectively eliminating prefix-cache-induced load skew. Furthermore, the centralized KV view simplifies correctness during dynamic layer-level and attention-level migration, avoiding the metadata coordination and consistency management required in distributed KV stores. These differences reflect distinct system objectives: whereas Mooncake emphasizes large-scale memory disaggregation, BanaServe prioritizes fine-grained resource rebalancing and low-latency PD-disaggregated serving.

3.3 | Load Balancing Scheduling

Load balancing scheduling techniques are essential in disaggregated architectures, as both are aimed at reducing SLO violation rates. In modern distributed systems, effective load balancing is crucial for optimizing resource utilization and ensuring that all components operate efficiently. By strategically distributing workloads across various processing units or nodes, load balancing prevents any single unit from becoming a bottleneck, which is particularly important when different phases of processing, such as prefill and decode, are handled by separate units.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Intelligent routing of requests based on current load and resource availability helps maintain consistent performance and meet SLOs. Notable approaches in this domain include Llumnix [34], which focuses on dynamic load balancing by continuously monitoring the performance of different nodes and redistributing tasks as needed to maintain equilibrium. SpotServe [35] takes a cost-effective approach by leveraging spot instances in cloud environments, optimizing resource allocation while managing the inherent volatility of these instances. UELLM [36] implements an adaptive load balancing strategy that adjusts dynamically to changing workloads, enhancing the system's resilience and ability to handle unexpected spikes in demand.

However, existing load balancing strategies also have notable limitations when applied to prefill and decode disaggregated LLM serving. Most prior approaches are designed for homogeneous workloads and do not explicitly address the intrinsic compute and memory imbalance between prefill and decode stages. Moreover, they typically assume that workload distribution decisions can be made independently of state placement, overlooking the constraints imposed by large KV Cache locality. As a result, cache-aware routing may still lead to load skew, where high-hit-rate nodes become overloaded while other nodes remain underutilized. These gaps motivate the need for new scheduling mechanisms that are both state-aware and capable of dynamically rebalancing heterogeneous resources across stages. These observations motivate the need for unified orchestration that jointly considers state placement, compute–memory heterogeneity, and dynamic balancing across phases. BanaServe addresses this gap by enabling fine-grained online rebalancing and cache-decoupled routing, which together provide stronger adaptability to non-stationary workloads.

3.4 | Summary

Existing LLM serving frameworks, including vLLM, SGLang, DeepSpeed-FastGen, TensorRT-LLM, and NVIDIA Dynamo, focus on kernel-level optimizations, KV Cache management, and execution graph compilation within monolithic architectures. While these systems achieve substantial performance gains, their tightly coupled execution of the prefill and decode phases exacerbates resource contention and limits flexibility under dynamic workloads. Phase-disaggregated architectures, such as DistServe, LoongServe, TetriInfer, Splitwise, Mooncake, KVDirect, WindServe, semi-PD, and MemServe, mitigate prefill and decode interference through physical or logical separation of phases. However, they introduce new challenges, including load imbalance, resource fragmentation, inter-phase migration stalls, and scalability limitations under heterogeneous hardware conditions. Complementary research on load balancing scheduling, such as Llumnix, SpotServe, and UELLM, optimizes request distribution but typically targets generic distributed inference and does not specifically address the unique constraints of LLM phase separation and prefix caching.

Table 1 summarizes representative systems and contrasts their characteristics with BanaServe. In particular, BanaServe addresses two limitations that remain unresolved by prior work. First, it uses a unified KV Cache store to eliminate the load imbalance caused by prefix-caching-aware routing in the prefill phase, enabling routing decisions to be based solely on node load. Second, it introduces fine-grained module migration at the layer and attention computation levels, facilitating dynamic resource rebalancing between prefill and decode phases to adapt to fluctuating request patterns.

4 | System Design

In this section, we present the architectural and algorithmic components of **BanaServe**, a LLM serving framework designed for PD disaggregated architecture in AI infrastructure. We first describe the Dynamic Migration mechanism for computational modules, which enables fine and coarse grained redistribution of workloads between GPUs in real time. We then introduce the Global KV Cache Store, a unified cache management layer that allows cross-instance KV Cache sharing and substantially simplifies scheduling logic. Building upon these designs, we formulate analytical models for PD disaggregation performance to guide system optimization. Finally, we detail two complementary runtime algorithms, Dynamic Migration and Load-aware Request Scheduling, which together maintain balanced utilization, minimize latency, and sustain high throughput under dynamic workload conditions.

4.1 | Dynamic Migration for Modules

To address the inherent inefficiencies of static PD disaggregation architectures, we propose a **dynamic resource allocation paradigm** based on module migration. This design enables the scheduler to adaptively redistribute computational modules, ranging from entire transformer blocks to portions of the KV Cache, across GPUs in response to real-time workload conditions. By dynamically relocating modules between prefill and decode instances, our framework achieves superior utilization of both computation and memory resources.

We formulate the migration decision as an optimization problem that aims to minimize the peak utilization across all devices:

$$\min_{\mathcal{M}} \max_{g \in \mathcal{G}} U_g(\mathcal{M}), \tag{1}$$

where $\mathcal{G}$ is the set of available GPUs, $\mathcal{M}$ denotes the chosen module migration plan, and $U_g(\mathcal{M})$ is the utilization of GPU g under $\mathcal{M}$.

Migration feasibility is constrained by a per-orchestration budget on latency overhead T_{budget}:

$$T_{\text{mig}}(\mathcal{M}) \leq T_{\text{budget}}, \tag{2}$$

where T_{mig} denotes the migration time, composed of *weight transfer* and *state transfer* latency.

We design and implement two complementary migration granularities, providing a flexible trade-off between rebalancing precision and migration overhead:

(1) Layer-level migration (Figure 3): In scenarios where workload imbalance is pronounced, we migrate a contiguous set of

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 | Comparison of representative LLM serving systems and **BanaServe**.

System	Optimization focus	Load imbalance	Prefix cache strategy
vLLM [14]	PagedAttention; KV Cache memory optimization	Not addressed	Per-instance KV reuse
SGLang [13]	Prefix caching; Cache-affinity routing	Not addressed (can cause hotspots)	Per-instance prefix cache
DeepSpeed-FastGen [24]	Adaptive communication scheduling; kernel fusion; Improved batching	Not addressed	None
TensorRT-LLM [25]	Graph compilation; Fused kernels; Quantization	Not addressed	None
NVIDIA Dynamo [17]	Distributed orchestration on TensorRT-LLM	Not addressed	None
DistServe [11]	GPU-level PD disaggregation; Parallelism tuning	Static PD allocation; coarse-grained balancing	None
LoongServe [26]	Sequence parallelism; Flash decoding	Static allocation	None
Mooncake [29]	Distributed KV Cache pool; SLO-aware scheduling	Limited balancing	Shared KV pool
MemServe [30]	Elastic memory pool across instances	Limited balancing	None
KVDirect [31]	RDMA-based KV Cache transmission	Limited balancing	Shared KV framework
BanaServe (ours)	**Global KV Cache Store; Fine-grained migration**	**Dynamic PD rebalancing at sub-layer granularity**	**Shared KV Cache across all prefill instances**

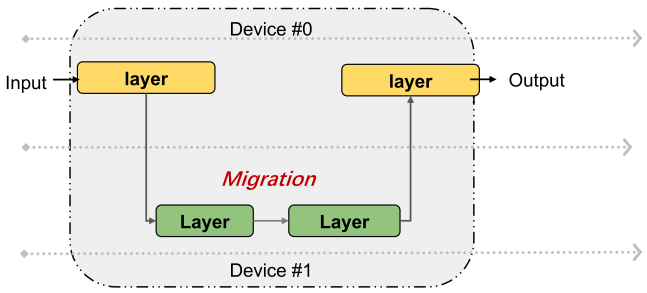


FIGURE 3 | Layer level migration in BanaServe: A set of adjacent Transformer layers is dynamically reallocated across prefill and decode GPUs, with both layer weights W_ℓ and KV Cache $\mathcal{KV}_\ell$ migrated. Local execution resumes after payload transfer, enabling coarse grained compute and memory rebalancing while preserving output correctness. Device #0 and Device #1 process different segments in parallel until migration completes, avoiding service disruption.

Transformer layers from one GPU instance to another. This *coarse grained* reallocation realizes *dynamic model parallelism*, shifting both **layer weights** and the associated **KV Cache** to redistribute substantial compute and memory footprint between prefill and decode stages.

Data volume to migrate: Let S_ℓ^{w} be the size of the migrated layer weights and S_ℓ^{kv} be the size of the corresponding KV Cache state. The total payload is:

$$S_\ell^{\mathrm{total}} = S_\ell^{\mathrm{w}} + S_\ell^{\mathrm{kv}}. \tag{3}$$

Latency model: Given effective interconnect bandwidth B_{net} and synchronization overhead T_{sync}, the migration latency is approximated by:

$$T_{\mathrm{layer}} \approx \frac{S_\ell^{\mathrm{total}}}{B_{\mathrm{net}}} + T_{\mathrm{sync}}. \tag{4}$$

Since $S_\ell^{\mathrm{w}} \gg S_\ell^{\mathrm{kv}}$ in most cases, the latency is dominated by weight transfer, but still remains practical within high bandwidth data center fabrics (NVLink, Infiniband).

Execution correctness: For a migrated layer $f_\ell(\cdot)$ with weights W_ℓ and KV Cache $\mathcal{KV}_\ell$, the same computation can be carried out on the new GPU without semantic change:

$$\mathbf{y}_\ell = f_\ell(\mathbf{x}_\ell; W_\ell, \mathcal{KV}_\ell), \tag{5}$$

where $\mathbf{x}_\ell$ is the input activation from the preceding layer. By transferring $(W_\ell, \mathcal{KV}_\ell)$ together, we ensure that auto-regressive decoding continues seamlessly after migration.

While this method incurs temporary weight relocation cost, it can rapidly mitigate severe imbalance by shifting large computational segments, and is particularly effective in bursty or highly skewed workloads.

(2) Attention-level migration (Figure 4): When fine-grained load adjustment is required, we partition the KV Cache along the attention head dimension and selectively migrate only the KV states of certain heads to auxiliary *cold* GPUs. Since no model

10970243x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

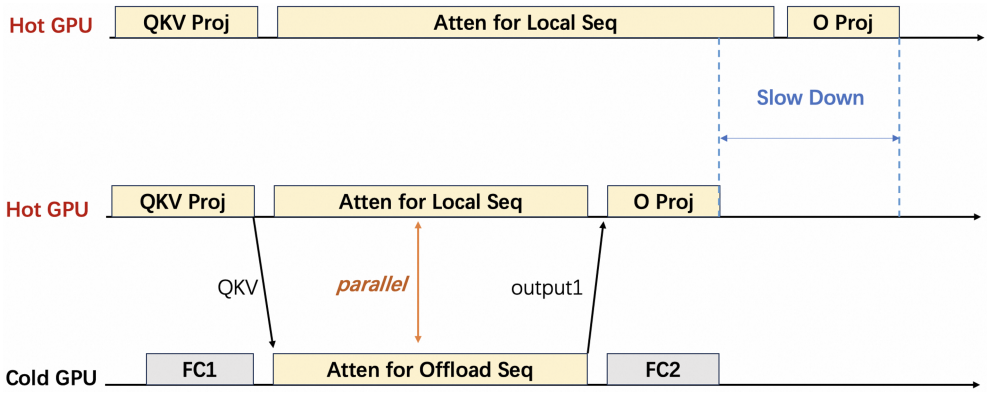


FIGURE 4 | Attention level migration in BanaServe. KV Cache is split along the attention head dimension into $K^{(1)}, V^{(1)}$ (hot GPU) and $K^{(2)}, V^{(2)}$ (cold GPU). Local attention (*Atten for Local Seq*) and offloaded attention (*Atten for Offload Seq*, preceded by FC1 and followed by FC2) are computed in parallel. Only the partial softmax denominator $\ell^{(1)}$ and output $O^{(1)}$ are exchanged across devices, enabling minimal data transfer and mitigating hotspots with negligible service disruption.

weights are transferred and only intermediate key and value activations are moved, this method imposes minimal migration overhead.

The pipeline begins with the *QKV Projection* layer, which generates query (Q), key (K), and value (V) matrices from the input hidden states. We split the attention heads into two disjoint subsets:

- $K^{(1)}, V^{(1)}$ ($\mathbf{KV}_1$): retained on the hot GPU for local processing.
- $K^{(2)}, V^{(2)}$ ($\mathbf{KV}_2$): offloaded to a cold GPU for remote processing.

The local device executes *Atten for Local Seq* using $K^{(1)}, V^{(1)}$, while the cold device executes *Atten for Offload Seq* using $K^{(2)}, V^{(2)}$ in parallel. Blocks **FC1** and **FC2** are lightweight fully connected layers bracketing the offloaded attention on the cold GPU, and *O Proj* is the output projection of the multi head attention (MHA).

Parallel computation feasibility: Let H be the total number of attention heads, d the head dimension, $Q \in \mathbb{R}^{B_q \times Hd}$ the query matrix, and $K^{(j)}, V^{(j)} \in \mathbb{R}^{B_k \times d_{h_j}}$ the key/value subsets assigned to device $j \in \{1, 2\}$. The attention score and value aggregation for each subset are:

$$S^{(j)} = Q \cdot (K^{(j)})^{\top}, \tag{6}$$

$$A^{(j)} = \exp\left(S^{(j)}\right), \tag{7}$$

$$\ell^{(j)} = \begin{cases} \sum_i A_i^{(1)}, & j = 1, \\ \ell^{(1)} + \sum_i A_i^{(2)}, & j = 2, \end{cases} \tag{8}$$

$$O^{(j)} = \frac{A^{(j)}}{\ell^{(j)}} \cdot V^{(j)}. \tag{9}$$

The final attention output is:

$$O = O^{(1)} + O^{(2)}. \tag{10}$$

Since $K^{(1)}, V^{(1)}$ and $K^{(2)}, V^{(2)}$ are from disjoint head partitions, $S^{(1)}$ and $S^{(2)}$ can be computed entirely in parallel on separate devices without intermediate data dependencies. Only $\ell^{(1)}$ needs to be communicated to the cold GPU before its normalization step, and $O^{(1)}$ (or $\ell^{(2)}$) is sent back before the final summation, ensuring correctness of the global softmax while keeping transfer size minimal.

Let S_{kv} denote the size of migrated KV data and B_{net} is the network bandwidth, thus the migration latency is:

$$T_{\text{attn}} \approx \frac{S_{\text{kv}}}{B_{\text{net}}}. \tag{11}$$

Typically $S_{\text{kv}} \ll S_{\ell}$ (layerwise state size), hence $T_{\text{attn}} \ll T_{\text{layer}}$. This makes attention level migration lightweight, latency friendly, and effective in real time load rebalancing for heterogeneous decoding workloads.

By combining coarse grained layer migration with fine grained attention migration, BanaServe can adaptively select at runtime the strategy that balances migration cost (T_{mig}) against the expected gain in utilization (minimizing $\max_g U_g$), under latency budget constraint $T_{\text{mig}} \leq T_{\text{budget}}$.

4.2 | Global KV Cache Store

The primary bottleneck of the prefix cache aware router lies in the imbalance of prefix cache storage across prefill nodes. Since the largest prefix cache segment cannot be shared among different prefill nodes, the router must consider both computation and cache placement when dispatching requests. To address this problem, we introduce a **Global KV Cache Store** that spans both prefill and decode nodes. With this design, all prefill nodes access a shared KV Cache, enabling the router to focus solely on balancing computational workload. All prefix cache `load` and `fetch` operations are delegated to the KV Cache Store (Figure 5).

A major challenge in implementing a Global KV Cache Store is the latency incurred by repeated `load` and `fetch` operations. To mitigate this, we leverage the **layer-wise** execution

10970243x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

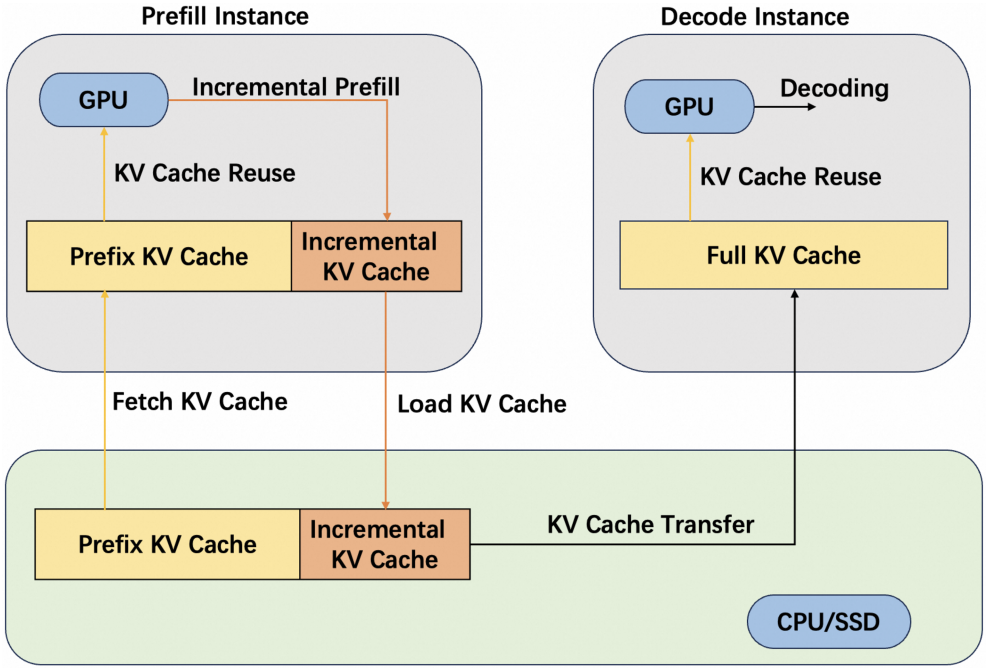


FIGURE 5 | KV Cache management and reuse in BanaServe. Prefill instances perform incremental prefill on incoming requests, reusing cached prefixes when available. Prefix KV Cache and newly generated incremental KV Cache are stored in a shared CPU/SSD-backed KV store, enabling both cache sharing across prefill GPUs and complete KV Cache assembly. During decoding, decode instances retrieve the full KV Cache from the shared store and reuse it for token generation, avoiding redundant computation and supporting efficient prefill and decode execution.

property of transformer models to design a **three-stage pipeline** where prefetching, computation, and loading are overlapped. This overlap hides most of the memory access latency, resulting in near-transparent prefill computation without observable throughput degradation.

We let T_F denote the total forward computation time for the prefill phase, r is the average prefix cache hit rate, L is the input sequence length in tokens, and N denotes the total number of Transformer layers. The per-layer forward computation time is then

$$T_{F,layer} = \frac{T_F \cdot r}{N}. \tag{12}$$

Similarly, let S_{kv} be the per-layer KV Cache size for a single token (in bytes), and B is the effective PCIe bandwidth. The per-layer KV Cache transfer time is

$$T_{KV} = \frac{S_{kv} \cdot L \cdot r}{B}. \tag{13}$$

For example, in a `llama-3.1-8B` model with hidden size $d_{model} = 4096$, total attention heads $h_{total} = 32$, KV heads under GQA $h_{kv} = 8$, and BF16 precision (2 bytes per element), the dimension per head is

$$d_{head} = \frac{d_{model}}{h_{total}} = 128. \tag{14}$$

Because each KV head stores both *key* and *value* activations, the per-layer KV Cache size is

$$\begin{aligned} S_{kv} &= h_{kv} \cdot d_{head} \cdot 2 \cdot 2 \\ \text{bytes} &= 8 \times 128 \times 2 \times 2 \text{ bytes} = 4096 \text{ bytes (4 KB)}. \end{aligned} \tag{15}$$

Multiplying by $N = 32$ layers gives the total KV Cache per token:

$$S_{kv,total} = 32 \times 4 \text{ KB} = 128 \text{ KB}. \tag{16}$$

In our evaluation with $L = 1000$ tokens, $r = 0.5$, $B = 200$ Gbps, and $T_F = 270$ ms, we have:

$$\begin{aligned} T_{F,layer} &= \frac{270 \text{ ms} \times 0.5}{32} \approx 4.22 \text{ ms}, \\ T_{KV} &= \frac{4 \text{ KB} \times 1000 \times 0.5}{200 \text{ Gbps}} \approx 0.082 \text{ ms}. \end{aligned} \tag{17}$$

Since $T_{KV} \ll T_{F,layer}$, the KV Cache transfer latency is negligible compared with computation time, enabling full overlap of communication and computation in the proposed pipeline. This overlap ensures that KV Cache fetch and store operations remain transparent to the serving workflow, sustaining high throughput without observable stalls in the prefill phase (Figure 6).

This analytic and empirical validation confirms that the three-stage pipeline design is effective in overlapping cache transfer with computation. Consequently, the Global KV Cache Store can serve prefix cache requests without introducing noticeable stalls in prefill execution, which is critical for sustaining high throughput in PD disaggregated architectures.

4.3 | Modeling PD Disaggregation Performance

We formulate the performance optimization in PD disaggregation systems as a multi-objective problem, aiming to *maximize resource utilization*, *minimize end-to-end latency*, and thereby *maximize throughput* across both prefill and decode stages.

10977024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

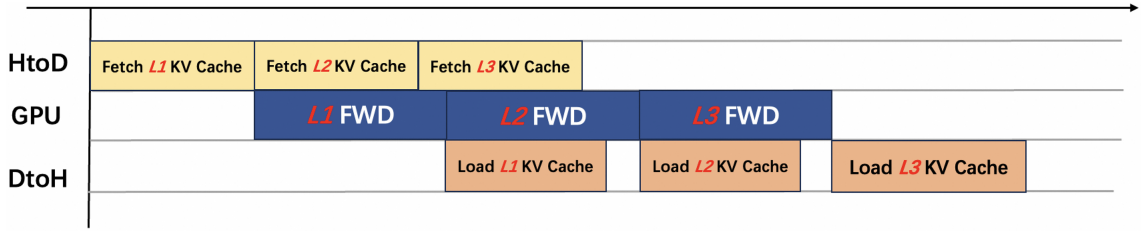


FIGURE 6 | Validation of the three-stage layer-wise KV Cache pipeline in BanaServe. The top section lists hardware parameters and measured latencies for a representative workload. Under a 50% average prefix cache hit rate, the per-layer forward time (4.22 ms) exceeds the per-layer KV Cache transfer time (0.082 ms), ensuring effective overlap. The timeline shows the execution for three layers ($L1-L3$): while the GPU executes forward computation for L_i, the host-to-device (HtoD) channel fetches KV Cache for L_{i+1} and the device-to-host (DtoH) channel stores cache from L_{i-1}. This concurrency hides communication latency and enables transparent prefill execution.

Let $\mathcal{G}_p$ and $\mathcal{G}_d$ denote the sets of prefill and decode GPUs respectively, and U_g the utilization of GPU g. Let T_{TTFT} denote the time to first token and T_{TPOT} is the time per output token. Let Θ denote the effective system throughput (tokens/sec).

4.3.1 | Optimization Objective

We define the joint optimization objective as:

$$\max_{\mathcal{M}} \ \alpha \cdot U_{\text{avg}}(\mathcal{M}) \ - \ \beta \cdot T_{\text{avg-latency}}(\mathcal{M}) + \gamma \cdot \Theta(\mathcal{M}), \tag{18}$$

where:

$$U_{\text{avg}}(\mathcal{M}) = \frac{1}{|\mathcal{G}_p \cup \mathcal{G}_d|} \sum_g U_g(\mathcal{M}),$$

$$T_{\text{avg-latency}} = \frac{1}{N} \sum_{i=1}^{N} T_{\text{TTFT}}^{(i)} + T_{\text{TPOT}}^{(i)}, \tag{19}$$

α, β, γ are weighting coefficients for utilization, latency, and throughput; $\mathcal{M}$ denotes the current module migration plan.

4.3.2 | Latency Model

The Time to First Token (TTFT) is modeled as:

$$T_{\text{TTFT}} = T_{\text{p}} + T_{\text{x}} + T_{\text{q}}, \tag{20}$$

where T_{p} is the prefill computation time, T_{x} is the KV Cache transfer time, and T_{q} is the queuing delay before decode. The KV transfer time is decomposed as

$$T_{\text{x}} = T_{\text{load}} + T_{\text{fetch}}, \tag{21}$$

with T_{load} denoting the time to load KV state from storage and T_{fetch} means the time to fetch KV state from remote GPUs.

For the decode phase, the time per output token (TPOT) is:

$$T_{\text{TPOT}} = T_{\text{d}} + T_{\text{c}} + T_{\text{m}}, \tag{22}$$

where T_{d} is the base decoding compute time, T_{c} is the cache access latency, and T_{m} is the memory bandwidth stall time.

4.3.3 | Resource Utilization Model

Consider a prefill instance with n_p Transformer layers. Let M_0 be the base memory overhead of the process, M_ℓ is the memory per layer, and K_{init} is the initial KV Cache allocation. The total memory footprint is

$$\text{Mem}_p = M_0 + n_p \cdot M_\ell + K_{\text{init}}. \tag{23}$$

Let C_ℓ be the compute cost per layer per token, B_{sz} is the batch size, and L_{in} denotes the input length in tokens. The total compute demand is

$$\text{Comp}_p = n_p \cdot C_\ell \cdot B_{\text{sz}} \cdot L_{\text{in}}. \tag{24}$$

Similarly, for a decode instance with n_d layers: let K_{acc} be the KV Cache size accumulated during decoding, and L_{gen} is the generation length in tokens. We have:

$$\text{Mem}_d = M_0 + n_d \cdot M_\ell + K_{\text{acc}}, \tag{25}$$

$$\text{Comp}_d = n_d \cdot C_\ell \cdot B_{\text{sz}} \cdot L_{\text{gen}}. \tag{26}$$

Given peak GPU compute capacity C_{gpu}, the average utilization of each stage is:

$$U_p = \frac{\text{Comp}_p}{C_{\text{gpu}}}, \quad U_d = \frac{\text{Comp}_d}{C_{\text{gpu}}}. \tag{27}$$

4.3.4 | Migration Cost

Migrating k modules between instances incurs total overhead:

$$\text{Cost}_{\text{mig}} = k\left(T_{\text{x\_lat}} + T_{\text{sync}} + T_{\text{mem\_realloc}}\right), \tag{28}$$

where $T_{\text{x\_lat}}$ is the KV+weight transfer latency, T_{sync} represents the synchronization time with ongoing computation, and $T_{\text{mem\_realloc}}$ denotes the time to reallocate buffers.

Module migration $\mathcal{M}$ must satisfy latency constraints:

$$T_{\text{TTFT}}(\mathcal{M}) \le T_{\text{TTFT}}^{\text{budget}}, \quad T_{\text{TPOT}}(\mathcal{M}) \le T_{\text{TPOT}}^{\text{budget}}. \tag{29}$$

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.3.5 | Throughput

Given N concurrent requests and total output length L_{out} (tokens) per request, the system throughput Θ is

$$\Theta = \frac{N \cdot L_{\text{out}}}{T_{\text{TTFT}} + L_{\text{out}} \cdot T_{\text{TPOT}}}, \tag{30}$$

and the scheduler periodically selects $\mathcal{M}^*$ to maximize the joint objective:

$$\mathcal{M}^* = \arg\max_{\mathcal{M}} \big(\alpha U_{\text{avg}}(\mathcal{M}) - \beta T_{\text{avg\_latency}}(\mathcal{M}) + \gamma \Theta(\mathcal{M})\big), \tag{31}$$

where α, β, γ are weighting coefficients corresponding to the utilization, latency, and throughput terms in the objective function. subject to migration cost and latency constraints.

4.4 | Adaptive Migration and Load-Aware Scheduling in BanaServe

Based on our performance models and experimental observations, BanaServe employs two complementary runtime algorithms:

1. a **Dynamic Migration Algorithm**that adaptively rebalances computation and memory between prefill and decode instances through module migration.
2. a **Load-aware Request Scheduling Algorithm** that dispatches requests solely based on instance load, enabled by a Global KV Cache Store.

4.4.1 | Dynamic Migration Algorithm

BanaServe uses a Dynamic Migration Algorithm to resolve compute and memory utilization imbalance between prefill and decode instances in PD disaggregation. The algorithm runs in a periodic control cycle that monitors real-time load and performs module migration to balance the workload while limiting migration overhead.

Let $D = \{d_1, d_2, \ldots, d_{|D|}\}$ be the set of devices (GPUs) currently participating in serving, cfg the current allocation of model modules, and $\delta > 0$ the load imbalance threshold. For each device d, we define its normalized utilization as

$$U_d = \frac{C_d}{C_d^{\max}} + \frac{M_d}{M_d^{\max}}, \tag{32}$$

where C_d is the current compute usage, M_d is the current memory usage, and $C_d^{\max}$, $M_d^{\max}$ are the respective hardware capacities. U_d ranges from 0 to 2.0, with larger values indicating heavier combined compute and memory load.

The algorithm operates in four sequential stages that directly correspond to the steps in Algorithm 1:

1. **Load measurement** (lines 3–5): For each device $d \in D$, compute the combined compute and memory utilization $load[d]$ by invoking `MeasureUtilization`. This provides each device's current normalized load, which will be used to determine imbalance.
2. **Load classification** (lines 6–7): Identify overloaded and underloaded devices by constructing:

$$overload = \{d \in D | load[d] - \min(load) > \delta\},$$
$$underload = \{d \in D | \max(load) - load[d] > \delta\}. \tag{33}$$

   Devices in $overload$ exceed the lightestloaded device's utilization by more than the threshold δ, while devices in $underload$ fall below the heaviestloaded device's utilization by more than δ.
3. **Migration decision and execution** (lines 9–17): While there exists $d_o \in overload$ and $d_u \in underload$, calculate the instantaneous load gap:

$$\Delta = load[d_o] - load[d_u]. \tag{34}$$

   If $\Delta \geq \delta$ and d_o supports layer-level migration, invoke `MigrateLayer`(d_o, d_u) to transfer contiguous transformer layers. If $\Delta \geq \delta$ and d_o supports attention-level migration, invoke `MigrateKVHeads`(d_o, d_u) to transfer selected KV Cache segments along attention heads. Before execution, evaluate the migration using:

$$Benefit(m) = \Delta_{\text{before}} - \Delta_{\text{after}},$$
$$Cost(m) = T_{\text{transfer}}(m) + T_{\text{sync}}(m), \tag{35}$$

   where $T_{\text{transfer}}(m)$ is the data transfer latency and $T_{\text{sync}}(m)$ is the synchronization cost with ongoing tasks. The operation proceeds only if $Benefit(m)/Cost(m) \geq \rho$, where ρ is a configurable efficiency ratio.

ALGORITHM 1 | Adaptive module migration algorithm in BanaServe.

1: **Input:** Set of devices D, current module allocation cfg, balance threshold δ
2: **Output:** Updated allocation cfg'
3: // Step 1: Measure current load
4: **for** each d in D **do**
5: $load[d] \leftarrow$ MeasureUtilization(d) ▷ Compute+Memory
6: **end for**
7: $overload \leftarrow \{d \in D \mid load[d] - \min(load) > \delta\}$
8: $underload \leftarrow \{d \in D \mid \max(load) - load[d] > \delta\}$
9: // Step 2: Migration decision loop
10: **while** $\exists\, d_o \in overload$ and $\exists\, d_u \in underload$ **do**
11: $\Delta \leftarrow load[d_o] - load[d_u]$
12: **if** $\Delta \geq \delta \wedge$ SupportsLayerMigration(d_o) **then**
13: MigrateLayer(d_o, d_u)
14: **else if** $\Delta \geq \delta \wedge$ SupportsAttentionMigration(d_o) **then**
15: MigrateKVHeads(d_o, d_u)
16: **end if**
17: Update $load[d_o], load[d_u]$
18: Update cfg' accordingly
19: **end while**
20: **return** cfg'

10970243x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ALGORITHM 2 | Load-aware request scheduling algorithm.

1: **Input:** Request queue Q, Prefill instance set P, Load threshold δ_L
2: **Output:** Dispatch plan D^*
3: // Step 1: Measure current load on each prefill instance
4: **for** each $p_i \in P$ **do**
5: $load[p_i] \leftarrow$ `MeasureUtilization`(p_i) ▷ Compute + Memory
6: $q\_len[p_i] \leftarrow$ `GetQueueLength`(p_i)
7: **end for**
8: // Step 2: Sort instances by load and queue length
9: $candidates \leftarrow$ `Sort`$(P, key = (load, q\_len), order =$ `ascending`$)$
10: // Step 3: Dispatch requests to least-loaded instance
11: **for** each $req \in Q$ **do**
12: $p_{target} \leftarrow$ `Select`$(candidates, policy =$ `least − loaded`$)$
13: **if** $load[p_{target}] < \delta_L$ **then**
14: `AssignRequest`(req, p_{target})
15: $load[p_{target}] \leftarrow load[p_{target}] +$ `EstimateLoad`(req)
16: **else**
17: $p_{target} \leftarrow$ `Select`$(candidates, policy =$ `lowest − queue`$)$
18: `AssignRequest`(req, p_{target})
19: **end if**
20: **end for**
21: // Step 4: Return final dispatch plan
22: $D^* \leftarrow$ `GetDispatchMap`(P)
23: **return** D^*

4. **Update allocation** (lines 18–20): After each migration action, update $load[d_o]$ and $load[d_u]$, revise cfg' to reflect the new module placement, and repeat until no device simultaneously meets overload and underload criteria. Finally, return the updated cfg'.

The complexity per control cycle is

$$O(|D| + N_m), \tag{36}$$

where $|D|$ is the number of devices and N_m is the number of modules on the overloaded device that can be migrated. The $O(|D|)$ term comes from load measurement and classification, and $O(N_m)$ from module selection and cost evaluation. In typical deployments, $|D|$ is in the tens and N_m is bounded by model depth, enabling real-time operation.

To prevent oscillations, hysteresis control is applied using distinct upward and downward thresholds $(\delta_\uparrow, \delta_\downarrow)$. Integration with the Global KV Cache Store ensures that all KV states are preserved during migration, avoiding recomputation and maintaining steady throughput. This dynamic migration mechanism allows BanaServe to adaptively balance workloads using both coarse and fine-grained reallocation strategies under highly dynamic traffic patterns.

4.4.2 | Load-Aware Request Scheduling Algorithm

BanaServe employs a Load-aware Request Scheduling Algorithm to efficiently distribute incoming requests among prefill instances. Unlike cache aware routing policies, this scheduler does not need to consider prefix cache hit rate due to the Global KV Cache Store, which enables KV Cache sharing across all prefill GPUs. The scheduling decision is therefore based solely on real-time load metrics, ensuring balanced resource utilization and minimizing queuing delays.

Let $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ denote the incoming request queue, $P = \{p_1, p_2, \ldots, p_{|P|}\}$ denote the set of active prefill instances, and $\delta_L > 0$ is the load threshold used to trigger alternate routing logic. For each prefill instance p_i, the normalized load is defined as

$$U_{p_i} = \frac{C_{p_i}}{C_{p_i}^{\max}} + \frac{M_{p_i}}{M_{p_i}^{\max}}, \tag{37}$$

where C_{p_i} and M_{p_i} represent the current compute and memory usage, and $C_{p_i}^{\max}$, $M_{p_i}^{\max}$ are the respective capacities.

The algorithm executes the following stages, with each step directly mapped to the corresponding lines in Algorithm 2:

1. **Load measurement** (lines 3–6): For each $p_i \in P$, measure U_{p_i} using `MeasureUtilization` and record its queue length $q\_len[p_i]$ via `GetQueueLength`.
2. **Sorting** (line 8–9): Sort all instances in ascending order by load and then by queue length. This produces the candidate list for dispatch decisions.
3. **Dispatch loop** (lines 11–19): For each request $req \in Q$, select the least-loaded instance p_{target} from the candidate list. If $U_{p_{target}} < \delta_L$, assign req to p_{target}; otherwise, select the instance with the smallest queue length and assign it to that instance. After assignment, increment $load[p_{target}]$ by the estimated load contribution of req.
4. **Finalize plan** (lines 21–23): Generate the dispatch map D^* using `GetDispatchMap` and return it to the orchestration component for execution.

The computational complexity per scheduling cycle is

$$O(|P| \log |P| + |Q|), \tag{38}$$

where $|P|$ is the number of prefill instances and $|Q|$ is the number of requests in the queue. The $O(|P| \log |P|)$ term results from sorting instances by current load and queue length, and the $O(|Q|)$ term covers the per-request assignment loop. In typical deployments, $|P|$ is small (dozens of GPUs), making the algorithm suitable for real-time scheduling.

By removing cache hit rate from the routing criteria through the Global KV Cache Store, this scheduler focuses solely on balancing load and minimizing queuing delay. This design reduces complexity, avoids hotspot formation in prefill nodes, and ensures stable throughput under dynamic workloads.

5 | Evaluations

In this section, we present a comprehensive empirical evaluation of the proposed **BanaServe** framework. Our study compares BanaServe against two state-of-the-art LLM serving systems across multiple model architectures, context length scenarios, and workload intensities. We detail the experimental

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

setup, benchmark selection, evaluation metrics, and load testing methodology, followed by an in-depth analysis of performance results including throughput, end-to-end latency, and scalability.

5.1 | Experimental Setup

To comprehensively evaluate the effectiveness of the BanaServe framework, we conducted an extensive empirical study comparing its performance against two leading state-of-the-art inference systems: DistServe [11] and vLLM [14]. Our evaluation methodology employs a multi-dimensional performance analysis across diverse operational conditions, model architectures, and workload characteristics.

5.1.1 | Model Selection and Configurations

We strategically selected two representative large language models to ensure coverage of different architectural paradigms at the same parameter scale. Table 2 summarizes their key characteristics. The LLaMA-13B[1] model serves as the primary evaluation target within the LLaMA family, enabling us to assess how BanaServe handles large-scale decoder-only transformers with increased model complexity. Additionally, we include OPT-13B[2] to facilitate cross-architecture validation. Although both models have the same parameter count, OPT-13B employs alternative architectural optimizations and training methodologies, allowing us to examine the feasibility of BanaServe across distinct transformer implementations.

TABLE 2 | Model configurations used in experimental evaluation.

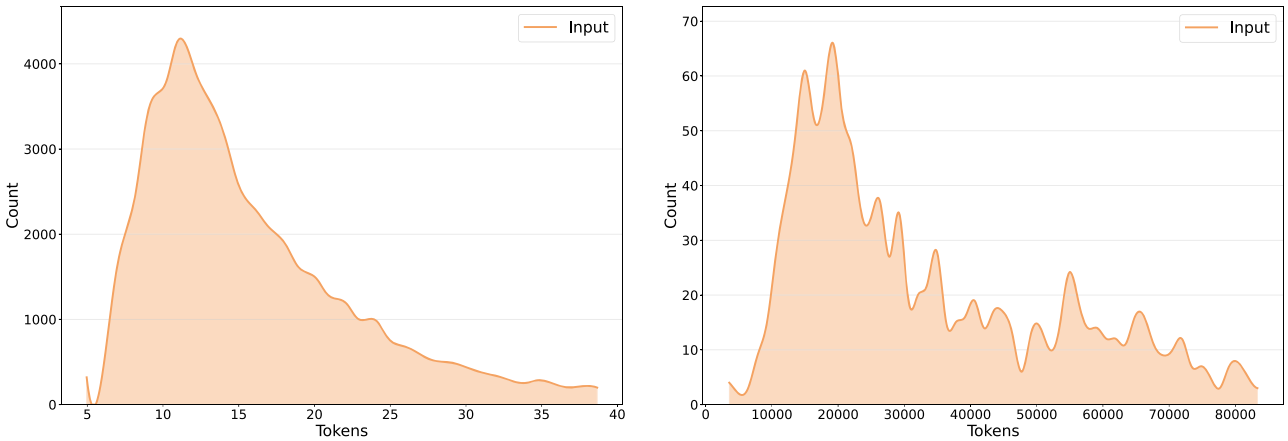
Model	Parameters	Architecture	Evaluation purpose
LLaMA-13B	13B	Decoder-only	Intra-family performance evaluation
OPT-13B	13B	Decoder-only	Cross-architecture validation

5.1.2 | Benchmark Selection and Workload Scenarios

Our experimental design leverages two publicly available, industry-standard benchmarks: **Alpaca**[3] and **LongBench**,[4] which together comprehensively evaluate inference performance under varying context lengths. These benchmarks represent complementary workload patterns commonly encountered in production deployments.

For **short-context** evaluation, we employ the Alpaca benchmark, which consists of 52,000 instruction-following examples generated using self-instruct techniques from Stanford. Alpaca represents typical production inference workloads with input sequences ranging from 4 to 50 tokens as shown in Figure 7a, capturing the characteristics of interactive applications such as chatbots, code assistants, and Q&A systems. The benchmark's diverse instruction types, spanning classification, generation, editing, and reasoning tasks, provide a broad assessment of inference efficiency across different computational patterns. These short-context experiments particularly stress the system's request scheduling mechanisms, time-to-first-token optimization, and serving infrastructure overhead under varied load conditions.

For **long-context** evaluation, we utilize LongBench, a comprehensive multi-task benchmark specifically designed to assess long-context understanding capabilities of large language models. LongBench encompasses 21 diverse tasks across 6 categories, with context lengths from ~2000 to over 85,000 tokens, as shown in Figure 7b. The benchmark includes multi-document QA (HotpotQA, 2WikiMultihopQA), single-document QA (NarrativeQA, Qasper), summarization (GovReport, MultiNews), few-shot learning tasks, synthetic tasks probing specific long-context abilities, and code completion tasks. This variation in sequence length acts as a stress test for KV Cache management and memory optimization strategies. Its multilingual nature, including English, Chinese, and programming languages, ensures our evaluation



(a) (b)

FIGURE 7 | Input length distributions for the two benchmarks used in our evaluation. (a) The Alpaca dataset consists primarily of short-context instruction-following inputs, with sequence lengths ranging from 5 to 40 tokens in our experiments. (b) The LongBench dataset contains long-context inputs with sequence lengths spanning from roughly 2000 to over 85,000 tokens, covering diverse multi-document and multi-task scenarios. In all experiments, the maximum *output length* is capped at 512 tokens to ensure consistency across benchmarks, avoid excessive generation latency, and maintain comparability of throughput and latency metrics while focusing on input-length variability.

10970243x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

captures performance variations across different tokenization schemes and vocabulary distributions.

We adopt a comprehensive metric suite to capture different aspects of system performance. Throughput, measured in tokens per second, quantifies the raw processing capability of each framework and directly reflects the efficiency of GPU utilization and batch processing strategies. This metric is particularly important for understanding system capacity under high-load scenarios. Total time encompasses the end-to-end latency from request submission to complete response generation, capturing all system overheads including model loading, attention computation, KV Cache management, and result post-processing. For user-facing applications, we measure average latency decomposed into TTFT and inter-token latency, providing critical insights into perceived responsiveness. TTFT is especially crucial for interactive applications where users expect immediate feedback, while inter-token latency determines the smoothness of the generation process.

5.1.3 | Load Testing Methodology

To evaluate system behavior under diverse operational conditions, we implemented a progressive load testing strategy with request rates systematically varied from 1 to 20 requests per second (RPS). This range captures system behavior from under-utilized states where latency is primarily determined by computational efficiency, through moderate loads where scheduling and batching strategies become critical, to fully saturated states where queueing delays and resource contention dominate performance. We employed a Poisson arrival process to simulate realistic traffic patterns, as this better represents the bursty nature of real-world inference requests compared to uniform arrivals, allowing us to evaluate system resilience under variable load conditions.

Each experimental configuration underwent rigorous testing with a 60-s warm-up period to ensure system stability, populate all caches, and reach steady-state performance before measurement collection. To ensure statistical validity, all experiments were repeated five times with different random seeds, and we report mean values along with 95% confidence intervals. This methodology allows us to distinguish genuine performance differences from random variations in system behavior and ensures reproducibility of our results.

5.2 | Results and Analysis

In this section, we present a comprehensive evaluation of BanaServe's end-to-end performance against the baseline systems across different workload scenarios and model architectures.

5.2.1 | Baselines

We compare BanaServe with two state-of-the-art LLM serving systems:

vLLM [14]: A widely adopted serving system in both academia and industry that employs continuous batching and PagedAttention for memory-efficient KV Cache management. However, vLLM's monolithic architecture struggles to balance the competing demands of prefill and decode phases, particularly under stringent latency requirements.

DistServe [11]: A disaggregated serving system that separates prefill and decode computations into specialized instances. While this approach mitigates interference between phases, it introduces additional communication overhead and requires careful coordination between disaggregated components.

5.2.2 | Performance

We evaluate **BanaServe** across two representative LLM architectures (LLaMA-13B and OPT-13B) and two context regimes (LongBench for extended sequences, Alpaca for short production-style workloads). Performance is reported across varying input rates (1–20 RPS) and expressed in terms of three primary metrics: throughput (tokens/s), total processing time, and average per-request latency.

LLaMA-13B with Short-context. Figure 8 presents results on the Alpaca benchmark with short, production-style inputs. Across the entire RPS spectrum, BanaServe delivers consistently higher throughput, ranging from 1.1× to 1.2× compared to both DistServe and vLLM, while maintaining lower latency. Reducing per-request latency ensures high responsiveness in interactive settings, and total processing times remain lower under all loads. These improvements stem from BanaServe's minimized scheduling overhead and efficient KV Cache handling, which allow rapid context switches without incurring significant pipeline stalls.

OPT-13B with Short-context. For OPT-13B under the same short-context workload as shown in Figure 9, performance gains are markedly higher. BanaServe achieves throughput improvements ranging from 2.8× to 3.9× compared to DistServe, and up to 3.9× compared to vLLM. Average latency reductions are likewise substantial, with decreases of 3.9% to 78.4% relative to vLLM, and 1.4% to 70.1% relative to DistServe. These results highlight BanaServe's efficiency in high-frequency context-switch scenarios, in which its batching and cache reuse strategies enable near-saturation of GPU utilization while maintaining responsiveness.

LLaMA-13B with Long-context. On the LongBench benchmark with extended sequences as demonstrated in Figure 10, BanaServe delivers throughput improvements of 1.3× to 1.5× over DistServe and up to 1.5× over vLLM. Latency reductions range from 20.6% to 65.3% compared to vLLM and 1.4% to 20.6% compared to DistServe. The performance gap becomes more pronounced under high load conditions (10–20 RPS), where conventional systems suffer from cache contention and pipeline blocking, whereas BanaServe's unified architecture reduces inter-component communication overhead and exploits dynamic batching to balance computational and memory resources across phases.

OPT-13B with Long-context. In the same LongBench setting as illusrated in Figure 11, BanaServe sustains throughput increases of 1.1× to 1.3× relative to DistServe and up to 1.3× relative to

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

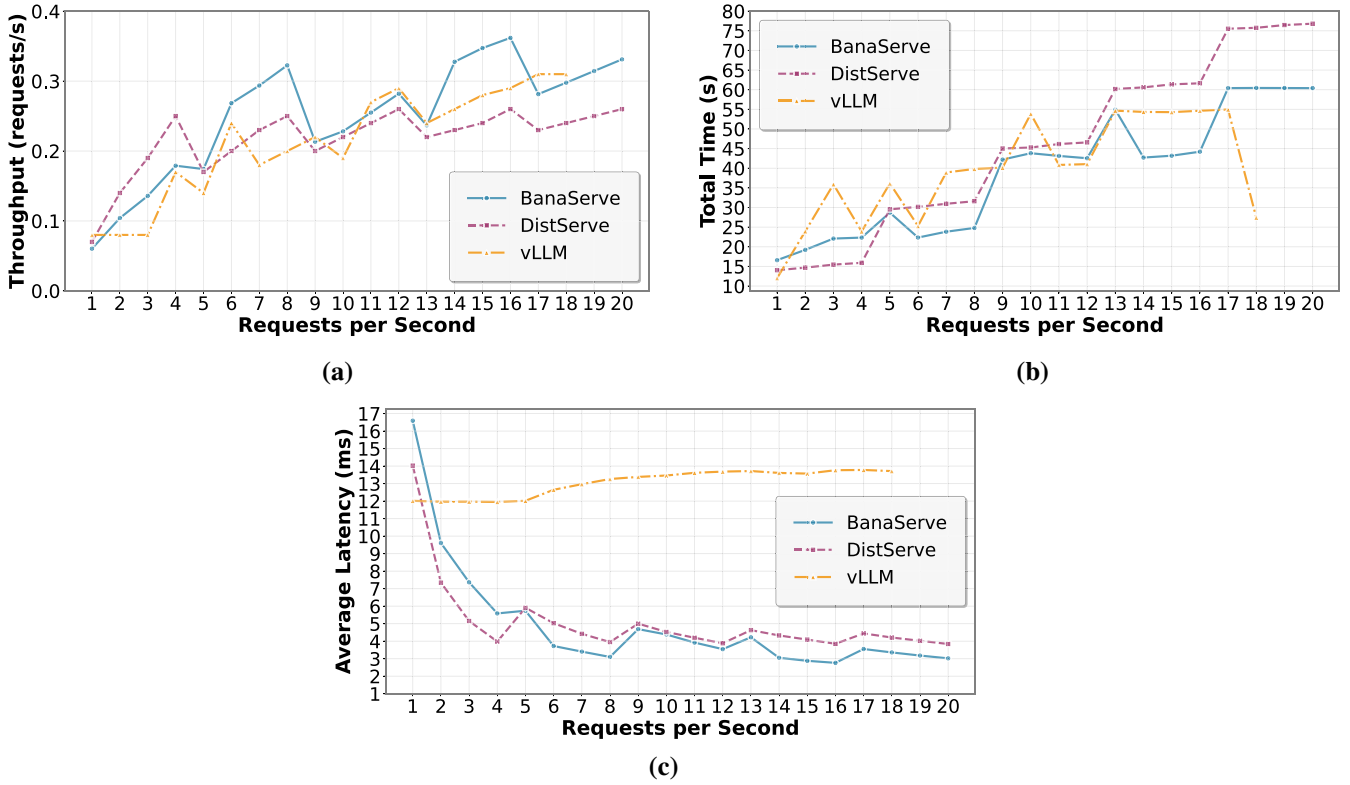


FIGURE 8 | Short-context performance results for **LLaMA-13B**. Experiments compare three inference frameworks: BanaServe, DistServe, and vLLM across different RPS (requests per second) loads. (a) Throughput, (b) total time, (c) average latency.

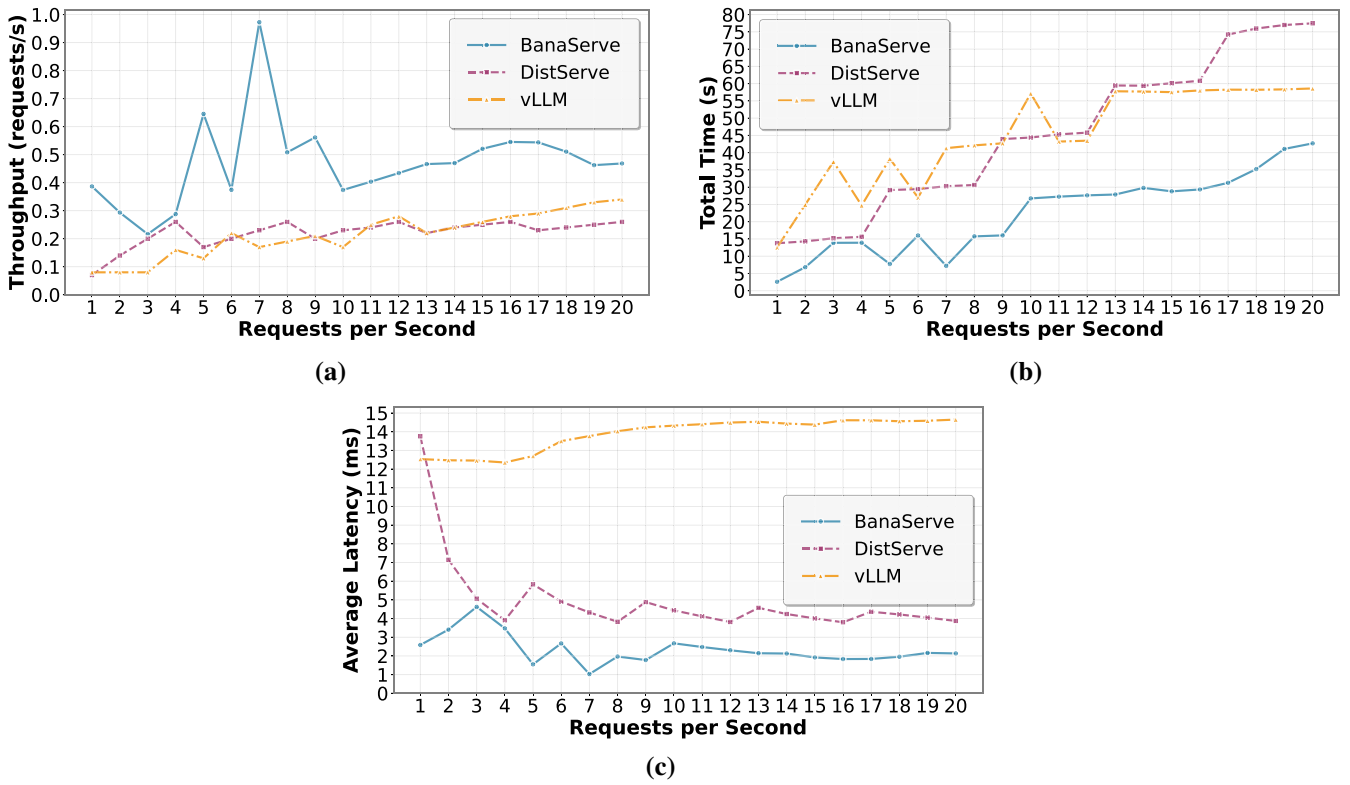


FIGURE 9 | Short-context performance results for **OPT-13B**. Experiments compare three inference frameworks: BanaServe, DistServe, and vLLM across different RPS loads. (a) Throughput, (b) total time, (c) average latency.

10977024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

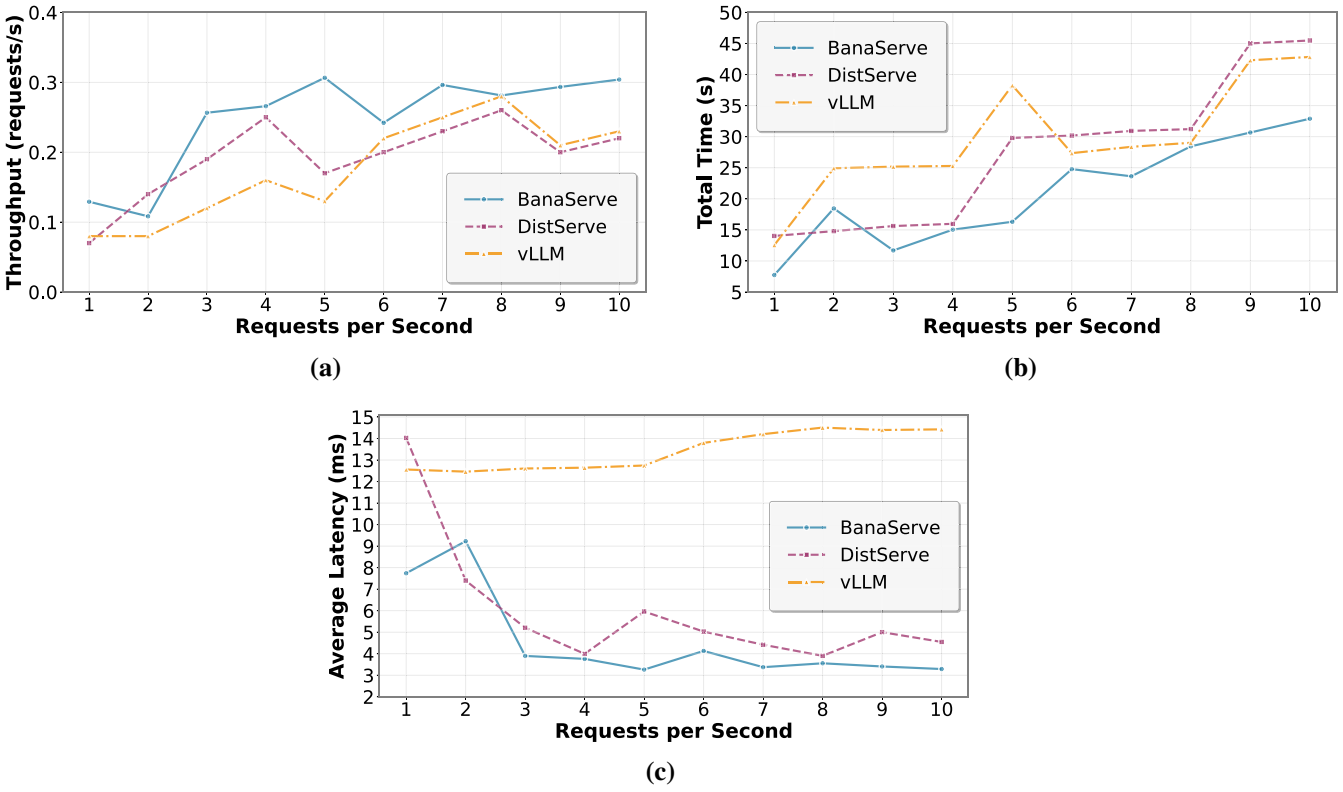


FIGURE 10 | Long-context performance results for **LLaMA-13B**. Experiments compare three inference frameworks: BanaServe, DistServe, and vLLM across different RPS (requests per second) loads. (a) Throughput, (b) total time, (c) average latency.

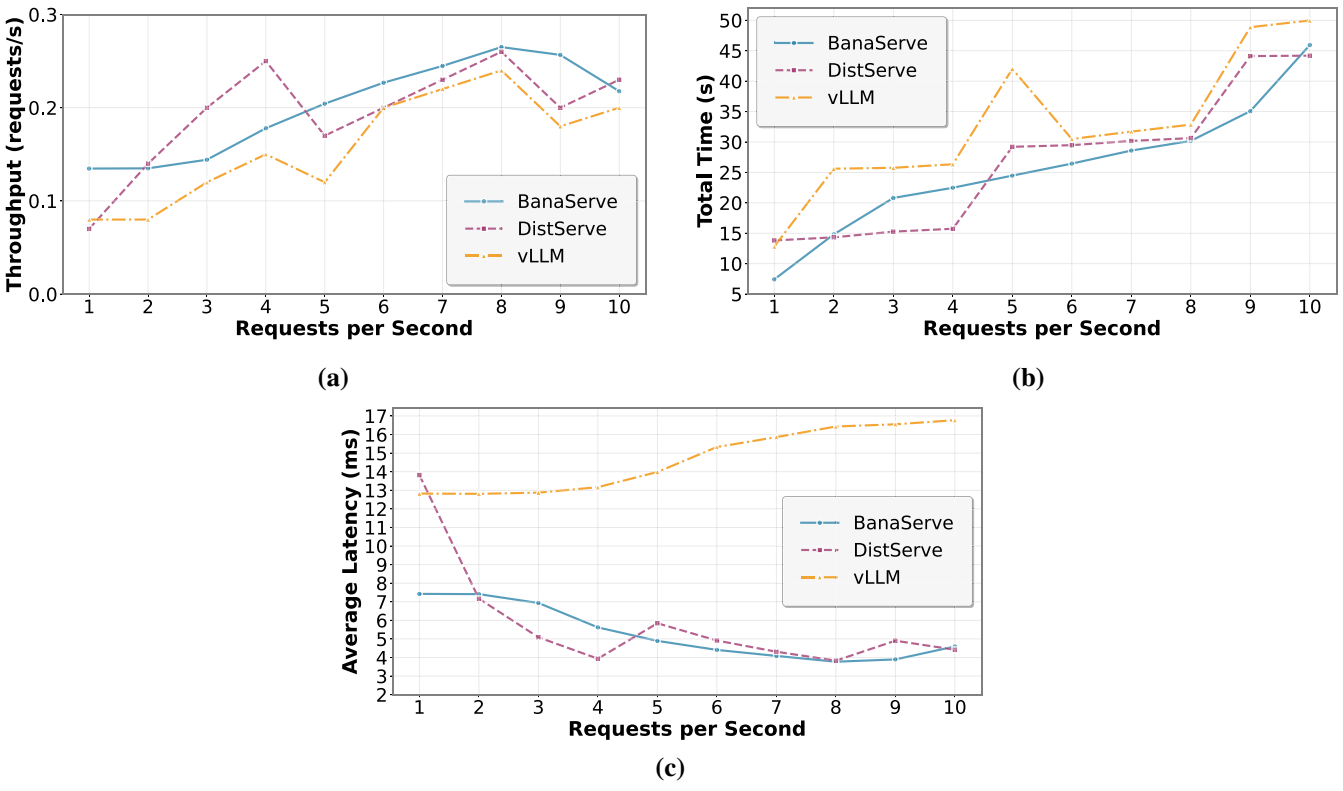


FIGURE 11 | Long-context performance results for **OPT-13B**. Experiments compare three inference frameworks: BanaServe, DistServe, and vLLM across different RPS loads. (a) Throughput, (b) total time, (c) average latency.

10977024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

vLLM. Although these gains are smaller than those observed for LLaMA-13B, they remain consistent across different load levels and can be attributed to reduced KV Cache fetch latency as well as more efficient memory allocation patterns. The latency improvements follow the same pattern, ensuring stable quality of service even when processing large context windows.

Scalability Analysis. As request rates increase from 1 to 20 RPS, BanaServe maintains consistent performance advantages across all configurations. Under light loads (1-5 RPS), the performance gap is primarily attributed to BanaServe's efficient scheduling and reduced system overhead. As load increases (10-20 RPS), BanaServe's superior batching strategies and optimized memory management enable it to sustain higher throughput while other systems experience performance degradation due to resource contention and inefficient cache utilization.

The results validate that BanaServe's unified yet flexible architecture successfully combines the benefits of monolithic systems (low overhead, efficient resource sharing) with those of disaggregated systems (phase-specific optimization, reduced interference), achieving superior performance across diverse workload characteristics and operational conditions.

5.3 | Azure Production Trace Experiments

To complement the synthetic benchmarks, we also evaluate BanaServe using realistic production traces from Azure's publicly released LLM inference datasets [37, 38]. These traces contain anonymized request-level logs from deployed Azure inference services, including timestamps, input/output token lengths, and multimodal request types. The workload exhibits substantial heterogeneity and burstiness, reflecting realistic production conditions.

We analyze the first one-hour segment of the trace, and Figure 12 presents the distributions of input lengths, output lengths, and request arrival rates (RPS). The input and output lengths span a broad range, covering short interactive queries, medium-length coding tasks, and long-form reasoning workloads. The RPS

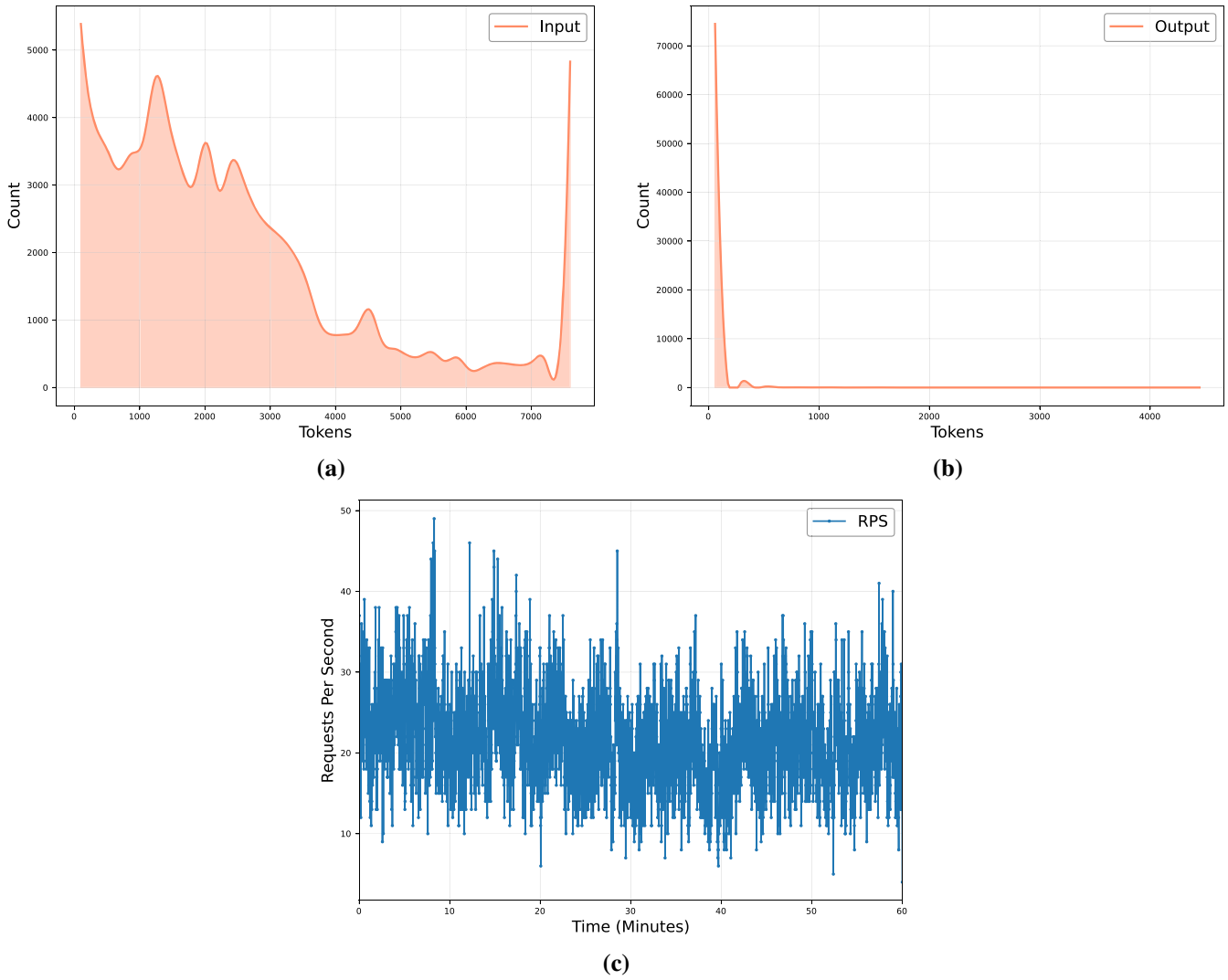


FIGURE 12 | Statistical properties of the first one-hour Azure production trace. The distributions show heavy-tailed input and output lengths and highly bursty request arrival patterns, reflecting realistic and diverse LLM serving workloads in production. (a) Input length, (b) output length, (c) RPS.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

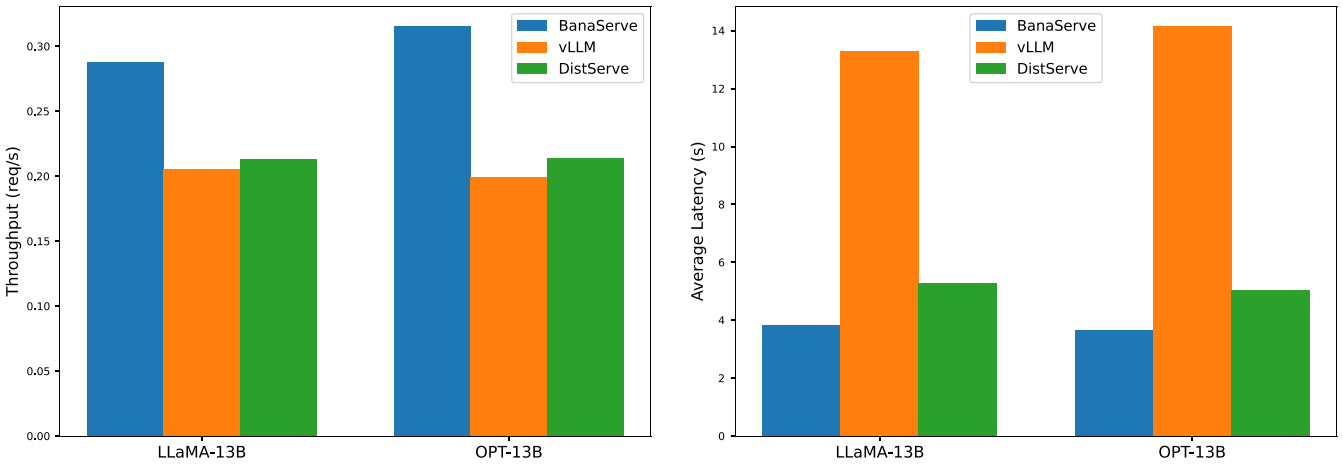


FIGURE 13 | Overall throughput and latency comparison on the Azure production trace for LLaMA-13B and OPT-13B. BanaServe achieves the highest throughput on both models (0.288 req/s on LLaMA-13B and 0.315 req/s on OPT-13B) and provides the lowest latency (3.828 and 3.658 s respectively), outperforming vLLM and DistServe under the same production workload. (a) Throughput, (b) latency.

distribution displays pronounced burstiness, consistent with non-stationary arrival patterns in production environments.

We replay the trace by preserving the original arrival timestamps and using the recorded input and output token lengths. Experiments are conducted on both LLaMA-13B and OPT-13B models. The resulting throughput and latency measurements are summarized in Figure 13.

Across both LLaMA-13B and OPT-13B, BanaServe consistently delivers higher throughput and lower latency than vLLM and DistServe. The improvements are especially pronounced under the heavy-tailed input/output distributions and bursty request arrivals characteristic of the Azure workload. These results show that BanaServe sustains efficient and stable performance when serving real production traces, validating its robustness beyond synthetic benchmarks.

In addition to the absolute values, the Azure trace results show clear relative performance gains. For LLaMA-13B, BanaServe improves throughput by 40.4% over vLLM and 35.2% over DistServe, while reducing average latency by 71.2% compared to vLLM and 27.6% compared to DistServe. For OPT-13B, the gains are even more pronounced: throughput improves by 58.3% over vLLM and 47.2% over DistServe, and latency is reduced by 74.1% relative to vLLM and 27.4% relative to DistServe. These improvements highlight BanaServe's ability to sustain high efficiency even under heavy-tailed input/output distributions and bursty traffic patterns. The consistent gains across both model architectures indicate that the system design generalizes effectively across different transformer implementations when deployed in production-style workloads.

6 | Conclusions, Discussions and Future Work

This paper presents **BanaServe**, a inference serving framework tailored to address compute and memory imbalance and KV Cache management challenges in PD disaggregated architectures for LLMs. We introduce two key components: a Global KV Cache Store for cross-instance KV Cache sharing, and two adaptive runtime algorithms, Dynamic Migration and Load-aware Request Scheduling, that coordinate PD execution without reliance on prefix cache hit rate. Our design leverages layer-wise pipeline overlap to hide communication latencies, granularity aware migration to balance workloads at both coarse (layer-level) and fine (attention-level) levels, and load-oriented scheduling to distribute requests across heterogeneous GPU clusters. Validation using the Alpaca and LongBench benchmarks demonstrates significant reduction in first-token latency, improved throughput under dynamic workloads, and robust performance across short- and long-context inference scenarios. These results confirm the practicality and scalability of BanaServe in production-like serving environments.

Although BanaServe demonstrates strong potential for efficient LLM inference serving in disaggregated architectures, its current implementation still faces several limitations. First, the system's scheduling and migration policies lack comprehensive optimization for heterogeneous hardware, such as mixed-precision accelerators and specialized communication fabrics, which may result in underutilized cluster resources. Second, its workload management is primarily reactive, relying on instantaneous load metrics without predictive modeling from historical patterns, making it less effective under rapid workload fluctuations. Finally, the design is restricted to single-region clusters and has yet to address distributed, multi-region deployment challenges, particularly in enabling WAN-aware KV cache sharing across geo-distributed sites.

As for future work, BanaServe can extend its capabilities along several promising directions. Hardware-aware scheduling and migration should be explored by incorporating device profiles that account for precision modes, interconnect bandwidth, and accelerator heterogeneity, enabling more efficient workload placement. Predictive orchestration, leveraging historical load traces and machine learning techniques such as

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

reinforcement learning, could proactively adjust resource allocation and request routing to better handle workload spikes. Additionally, supporting multi-region scaling with WAN-optimized KV cache synchronization would allow BanaServe to achieve low-latency inference across geo-distributed deployments. Finally, integrating with container orchestration platforms like Kubernetes can facilitate realistic deployment, enabling the evaluation of scaling, scheduling, and placement policies in production environments.

Author Contributions

All authors participated in extensive discussions and contributed to the conceptualization, analysis, and finalization of this manuscript. The specific contributions are as follows: Yiyuan He conceived the original research idea and implemented the experimental framework, conducted comprehensive experiments, and authored the primary content of the manuscript. Minxian Xu provided overarching guidance for the experimental methodology and manuscript development. Jingfeng Wu contributed to manuscript enhancement through detailed revisions, language refinement, and improvements in overall presentation quality. Jianmin Hu primarily assisted in the collection and verification of experimental data, ensuring the accuracy and reliability of results. Kejiang Ye and Chengzhong Xu provided critical insights into application scenarios and scalability considerations, and also contributed to the refinement of the manuscript. Authors from Alibaba Group AIOS Team (Chong Ma, Min Shen, Le Chen, and Lin Qu) have provided Alibaba's large-scale experimental platform for performance validation and technical supports on developing BanaServe system.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grant 62572462, Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010251, 2023B1515130002), Key Research and Development and Technology Transfer Program of Inner Mongolia Autonomous Region (2025YFHH0110), and Shenzhen Science and Technology Program under Grant JCYJ20240813155810014. We also thank Alibaba Group AIOS Team for providing large-scale platform and technical support.

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

Endnotes

[1]https://huggingface.co/meta-llama/Llama-2-13b.

[2]https://huggingface.co/facebook/opt-13b.

[3]https://github.com/tatsu-lab/stanford_alpaca.

[4]https://github.com/THUDM/LongBench.

References

1. J. Achiam, S. Adler, S. Agarwal, et al., "Gpt-4 Technical Report," *arXiv Preprint, arXiv:2303.08774* (2023), https://arxiv.org/abs/2303.08774.

2. H. Touvron, T. Lavril, G. Izacard, et al., "Llama: Open and Efficient Foundation Language Models," *arXiv Preprint, arXiv:2302.13971* (2023), https://arxiv.org/abs/2302.13971.

3. Anthropic Claude, "Large Language Model," 2025, https://claude.ai.

4. M. Monteiro, B. C. Branco, S. Silvestre, G. Avelino, and M. T. Valente, "NoCodeGPT: A No-Code Interface for Building Web Apps With Language Models," *Software: Practice and Experience* 55 (2025): 1408–1424.

5. A. Nguyen-Duc, B. Cabrero-Daniel, A. Przybylek, et al., "Generative Artificial Intelligence for Software Engineering—A Research Agenda," *Software: Practice and Experience* 55 (2025): 1806–1843.

6. Z. Zeng, T. Zhang, Z. Lu, et al., "Subkv: Quantizing Long Context KV Cache for Sub-Billion Parameter Language Models on Edge Devices," *Software: Practice and Experience* 55 (2025): 1287–1304.

7. A. Vaswani, N. Shazeer, N. Parmar, et al., "Attention is All You Need," *Advances in Neural Information Processing Systems* (2017): 6000–6010, https://dl.acm.org/doi/10.5555/3295222.3295349.

8. Z. Zhou, X. Ning, K. Hong, et al., "A Survey on Efficient Inference for Large Language Models," arXiv preprint, arXiv:2404.14294 (2024): 1–36.

9. Z. Zhang, Y. Sheng, T. Zhou, et al., "H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models," *Advances in Neural Information Processing Systems* 36 (2023): 34661–34710.

10. S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao, "Model Tells You What to Discard: Adaptive KV Cache Compression for LLMS," *The Twelfth International Conference on Learning Representations* (2023), 1–14, https://openreview.net/forum?id=uNrFpDPMyo.

11. Y. Zhong, S. Liu, J. Chen, et al., "DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (USENIX, 2024), 193–210.

12. P. Patel, E. Choukse, C. Zhang, et al., "Splitwise: Efficient Generative LLM Inference Using Phase Splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)* (ACM, 2024), 118–132.

13. L. Zheng, L. Yin, Z. Xie, et al., *Sglang: Efficient Execution of Structured Language Model Programs* (NIPS, 2024), 62557–62583.

14. W. Kwon, Z. Li, S. Zhuang, et al., "Efficient Memory Management for Large Language Model Serving With Pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles* (USENIX, 2023), 611–626.

15. T. Wolf, L. Debut, V. Sanh, et al., "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Association for Computational Linguistics, 2020), 38–45.

16. R. Taori, I. Gulrajani, T. Zhang, et al., "Stanford Alpaca: An Instruction-Following LLaMA Model," 2023, https://github.com/tatsu-lab/stanford_alpaca.

17. NVIDIA Corporation, "NVIDIA Dynamo: Distributed Orchestration for Large-Scale LLM Serving," 2025, https://developer.nvidia.com/nvidia-dynamo.

18. Y. Bai, X. Lv, J. Zhang, et al., "{L}ong{B}ench: A Bilingual, Multitask Benchmark for Long Context Understanding," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, vol. 1 (Association for Computational Linguistics, 2024), 3119–3137, https://aclanthology.org/2024.acl-long.172.

19. T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and Memory-Efficient Exact Attention With Io-Awareness," *Advances in Neural Information Processing Systems* 35 (2022): 16344–16359.

20. J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and Accurate Attention With Asynchrony and Low-Precision," *Advances in Neural Information Processing Systems* 37 (2024): 68658–68685.

21. S. Mukherjee, A. Mitra, G. Jawahar, S. Agarwal, H. Palangi, and A. Awadallah, "Orca: Progressive Learning From Complex Explanation Traces of Gpt-4," arXiv Preprint, arXiv:2306.02707 (2023): 1–51.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

22. A. Agrawal, N. Kedia, A. Panwar, et al., "Taming Throughput-Latency Tradeoff in LLM Inference With Sarathi-Serve," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (USENIX, 2024), 117–134.

23. D. Zhang, H. Wang, Y. Liu, et al., "BlitzScale: Fast and Live Large Model Autoscaling With O(1) Host Caching," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)* (USENIX, 2025), 275–293.

24. C. Holmes, M. Tanaka, M. Wyatt, et al., "Deepspeed-Fastgen: High-Throughput Text Generation for LLMS via MII and Deepspeed-Inference," arXiv Preprint, arXiv:2401.08671 (2024): 1–14.

25. NVIDIA Corporation, "TensorRT-LLM: High-Performance LLM Inference Library From NVIDIA," https://developer.nvidia.com/tensorrt-llm.

26. B. Wu, S. Liu, Y. Zhong, P. Sun, X. Liu, and X. Jin, "Loongserve: Efficiently Serving Long-Context Large Language Models With Elastic Sequence Parallelism," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (ACM, 2024), 640–654.

27. K. Hong, G. Dai, J. Xu, et al., "Flashdecoding++: Faster Large Language Model Inference With Asynchronization, Flat Gemm Optimization, and Heuristics," *Proceedings of Machine Learning and Systems* 6 (2024): 148–161.

28. C. Hu, H. Huang, L. Xu, et al., ShuffleInfer: Disaggregate LLM Inference for Mixed Downstream Workloads, *ACM Transactions on Architecture and Code Optimization*. 2025, Association for Computing Machinery, 1544–3566, https://doi.org/10.1145/3732941.

29. R. Qin, Z. Li, W. He, et al., "Mooncake: Trading More Storage for Less Computation — A KVCache-Centric Architecture for Serving LLM Chatbot," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)* (USENIX Association, 2025), 155–170, https://www.usenix.org/conference/fast25/presentation/qin.

30. C. Hu, H. Huang, J. Hu, et al., "Memserve: Context Caching for Disaggregated Llm Serving With Elastic Memory Pool," arXiv preprint, arXiv:2406.17565 (2024): 1–14.

31. S. Chen, R. Jiang, D. Yu, et al., "KVDirect: Distributed Disaggregated LLM Inference," arXiv preprint, arXiv:2501.14743 (2024).

32. J. Feng, Y. Huang, R. Zhang, S. Liang, M. Yan, and J. Wu, "WindServe: Efficient Phase-Disaggregated LLM Serving With Stream-Based Dynamic Scheduling," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture* (ACM, Association for Computing Machinery, 2025), 1283–1295.

33. K. Hong, L. Chen, Z. Wang, et al., "Semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage," arXiv preprint, arXiv:2504.19867 (2025).

34. B. Sun, Z. Huang, H. Zhao, et al., "Llumnix: Dynamic Scheduling for Large Language Model Serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (USENIX. USENIX Association, 2024), 173–191.

35. X. Miao, C. Shi, J. Duan, et al., "SpotServe: Serving Generative Large Language Models on Preemptible Instances," in *ASPLOS '24* (Association for Computing Machinery, 2024), 1112–1127.

36. Y. He, M. Xu, J. Wu, W. Zheng, K. Ye, and C. Xu, "UELLM: A Unified and Efficient Approach for Large Language Model Inference Serving," in *Service-Oriented Computing: 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3–6, 2024, Proceedings, Part I* (Springer-Verlag, 2024), 218–235.

37. M. Azure, "Azure LLM Inference Dataset 2024," 2024, https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2024.md.

38. M. Azure, "Azure Multimodal Model Inference Dataset 2025," 2025, https://github.com/Azure/AzurePublicDataset/blob/master/AzureLMMInferenceDataset2025.md.

1097024x, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.70054 by Minxian Xu - Cas/Shenzhen Institute Of Advanced Technology/Library , Wiley Online Library on [21/01/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License