DynaServe: Unified and Elastic Tandem-Style Execution for Dynamic Disaggregated LLM Serving

Chaoyi Ruan*
NUS
ruancy@comp.nus.edu.sg

Yinhe Chen*
USTC
chenyh18@mail.ustc.edu.cn

Dongqi Tian
USTC
dongqitian@mail.ustc.edu.cn

Yandong Shi
USTC
yandongshi@mail.ustc.edu.cn

Yongji Wu
UCB
yongji.wu@berkeley.edu

Jialin Li
NUS
lijl@comp.nus.edu.sg

Cheng Li
USTC
chengli7@ustc.edu.cn

Abstract

Modern large language model (LLM) serving must efficiently handle highly dynamic workloads, where prompt and response lengths vary significantly across requests. Existing systems typically adopt either colocated execution—where prefill and decode stages share the same GPU for high throughput—or disaggregated execution, which decouples the two stages and assign their tasks to dedicated GPUs for interference avoidance. However, both paradigms face critical limitations: colocation suffers from resource contention and prolonged tail latency, whereas disaggregation likely leads to resource wasting when prefill or decode GPUs are not fully occupied.

To address the above limitations, we introduce DynaServe, a unified LLM serving framework based on the *Tandem* Serving model. Under this model, DynaServe elastically decomposes each request into two virtual sub-requests that are collaboratively processed by a pair of GPU instances. The Lead GPU handles the initial prompt and early generation, while the Follow GPU completes decoding—enabling dynamic load balancing, fine-grained batching, and coherent execution across distributed resources. By coordinating computation and memory across the cluster, DynaServe adapts to diverse and bursty workloads while maintaining stringent latency service-level objectives (SLOs). Evaluations on real-world traces show that DynaServe improves end-to-end Serving Capacity by up to 1.23 ×, increases the overall goodput from 1.15 × to 4.34 ×, and improve the memory utilization by up to 49% compared to state-of-the-art colocated and disaggregated systems.

1 Introduction

Large language models (LLMs) have emerged as foundational components in modern applications ranging from code generation and customer support to scientific computing. These models perform inference through autoregressive decoding and long-input sequence processing, making the process both compute-intensive and memory-bound. To optimize this workflow, LLM serving systems typically decompose inference into two phases: the prefill phase, which computes and caches key-value (KV) pairs for input tokens, and the decode phase, which generates tokens sequentially.

State-of-the-art LLM serving systems choose between two paradigms: colocated execution, where prefill and decode phases share the same GPU, or prefill-decode (PD) disaggregation, which assigns them to separate devices. While colocation improves resource utilization and inference throughput, it suffers from contention between prefill and decode. Contention results in elevated tail latency, particularly for long-context prompts, leading to more frequent service-level objective (SLO) violations. Disaggregation, on the other hand, eliminates interference and thus improves end-to-end inference latency. However, the approach often underutilizes hardware resources; its rigid nature also reduces scheduling flexibility under diverse, bursty workloads. Figure 1 illustrates the trade-offs between the two paradigms.

Is this latency-throughput trade-off fundamental in LLM serving? In this work, we answer this question in the negative by proposing DynaServe, a unified and dynamic LLM serving framework based on a novel Tandem Serving execution model [1]. The model splits each inference request into two virtual subrequests that are processed collaboratively across two sets of GPU instances. Critically, unlike prior systems that physically separate the prefill and decode phases, we explicitly permit the two subrequests to execute on either GPU instance. Consequently, DynaServe supports elastic and hybrid splitting, allowing partial prefill and early decoding to be handled together when beneficial. The design enables fine-grained control over the ratio of prompt length to generation length, maximizing resource utilization while meeting latency SLOs.

*Chaoyi Ruan and Yinhe Chen equally contributed to this work.

[1]The model name is inspired by the coordination of a tandem bicycle.

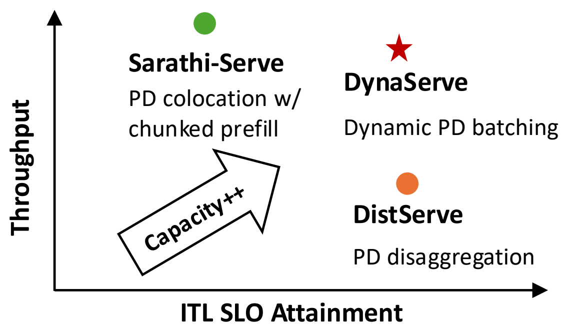


Figure 1. Throughput vs. SLO attainment across serving architectures. **Sarathi-serve** maximizes GPU usage but often misses latency targets. **Distserve** meets latency goals but underutilizes hardware. **DynaServe** achieves higher capacity—defined as the maximum throughput while keeping latency violations under control—by balancing both, moving toward the top-right.

For each inference request, the DynaServe runtime scheduler selects the Leader GPU, which processes the initial portion of the request, and the Follower GPU, which handles the remaining prompt and decoding. The scheduler dynamically composes optimal batches and chunk sizes based on current load and memory availability, adapting to real-time workload changes. This approach unifies colocated, disaggregated, and hybrid modes within a single execution-aware framework.

In summary, this paper makes the following contributions:

- The Tandem Serving abstraction: We introduce a novel execution model in which each LLM request is split into cooperating α/β sub-requests that can be executed across multiple GPUs in a fluid and dynamic manner.
- A dynamic scheduling framework: We design a system-aware global scheduler that optimizes subrequest placement, batch composition, and split ratios to meet latency SLOs while maximizing GPU utilization.
- End-to-end implementation and evaluation: We implement DynaServe and evaluate it in production-scale GPU clusters, demonstrating up to 1.23 × and 1.17 × serving capacity improvement over state-of-the-art PD colocation (chunked prefill) and PD disaggregation systems, respectively.

2 Background

2.1 LLM Inference

A large language model (LLM) is composed of multiple transformer layers that sequentially process continuous token embeddings to predict the next token. The inference process is autoregressive—meaning the model generates one token at a time, with each output conditioned on all previously generated tokens. This generation continues until a stopping criterion is met, such as encountering an end-of-sequence (EOS) token or reaching a predefined length limit. However, as the context length increases, the autoregressive approach introduces growing computational redundancy. This makes the attention mechanism a major performance bottleneck during LLM inference [4, 18].

To mitigate this inefficiency, the KV Cache technique has been introduced, enabling the model to reuse previously computed key and value vectors [7]. This optimization divides inference into two distinct stages: prefill and decode. In the prefill phase, the input prompt is processed in parallel, and key/value (*K/V*) vectors are computed for each token at every transformer-attention layer, forming the initial KV Cache. Then, during the decode phase, token generation occurs step-by-step. At each step, the model only computes the query/key/value for the current token, retrieves the stored *K/V* vectors from the cache, and performs attention to predict the next token. The newly generated token's *K/V* vectors are then added to the cache, allowing for efficient incremental decoding.

2.2 PD Colocation

LLM serving has traditionally followed a colocated architecture, where the compute-intensive prefill and memory-bound decode phases are executed within the same GPU instance. This method often suffers from performance bottlenecks—particularly long-tail latency—when long input sequences create contention between the two phases. To address this, Sarathi-Serve[1] introduces chunked prefill, a technique that splits prefill tokens into smaller chunks with a fixed maximum size, and schedules them incrementally alongside ongoing decode tasks. This continuous hybrid batching approach prevents individual requests from monopolizing compute resources and helps reduce tail latency. Building on this idea, POD Attention[6] further improves efficiency by fusing prefill and decode into a single GPU kernel, enhancing overlap and resource utilization.

However, these techniques remain inherently tied to colocated execution and cannot fully eliminate interference. Decode operations can still experience latency spikes due to bursty or overlapping prefill workloads. Reducing the maximum number of tokens in a batch (i.e., chunk size) can help mitigate tail latency by minimizing contention, but it increases KV-cache memory access due to repeated reads from prior chunks, leading to sub-optimal performance. Conversely, increasing chunk size improves throughput but exacerbates interference, resulting in longer tail latency. As a result, tunig chunk size presents a fundamental throughput–latency trade-off and is hard to fit all workloads.

2.3 PD Disaggregation

To further improve latency under strict service level objective (SLO) constraints, prefill-decode (PD) disaggregated serving

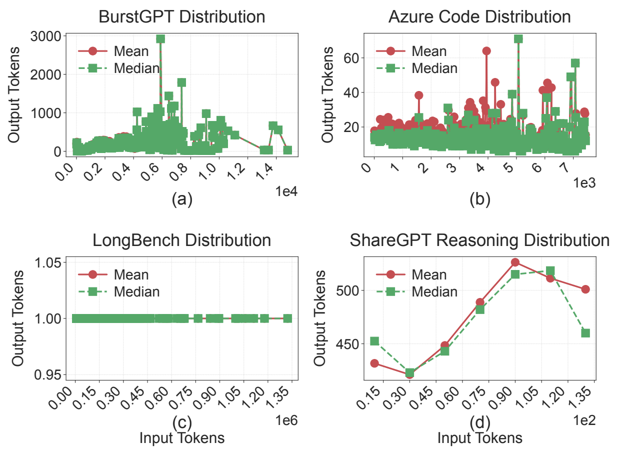


Figure 2. The distrbution of request and response length across four datasets

has emerged as a promising alternative to colocated execution [15–17, 21]. This architecture physically separates the prefill and decode stages into distinct execution units, effectively eliminating resource interference between them. By isolating the compute-intensive prefill stage from the more latency-sensitive decode stage, PD disaggregation helps ensure more stable and predictable response times. A number of recent studies have explored this design to improve efficiency and scalability. DistServe [21] splits prefill and decode workloads across different GPUs to reduce contention and independently tune each stage. Splitwise [15] goes further by jointly optimizing cost, throughput, and power efficiency in disaggregated setups. Mooncake [16] enhances system scalability with a disaggregated key-value (KV) cache architecture that utilizes CPU, DRAM, and SSD resources. More recently, Adrenaline [8] partially improves prefill GPU utilization by offloading attention computation to alternative resources.

Despite these advances, the rigid separation (prompt-based) in PD disaggregation can lead to inefficiencies in certain scenarios, leaving either the decode-side or prefill-side instances underutilized. As a result, balancing high resource utilization with low tail latency remains a central challenge in LLM serving system design.

3 Motivations and Challenges

Apart from the inherent limitation of PD disaggregation and PD colocation, in reality, workloads are various and contain multiple types, long context or long output, exposing significant challenges to LLM serving. In this section, we will dive into the dynamic nature of LLM workloads and its relevant challenges and potention solutions.

3.1 Dynamic Workloads Characteristics

Real-world LLM requests exhibit highly dynamic in terms of both input and output lengths. As shown in Figure 2, token distributions vary widely across representative datasets like BurstGPT, Azure Code, LongBench, and ShareGPT Reasoning. BurstGPT includes long inputs and highly variable outputs; Azure Code shows short but unpredictable patterns; LongBench is memory-intensive with extremely long prompts and short outputs; and ShareGPT Reasoning features stable, high-output lengths from moderate inputs. This variability is further echoed by Infinite-LLM [9], which observes context lengths ranging from a few tokens to over 2 million in production settings.

Such diversity in workloads poses significant system challenges, in terms of handling extreme fluctuations in compute loads and KV cache memory usage across different requests. Next, we will discuss how such variance in workloads impact the effectiveness of PD disaggregation and colocation solutions.

3.2 Disaggregation vs. Colocation: Trade-offs in LLM Serving

While prefill-decode (PD) disaggregation and colocation are the dominant paradigms in LLM serving, our analysis shows that neither offers a universally optimal solution. As illustrated in Figure 3, each approach presents trade-offs in GPU utilization, batching efficiency, and tail latency under varying workloads and latency service-level objectives (SLOs).

Colocation executes both prefill and decode phases on the same GPU. As shown in Figure 3-a, this typically leads to higher model FLOPs utilization (MFU) and more balanced GPU resource usage, since each GPU handles both compute- and memory-bound tasks. However, colocation also introduces significant intra-GPU contention, particularly with long-context prompts, leading to elevated tail latency and difficulty in meeting tight SLOs, as observed in Figure 3-c.

In contrast, PD disaggregation assigns prefill and decode tasks to separate GPUs, improving isolation and making it easier to meet latency constraints. However, this comes at the cost of imbalance across instances. As shown in Figure 3-a and Figure 3-b, prefill GPUs often have high MFU but underutilize their memory (e.g., KV cache), while decode GPUs face memory saturation and lower MFU, resulting in reduced batching efficiency and lower overall throughput.

Therefore, static colocation favors throughput and resource efficiency but struggles with latency guarantees, while disaggregation improves latency compliance but sacrifices utilization and scheduling flexibility. These limitations motivate the need for a dynamic, hybrid approach that adapts execution to workload patterns and system state.

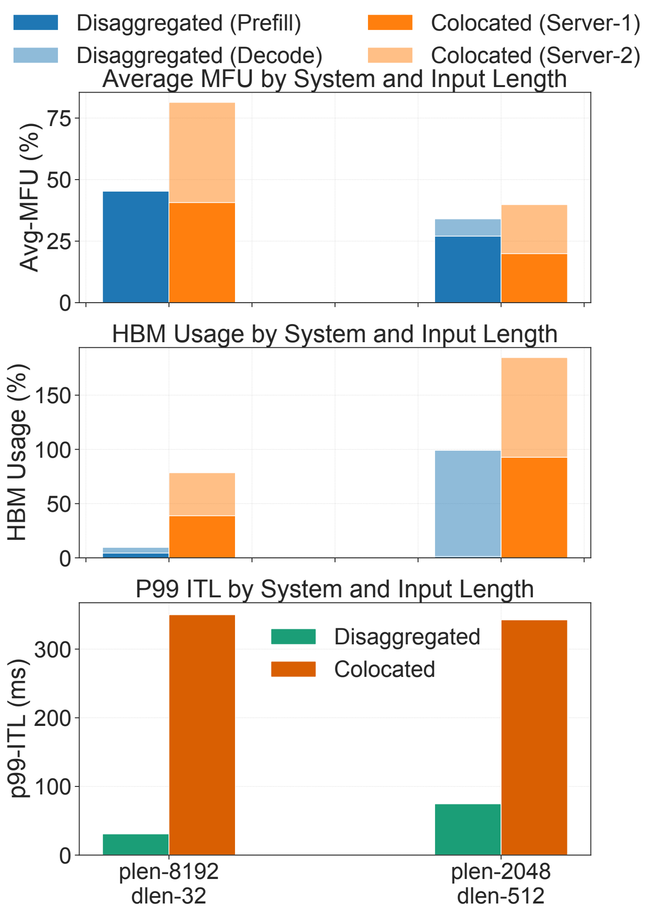


Figure 3. GPU compute, KV cache utilization and inter-token latency when serving the Qwen-2.5-14B [3] model across 2 A100 GPUs, under both PD disaggregation and colocation setups. We tune the request rates to saturate the GPUs. For disaggregation, we show the GPU utilization for both prefill and decode instances, as well as their average. Since the utilization is near identical on the 2 GPUs for colocation, we only present the average.

3.3 Elastic, unified, and virtual request execution.

To address the rigidity of traditional static colocation and prompt-bound disaggregation strategies, we propose a Tandem Serving paradigm – an execution abstraction where two GPU instances work collaboratively, akin to the front and rear wheels of a bicycle. In this metaphor, the Lead GPU (the front wheel) initiates the request by processing the initial prompt and, if beneficial, generating early tokens. This is encapsulated in the virtual request α. The Follow GPU (the rear wheel) subsequently assumes responsibility for completing the remaining prompt and executing the bulk of the decoding, represented by β. This tandem execution enables seamless handoff between phases, maintaining continuous GPU utilization and achieving low-latency responses—even under bursty and dynamic workloads.

The principal insight behind Tandem Serving is an elastic and unified abstraction that decouples request execution from rigid prefill/decode boundaries. Rather than statically assigning an entire prompt or response phase to a single GPU, Tandem Serving allows logical decomposition and scheduling of requests at fine granularity. This dynamic orchestration adapts to real-time cluster conditions – such as compute availability, memory pressure, or bandwidth constraints—thereby increasing throughput and responsiveness. As depicted in Figure 5, this approach fluidly manages computation and memory across distributed resources, significantly reducing resource fragmentation and unlocking higher serving efficiency for modern LLM workloads.

Challenge: However, realizing the full benefits of virtualized and elastic execution brings several technical challenges.

Unified Execution Management. Dynamic request disaggregation introduces a wide range of execution granularities—from fully split prefill and decode stages to partially chunked execution—which may coexist within the same system or even a single request. This flexibility breaks the assumptions of prior serving systems that rely on fixed execution boundaries. The system must now manage non-deterministic phase transitions, coordinate transient states like KV Cache and token positions, and enable seamless handoffs between execution stages without a predefined order. Supporting this requires a runtime capable of fine-grained tracking, adaptive dispatching, and low-latency communication across heterogeneous GPU instances.

Dynamic Scheduling Complexity. Another significant challenge is the scheduling complexity introduced by dynamic splitting. Unlike static policies such as chunked prefill, Tandem Serving must make per-request splitting decisions at runtime based on current system load and workload characteristics. Each decision affects compute distribution, memory usage, and communication overhead, expanding the optimization space dramatically. The scheduler must reason over these dimensions in real time—balancing batching efficiency, instance placement, and resource contention—while still adhering to strict inter-token latency objectives.

4 Analysis

Modern LLM serving systems operate under tight latency budgets while trying to maximize GPU utilization. This section provides a structured analysis of how batching strategies and dynamic execution affect throughput and efficiency. We begin by identifying the key factors influencing system performance, then analyze hybrid batching trade-offs using microbenchmarks. Based on these insights, we present a formal model that captures the optimization landscape of dynamic request execution.

4.1 Key Parameters of Modeling

Table 1 provides a formal notation for modeling the problem of optimizing large language model (LLM) serving throughput by defining key parameters and constraints.

Each incoming request r from the global request set ($\mathcal{R}$) is logically split into two virtual requests to enable elastic execution: The first, denoted by r^{α}, primarily handles prefill phase but may also include a small portion of token generation. The second virtual request r^{β}, takes over the remaining part of the processing, which typically consists of most of token generation steps. These virtual requests are assigned to instances as part of sets $\mathcal{R}^{\alpha}$ and $\mathcal{R}^{\beta}$. The exact partition of each original request into its r^{α} and r^{β} components is determined by the global scheduler (GS), based on the current system load and resource availability.

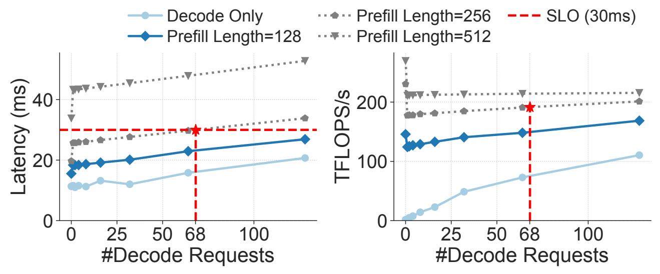
Requests are processed in batches ($\mathcal{B}_i$) on each instance using a First-Come-First-Serve ($FCFS$) policy, with execution times $T(b)$ and memory consumption $M(b)$ constrained by system-level limits such as high-bandwidth memory (HBM) and inter-token latency service level objective (SLO, r_{ITL}). The α and β stages are further optimized by tuning parameters such as the partitioning factor (ϕ) and chunk size (C), ensuring efficient transitions and batch execution. Compute efficiency of one batch is measured in terms of FLOPS ($FLOPS(b)$), with the objective of maximizing throughput while adhering to system constraints. This structured approach enables precise modeling and optimization of LLM serving pipelines.

Symbol	Definition
$\mathcal{R}$	Global set of requests.
$\mathcal{R}_i$	Requests processed by instance i.
$\mathcal{R}_i^{\alpha,\beta}$	Specific virtual requests assigned to instance i.
$\mathcal{I}_r^{\alpha,\beta}$	Instance indices assigned to virtual request α or β.
$\mathcal{B}_i$	Scheduled Batches on instance i.
GS	Global scheduler partitioning $\mathcal{R}$ into $\{\mathcal{R}_i\}$.
ϕ	PD factor.
$FCFS$	First-Come-First-Serve scheduler in local instance.
C_i	Chunk size for $instance_i$.
r_{ITL}	Inter-token latency SLO for r.
$T(b)$	Execution time of batch b.
$M(b)$	Memory consumption of b.
HBM	High-bandwidth memory limit.
$FLOPS(b)$	FLOPS when computing batch b.

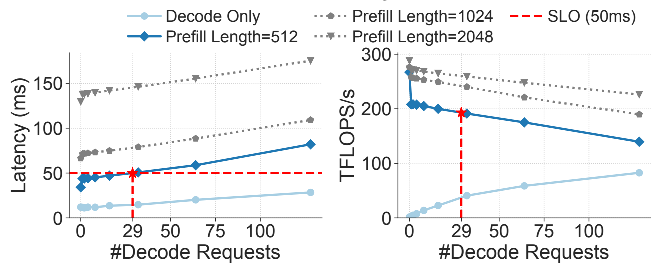
Table 1. Key notation

4.2 Latency and Throughput Trade-offs in Hybrid Batching

Then, we study how the composition of a single inference batch impacts latency and throughput. In Figure 4, we present the relationship between the number of decode requests and two metrics: batch latency (left) and GPU compute utilization in terms of TFLOPs/s (right), under different prefill chunk sizes mixed with the decode. We use Llama-3.1-8B model and a A100 GPU. Two scenarios are evaluated: one with shorter context lengths (128 tokens; top row), and one with longer context lengths (1024 tokens; bottom row).



(a) Decode Length=128



(b) Decode Length=1024

Figure 4. Latency and GPU compute utilization under different batching strategies for Llama-3.1-8B on a A100 GPU. Dashed red line represents latency SLO and the number of decode requests being processed. With a fixed decode workload, different sizes of prefill chunks can be mixed in a batch.

On the left-hand side (latency-wise), we draw a red dashed horizontal line to represent the latency service-level objective (SLO), set at 30 *ms* and 50 *ms* for short context and long context length respectively. This line intersects with each latency curve at a specific number of decode requests. We define the Latency-Constrained Utilization Point (LCU Point) as the point on the batch latency curve where it intersects the latency service-level objective (SLO). The corresponding throughput value, as projected on the TFLOPs curve, represents the achievable throughput bound under the given latency constraint. The position of the LCU Point varies with prefill and decode lengths, making it a key indicator for batching efficiency under real-time workload conditions.

From these results, we derive three key insights:

Insight 1: Decode-only batches meet latency targets but underutilize GPUs. Across both decode lengths, decode-only configurations maintain latency well below the 50 ms SLO, regardless of decode request count. However, the corresponding TFLOPs are significantly lower than mixed configurations. Decode-only batches are memory-bound and cannot fully utilize GPU's compute capabilities. As a result, their SLO intersection point is high, but their peak throughput under that constraint is limited, highlighting the inefficiency of decode-only execution.

Insight 2: Adding prefill tokens improves utilization but at a latency cost. Introducing a moderate prefill length (e.g., 512 tokens) boosts GPU utilization and throughput by better overlapping compute- and memory-intensive phases. However, this also increases batch latency. For instance, with decode length 1024, the SLO intersection point for the 512-token prefill batch occurs at 29 decode requests—beyond this, latency exceeds the 50 ms target. While these configurations offer significantly higher TFLOPs before the SLO intersection point, they cannot maintain responsiveness as workload intensity increases.

Insight 3: Prefill length and decode number are key performance drivers—but the optimal configuration is dynamic. Larger prefill chunks (1024 or 2048 tokens) may yield higher throughput in low-latency-tolerant settings but rapidly exceed SLO thresholds as decode demand grows. Similarly, longer decode sequences exacerbate memory contention and delay handoff between phases. There is no universally optimal static configuration—the best batching strategy depends on the precise mix of request lengths, system load, and latency goals. This variability calls for intelligent request partitioning and scheduling that responds to changing workload conditions.

Insight 4: Context-aware batching is essential for scalable and efficient throughput. Throughput behavior varies significantly with context length. For short prompts (e.g., prefill = 128), throughput scales with decode requests up to the LCU Point, then plateaus—forming a roofline pattern. Here, the LCU Point defines a practical upper bound. For long prompts (e.g., prefill = 1024), throughput degrades as decode load increases due to resource contention. These patterns suggest that batching policies must adapt to context length, scaling decode requests aggressively for short contexts, and conservatively for long ones to avoid oversaturation and SLO violations.

Taken together, both prefill and decode number significantly shape batching performance, but their ideal balance is highly sensitive to real-time system dynamics. The LCU Point offers a concrete metric to evaluate throughput under latency constraints. To achieve high GPU utilization without violating SLOs, serving systems must move beyond fixed chunk sizes and static batching policies. Instead, adaptive request splitting and scheduling are crucial for handling diverse prompt lengths and dynamic loads, enabling responsive and efficient LLM inference.

4.3 System-Level Optimization Problem Formulation

Building on the insights from batch-level performance analysis, we now formulate a global optimization framework for maximizing LLM serving throughput under dynamic request splitting and batching. Specifically, we jointly optimize two key parameters: the per-request split ratio ϕ_r and the chunk size configuration C_i. The split ratio ϕ_r determines how each request is divided across instances, thereby controlling the distribution of compute and memory (HBM) load across different instances. By dynamically adjusting ϕ_r, the system can balance workload across heterogeneous stages and improve overall resource utilization. The chunk size configuration C_i, in turn, interacts with the local First-Come-First-Serve (FCFS) scheduler to shape the composition of each batch. By tuning C_i, the system controls how prefill and decode requests are grouped into a batch under the local First-Come-First-Serve (FCFS) scheduler. This directly shapes the execution time of each batch. Leveraging the batch execution model discussed earlier, adjusting C_i enables the system to ensure that batch latency remains within SLO constraints, while maximizing MFU within those limits.

Formally, we express the optimization objective as:

$$\max_{\phi,C} \sum_{i\in\mathcal{I}} Q_i = \max_{\phi,C} \sum_{i\in\mathcal{I}} \frac{\sum_{r\in\mathcal{R}_i} FLOPs(r,\phi_r)}{T(\mathcal{R}_i, C_i)} \tag{1}$$

$$\text{where} \tag{2}$$

$$\{\phi_r, r^\alpha, r^\beta, \mathcal{I}_r^\alpha, \mathcal{I}_r^\beta | r\in\mathcal{R}\} = GS(\sum_{r\in\mathcal{R}} r) \tag{3}$$

$$\mathcal{R}_i = \{r_i^\alpha\} \quad \forall i\in\mathcal{I}^\alpha, \quad \mathcal{R}_j = \{r_j^\beta\} \quad \forall j\in\mathcal{I}^\beta \tag{4}$$

$$\mathcal{B}_i = FCFS(\mathcal{R}_i, C_i), \quad \forall i\in\mathcal{I} \tag{5}$$

$$T(\mathcal{R}_i, C_i) = T(\mathcal{B}_i) = \sum_{b\in\mathcal{B}_i} T(b) \tag{6}$$

$$\text{s.t.} \tag{7}$$

$$T(b) \le r_{\text{ITL}}, \quad \forall b\in\mathcal{B}_i, r\in b, \forall i\in\mathcal{I} \tag{8}$$

$$M(b) \le HBM, \quad \forall b\in\mathcal{B}_i, \forall i\in\mathcal{I} \tag{9}$$

To maximize overall GPU throughput in LLM serving, we formulate the problem as a joint optimization over the per-request split ratio ϕ_r and the chunk size configuration C_i for each instance. For every incoming request $r\in\mathcal{R}$, the global scheduler GS determines a split ratio ϕ_r, partitions the request into its α and β components r^α and r^β, and assigns each component to a specific instance, denoted $\mathcal{I}_r^\alpha$ and $\mathcal{I}_r^\beta$, respectively. As a result, the workload assigned to each instance $i\in\mathcal{I}$ is defined by a request set $\mathcal{R}_i$, where $\mathcal{R}_i = \{r_i^\alpha\}$ for α instances and $\mathcal{R}_j = \{r_j^\beta\}$ for β instances.

Within each instance, a local First-Come-First-Serve (FCFS) scheduler groups requests into batches $\mathcal{B}_i$, controlled by a local chunk size C_i. The total execution time of instance i is then given by the sum of its batch execution times, $T(\mathcal{R}_i, C_i) = \sum_{b\in\mathcal{B}_i} T(b)$. The total FLOPs consumed by each instance is computed over the assigned requests, where each request's computational cost is a function of its PD ratio, represented as $FLOPs(r, \alpha_r)$. The overall objective is to maximize the sum of per-instance FLOPs throughput, that is, the total compute load divided by the total execution time for all instances.

The scheduling and batching process is subject to several constraints. First, each request must satisfy its latency service-level objective (SLO), which bounds the execution time $T(b)$ of any batch that includes request r to be no more than its inter-token latency requirement r_{ITL}. Second, the memory usage of each batch $M(b)$ must remain within the GPU's high-bandwidth memory (HBM) limit. Additionally, to preserve the request's semantic execution order, once the prefill phase completes, the decode component must be forwarded to the assigned decode instance, with the decode arrival time r_{dtime} calculated based on the completion of its prefill batch sequence.

This formalization allows the system to reason globally over compute, memory, and latency trade-offs-enabling dynamic, fine-grained scheduling decisions. By tuning ϕ_r and C_i in response to workload conditions, we can approach near-optimal throughput while satisfying stringent SLOs in dynamic environments.

5 Design and Implementation

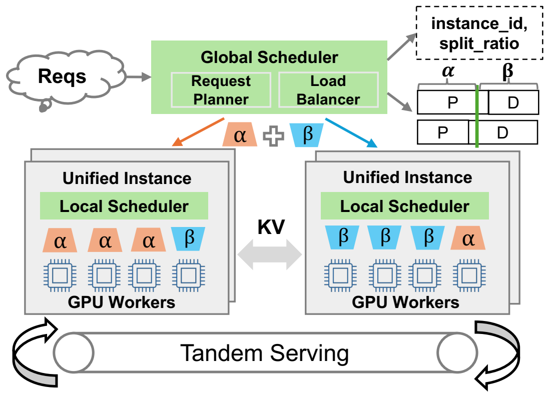


Figure 5. System overview for unified instance and dynamic serving scheduler

DynaServe is a scalable and adaptive LLM serving system designed for large-scale GPU clusters operating under highly dynamic workloads. It addresses the core limitations of static colocation and rigid PD disaggregation by introducing a Tandem Serving model, where requests are elastically split and jointly executed across distributed GPU instances. As illustrated in Figure 5, DynaServe consists of two main components: a centralized Global Scheduler and a distributed Unified Execution Engine.

Global Scheduler The Global Scheduler orchestrates execution across the cluster by continuously monitoring compute and memory usage via its Resource Monitor. Based on real-time system states, the Request Planner determines how to decompose each incoming request into two virtual sub-requests: α and β. Sub-request α typically handles most of the prompt (prefill) and potentially a small portion of token generation, while β carries out the remainder of prompt and the decoding phase. This asymmetric partitioning enables fine-grained control over execution granularity and better resource balance across heterogeneous requests.

Load Balancer The Load Balancer then maps each sub-request to a specific instance, using placement strategies that consider both split ratio and system utilization. The output is a scheduling tuple—`(instance_id, split_ratio)` for each request, which guides the distributed execution of the two sub-tasks.

Unified Execution Engine On the execution side, each Unified Instance runs a Local Scheduler that supports flexible execution of all sub-request types, including prefill-only, decode-only, hybrid, and chunked segments. These local schedulers apply a first-come-first-serve (FCFS) batching strategy, while respecting constraints like high-bandwidth memory (HBM) limits and inter-token latency (ITL) SLOs.

Crucially, sub-requests α and β can be processed on different instances without compromising correctness or performance. This is enabled by KV cache transfer across instances, ensuring that contextual information is preserved even when a request spans multiple devices. By eliminating the need for rigid execution roles and enabling coherent distributed execution, the Unified Execution Engine maximizes hardware utilization and system responsiveness.

Tandem Serving as realized by DynaServe, unifies the strengths of colocated and disaggregated LLM serving while mitigating their weaknesses. Through elastic request decomposition and system-aware scheduling, DynaServe achieves robust performance under diverse workload patterns, delivering higher throughput, better resource balance, and consistent adherence to latency objectives.

6 Evaluation

In this section, we evaluate the performance of DynaServe with state-of-the-art frameworks on real-workload traces and show the effectiveness of our techniques.

6.1 Experimental Setup

Model. We use the Llama-3.1-8B model [12] and Qwen-2.5-14B [3] as the standard model for most of our tests, which is one of the most popular open-source models in the world, widely adopted by the industry and academia.

Testbed. We evaluate DynaServe on servers each with eight NVIDIA A100 80 GB GPUs, 128 CPUs, 1024 GB of host memory, and four 200 Gbps ConnectX-6 RoCE NICs. The NVLink bandwidth between two GPUs is 600 GB/s. We use PyTorch 2.5.1, CUDA 12.4 for our evaluation. Most experiments are conducted on two servers, and we also evaluate the multi-node performance of DynaServe on a large scale cluster.

To verify the generality of our method, we also evaluate our systems on various types of hardware, A40 servers, each with eight NVIDIA A40 A40 GB GPUs, 192 CPUs and 1.48T memory, connected by two 200GBps ConnectX-5 NICs.

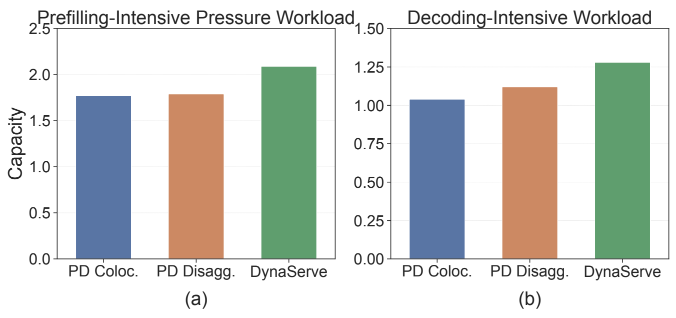


Figure 6. Serving capacity comparison for DynaServe, PD disaggregation and PD colocation under two workloads

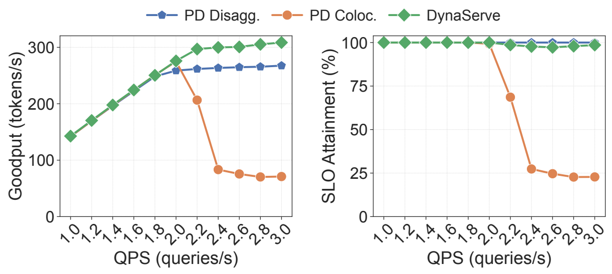


Figure 7. Dynamic P-to-D throughput across various QPS

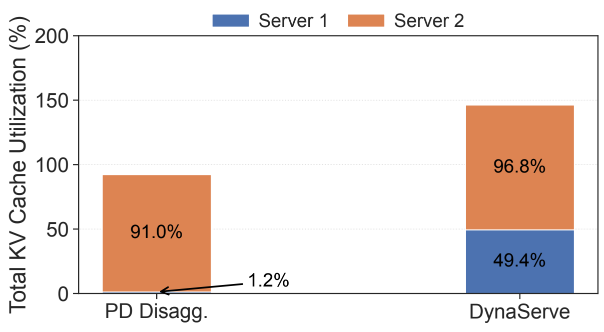


Figure 8. Total KV cache utilization across two GPU instances

Workloads. Similar to prior work [7, 17, 21], the arrival pattern of requests is generated by a Poisson process. The input lengths and output lengths of requests are sampled from the following real-world datasets such as Azure Code [2] and ShareGPT [11]

Baselines. We compare DynaServe with the following state-of-the-art LLM serving system:

- **vLLM.** It is one of the most popular LLM serving systems. In its latest implementation, chunk prefill is its default batching policy.
- **vLLM-PD.** To fully verify the advantage of it, we also configure vLLM to disaggregated prefill mode.

We regard these two baselines as PD colocation (PD Coloca.) and PD disaggregation (PD Disagg.) in the following paragraphs.

Metrics . We evaluate serving throughput under SLO latency constraints, focusing on serving capacity—the maximum queries-per-second (QPS) a system can handle while meeting a target latency, as defined in Sarathi Server [1]. Following prior work [17, 21], we also report goodput (tokens/s) under varying QPS with SLO guarantees. Additionally, we examine resource utilization, including Model FLOPs Utilization (MFU) and GPU memory usage (MBU).

6.2 Overall Result

We evaluate DynaServe against two baselines, PD disaggregation (vLLM-PD) and PD colocation (chunk prefill based vLLM) on a two-GPU system using real-world workloads traces. Experiments cover both prefill-intensive and decode-intensive scenarios, with a latency SLO of 100 ms.

Serving capacity. As shown in Figure 6, DynaServe consistently achieves higher serving capacity under both workload types. In the prefill-heavy workload (Figure 6-(a)), where prefill length is 100 × the decode length (e.g., Azure Code), DynaServe dynamically offloads portions of the prefill phase to the second GPU, balancing load and maximizing utilization. This results in up to 1.17 × higher capacity compared to PD disaggregation and 1.18 × over PD colocation.

In the decode-heavy workload (Figure 6-(b)), where the decode-to-prefill ratio is high, memory pressure on the decoding GPU causes bottlenecks in static schemes. DynaServe adaptively redirects part of the decoding workload to the server-1's GPU, alleviating memory saturation. This achieves up to 1.14× serving capacity improvement over disaggregation and 1.23× over colocation. These results highlight the effectiveness of elastic request scheduling in adapting to heterogeneous workloads.

SLO Attainment. To further evaluate responsiveness, we measure goodput (tokens/s) and SLO attainment under increasing query loads using the long-prompt workload (Figure 7). DynaServe sustains up to 1.15 × higher goodput than PD Disagg. and 4.34 × higher than PD Coloc., while maintaining near-perfect SLO attainment. In contrast, PD colocation degrades sharply under high QPS due to resource contention, and PD disaggregation suffers from static load imbalance. DynaServe 's flexibility allows it to scale while meeting latency constraints.

Memory Measurement. Then, we also verify why decode-to-prefill can achieve a performance gain. Figure 8, explains the performance gains by comparing memory usage. In decode-heavy scenarios, PD disaggregation overloads the decode-side GPU (server-2) while underutilizing the prefill GPU (server-1). DynaServe effectively reuses idle memory on server-1 by migrating decode tasks, improving global resource utilization and enabling higher throughput without violating memory limits.

7 Other Related Work

Optimizing Large Language Model (LLM) inference involves various strategies aimed at enhancing throughput and reducing latency. Recent research has explored optimizations at different granularities, including request-level, phase-level, operation-level, and parallelism-level approaches.

Request-Level Batching. Traditional batching methods often wait for all requests in a batch to complete before processing new ones, leading to potential inefficiencies. Orca [19] introduces continuous batching, an iteration-level scheduling technique that dynamically refills the batch as soon as a request is processed. This approach maximizes batch size and improves throughput by allowing new requests to enter the processing pipeline without waiting for the current batch to finish. Recently, MLC-LLM [14] adopts a static migration strategy that offloads prefill to decode instances, but this ratio is fixed for all requests and must be manually tuned. Moreover, it does not support migrating the decode phase. In contrast, DynaServe introduces a general and flexible abstraction that enables dynamic request decomposition and supports fine-grained, adaptive scheduling of both prefill and decode phases across instances.

Phase-Level Scheduling DistServe [21] and Splitwise [15] propose disaggregating these phases into separate clusters. By assigning prefill and decode computations to different GPUs or machines, these systems reduce interference between the two phases and allow for phase-specific resource optimization, thereby enhancing overall performance.

Chunked Prefill Policy DeepSpeed-FastGen [5] and Sarathi-Serve [1] introduce a chunked prefill strategy, which splits a prefill request into multiple smaller chunks. This method enables the batching of prefill and decode requests together, improving resource utilization and reducing latency. By processing smaller chunks, the system can better manage computational resources and maintain a balance between throughput and latency.

Operation-Level Optimizations. Beyond scheduling strategies, specific operation-level optimizations have been proposed to tackle inference inefficiencies. For instance, PagedAttention [7] enhances memory management during attention operations by organizing the key-value cache in a paged manner, reducing memory fragmentation and improving access efficiency. Additionally, speculative decoding [13] techniques aim to predict and verify future tokens to expedite the decoding process. Nanoflow [22] explores the overlapping of communication and computation among nano-batches to enhance intra-device parallelism.

Inference Parallelism. Tensor parallelism is a standard technique in existing frameworks like vLLM [7] and SGLang [20], enabling the distribution of model computations across multiple GPUs. LoongServe proposes elastic sequence parallelism, which dynamically adjusts resource allocation during the decode phase to efficiently handle super long-context inference. Expert parallelism is also applied in Mixture of Experts (MoE) models [10], distributing experts across multiple devices to facilitate large-scale parallel computation of specialized sub-networks.

Collectively, these approaches represent significant advancements in optimizing LLM inference, addressing challenges at multiple levels to enhance efficiency and performance. DynaServe sits at the scheduler layers, seamlessly working with this optimization and further improving the resource utilization.

8 Conclusion

We introduced DynaServe, a scalable and adaptive LLM serving system that unifies and elastically executes prefill and decode stages to handle the highly dynamic nature of real-world workloads. By leveraging the tandem serving abstraction, DynaServe enables fine-grained request decomposition, dynamic scheduling, and hybrid batch execution across distributed GPU instances. This approach bridges the gap between colocated and disaggregated serving, achieving both high resource utilization and low latency.

References

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

[2] Azure. Azurepublicdataset. https://github.com/Azure/AzurePublicDataset/tree/master, 2024. "[accessed-April-2025]".

[3] Alibaba Cloud. Qwen/qwen2.5-14b-instruct. https://huggingface.co/Qwen/Qwen2.5-14B-Instruct, 2024. "[accessed-April-2025]".

[4] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-decoding for long-context inference. https://pytorch.org/blog/flash-decoding/, 2023. "[accessed-April-2025]".

[5] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

[6] Aditya K Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-attention: Unlocking full prefill-decode overlap for faster llm inference. *arXiv preprint arXiv:2410.18038*, 2024.

[7] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[8] Yunkai Liang, Zhangyu Chen, Pengfei Zuo, Zhi Zhou, Xu Chen, and Zhou Yu. Injecting adrenaline into llm serving: Boosting resource utilization and throughput via attention disaggregation. *arXiv preprint arXiv:2503.20552*, 2025.

[9] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, Shen Li, Zhigang Ji, Tao Xie, Yong Li, and Wei Lin. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache, 2024.

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

[11] LucidityAILabs. Lucidityai/sharegpt-reasoning. https://huggingface.co/datasets/LucidityAI/sharegpt-reasoning/tree/main, 2024. "[accessed-April-2025]".

[12] Meta. meta-llama/llama-3.1-8b-instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, 2024. "[accessed-April-2025]".

[13] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[14] MLC team. MLC-LLM, 2023-2025.

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

[16] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[17] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 640–654, 2024.

[18] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[19] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[20] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[21] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[22] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.