Libra: Flexible Request Partitioning and Scheduling for Serving Unbalanced and Dynamic LLM Workloads

Chaoyi Ruan[1], Yinhe Chen[2], Dongqi Tian[2], Yandong Shi[2], Yongji Wu[3], Jialin Li[1], and Cheng Li[2,4]

[1]National University of Singapore, [2]University of Science and Technology of China, [3]UC, Berkeley,
[4]Institute of Artificial Intelligence, Hefei Comprehensive National Science Center

Abstract

LLM inference must meet strict latency SLOs while maximizing throughput. Yet, real-world variability in prompt and response lengths skews compute-intensive prefill and memory-bound decode phases, making both colocated (even with chunked prefill) and disaggregated deployments unable to simultaneously deliver low tail latency and high throughput.

We introduce Libra, a high performance LLM serving system that maximizes goodput under SLO constraints even when handling imbalanced and dynamic workloads. At the core of Libra is a *micro-request* based *flexible partitioning and scheduling* (FPS) abstraction. The abstraction splits each request at any token boundary into multiple cooperating segments. Libra then designs a *two-level scheduling* framework that balances micro-request load across unified GPU instances. The framework consists of a global scheduler that selects per-request split points, and a local scheduler on each GPU instance to form SLO-aware batches. Finally, Libra uses chunked KV cache transfers to support cross-instance micro-request execution. On real-world traces, Libra improves goodput by up to 1.91× and 1.61×, increases serving capacity from 1.15× to 3.07×, and improves serving performance by up to 74.2% in a hybrid workload under strict SLOs and A100/H100 GPUs compared to state-of-the-art colocated and disaggregated baselines.

1 Introduction

Large language models (LLMs) power modern applications from code generation [5, 12] to chatbots [32, 39] and scientific assistants [10]. An LLM serving system must handle high volume of concurrent inference requests, each imposing stringent latency SLOs to preserve user experience. In particular, modern systems often target a 100 ms P99 time-between-tokens (TBT) bound to achieve fluid, real-time generation [41]. Also, service providers must maximize throughput to improve cost-efficiency and resource utilization. Balancing tight latency SLOs with high throughput is critical yet challenging.

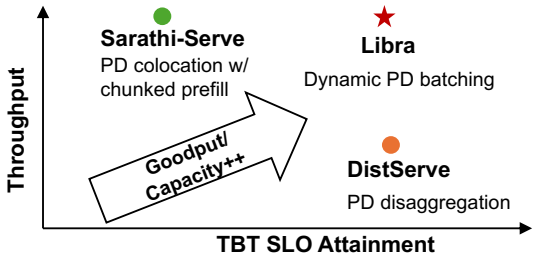

Figure 1: Throughput vs. SLO attainment across serving architectures. PD colocation with chunked prefill reaches high throughput but violates the latency SLO. PD disaggregation satisfies the SLO but under-utilizes GPUs. Libra balances the two, advancing the frontier toward the top-right with higher goodput (i.e., effective throughput while meeting SLO).

The LLM inference process naturally consists of two stages. The *prefill* stage processes all prompt tokens in parallel to generate the first token and populate the KV cache, while *decode* stage then produces subsequent tokens one-at-a-time. Prefill workloads are compute-intensive, whereas decode workloads are memory-bound. Because they share weights and KV cache, conventional serving systems typically colocate both phases on the same GPU instance to amortize resource usage [40]. However, this simple solution causes interference: long prefill batches stall latency-sensitive decode steps, inflating TBT tail-latencies, while heavy decode phases can degrade prompt-processing responsiveness.

To mitigate interference, *chunked prefill* extends the conventional colocation solution by splitting long prompts into smaller segments so decodes can interleave with prefill chunks [2, 16]. However, it only provides coarse latency bounds. *PD disaggregation* [41] partitions the request to assign prefill and decode to separate GPU instances, eliminating interference entirely but often under-utilizing hardware due to mismatched stage loads. Moreover, given a fixed, pre-allocated pool of GPUs, it is challenging to determine a priori

Chaoyi Ruan and Yinhe Chen equally contributed to this work. And Cheng Li and Jialin Li are corresponding authors.

the optimal split of GPU resources between prefill and decoding. Real-world workloads are dynamic and mixed, with prompt lengths and generation sizes varying widely (as shown in §2.3). Consequently, any fixed partitioning will inevitably be suboptimal. Figure 1 illustrates the latency-throughput trade-offs between the two paradigms, and their drawbacks are magnified under dynamic, imbalanced workloads.

In this paper, we introduce a new paradigm for LLM serving, termed *Flexible Partition and Scheduling* (FPS). FPS treats all GPUs in a cluster as a single unified resource pool. At its core is a new *micro-request abstraction* that splits individual LLM requests at any token boundary. The abstraction enables dynamic partitioning at the finest granularity on a per-request basis. FPS eliminates the rigid trade-off between colocation and disaggregation, adapting GPU resource allocation in real-time to match workload composition, balance load across GPUs, and minimize interference to meet tight SLOs. Crucially, it can *generalize* to handle complex, multi-stage inference patterns like Chain-of-Thought by partitioning a request into more than two segments.

In this work, we present a concrete instance of this FPS paradigm, Libra. At the top level, Libra implements a global scheduler that predicts decode length roughly and stage-wise costs for each inference request. It then quickly searches the vast space of possible split token positions, often tens to thousands per request, to identify near-optimal micro-request partitions within just a few milliseconds. By leveraging lightweight cost models and a handful of probes, it balances per-request latency constraints against overall GPU load, then routes micro-requests in round-robin fashion to the unified GPU pool for fine-grained execution.

Within Libra, all GPU instances are equal and unified, unlike two GPU roles in PD disaggregation. Any instance can process any micro-request. A local scheduler on each GPU instance carefully composes incoming micro-requests into batches. For each batch, it tunes three key factors, i.e., batch size, prefill-to-decode token ratio, and decode context length, to sustain high utilization under a P99 TBT SLO. This per-batch adaptation also takes place ultra-fast and prevents stalls and idle cycles, even as workloads shift. Additionally, we introduce a chunk-based KV-transfer mechanism that efficiently manages the fine-grained KV cache transfers between unified GPU instances induced by micro-request partitioning.

We comprehensively evaluate Libra on A100/H100 GPU clusters using real-world workloads such as BurstGPT [36] and Azure Code [4]. On A100 GPUs, Libra consistently outperforms prior designs: It attains 1.15×–3.07× higher serving capacity and up to 1.91× higher goodput than PD colocation; it achieves 1.09×–1.67× higher capacity and up to 1.61× higher goodput than PD disaggregation; Libra also improves performance by 60%/25% in a hybrid workload over PD colocation/disaggregation. Libra's adaptability shines even against stronger asymmetric baselines where more GPUs are allocated to the bottleneck stage, improving per-GPU goodput by up to 74.2% against asymmetric tensor parallelism and 22.8% against asymmetric data parallelism. Furthermore, on newer H100 hardware and under a stricter SLO, Libra extends its performance gains, boosting goodput by up to 134%. These results highlight Libra's efficiency and robustness across diverse workloads, hardware generations, and strict production-like latency constraints.

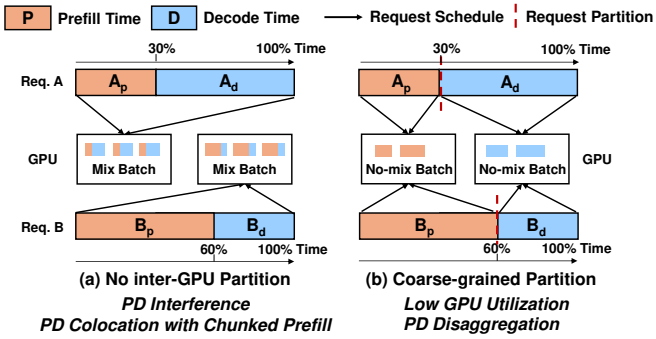

Figure 2: Partition and scheduling strategies in LLM serving. (a) PD Colocation, applying chunked prefill to reduce interference. (b) coarse-grained PD disaggregation, which avoids interference but leads to GPU under-utilization.

2 Background and Motivation

2.1 LLM Inference

LLM inference [33] consists of two distinct stages, where a *prefill* stage is followed by a *decode* phase. In the prefill stage, the model processes the entire input prompt in a single forward pass, which computes the key-value (KV) cache for all tokens and generates the first output token. The input tokens' KV cache is stored in GPU. Then the decode stage generates the remaining output tokens one by one, each using the accumulated KV cache from previous steps, until a stopping criteria is met (e.g., token limit or stop word). It is well-known that the prefill stage is compute-intensive, while decode is memory-bound with low arithmetic intensity. Since all input prompt tokens are available up front, they can be processed in parallel during the prefill stage, achieving high GPU utilization. In contrast, the decode stage emits one token at a time, serializing work on the GPU and leaving many compute units idle. The relative intensity of these two phases strongly depends on the lengths of the input and output sequences.

2.2 LLM Serving Systems and Optimizations

Real-time online LLM services must simultaneously handle concurrent requests while meeting strict SLOs. Key latency metrics include TTFT (time to first token) for prefill speed and TBT (time between tokens) for decode efficiency. The sum of TTFT and the cumulative TBT determines overall request latency; a bottleneck in either phase directly degrades user experience. Overall throughput or GPU utilization, driven by concurrency, is another major metric for mainstream serving systems. Since latency and throughput often conflict, most sys-

tem optimizations revolve around balancing these objectives. Our goal is to maximize LLM serving goodput [41], defined as the number of output tokens generated per second under a latency SLO. We begin by reviewing two common serving architectures (Figure 2) and their performance trade-offs.

No inter-GPU partition (PD colocation). The simplest architecture runs both prefill and decode phases of a serving request on the same instance (Figure 2-(a)) without any partition. Colocation simplifies scheduling and enables straightforward continuous batching [40]. But since each instance handles prefill and decode phases of different requests concurrently, the scheme creates prefill-decode interference: heavy prefilling workloads may stall latency-sensitive decode steps, leading to long tail latencies between tokens. To mitigate PD interference, recent systems like Sarathi-Serve [1] introduces *chunked prefill*. As illustrated in Figure 2-(a), the approach breaks an input prompt into smaller segments, each interleaved with decode execution. POD Attention [16] further fuses prefill and decode into a single GPU kernel, improving their overlap and GPU utilization. However, effectiveness of chunked prefill is highly workload-dependent. Only when the decode phase dominates (e.g., reasoning tasks where outputs can be 10× longer than inputs), can chunked prefill maintain acceptable tail latency as the shorter prefill stage causes minimal contention. Furthermore, their latency and throughput are sensitive to the chunk size hyperparameter. Mainstream solutions [28, 34] choose a static, coarse-grained chunk size, falling short when facing dynamic workloads unfortunately.

Coarse-grained partition (PD disaggregation). To eliminate interference, PD disaggregation (Figure 2-(b)) splits reuqests statically at the end of prompt, separating prefill and decode stages onto dedicated GPU pools. Systems such as DistServe [41], Mooncake [25], and Splitwise [23] apply such an architecture and tune each phase independently. Physical isolation offers more stable tail latencies. The architecture achieves optimal throughput when computation time between prompt processing and output generation are well-balanced. However, any imbalance such as prompt lengths increase or output lengths shrink, creates uneven resource utilization across the two GPU pools, resulting in resource fragmentation and degraded overall efficiency. This sensitivity to workload skew presents a key challenge for disaggregated designs in serving where prompt/output patterns are highly dynamic.

2.3 Unbalanced and Dynamic Workloads

We examine the Azure Code [23] and BurstGPT [37] request traces to understand the time-varying imbalance between the compute demands of the prefill and decode stages in LLM inference. Figure 3 shows the prompt token count (blue) and response token count (orange) from two real-world traces: (a) per-minute from Azure Code and (b) per-day from BurstGPT. We also plot a "balanced" decode curve (green), which shows the number of output tokens whose decode time would exactly match the prefill time. This calculation is done using

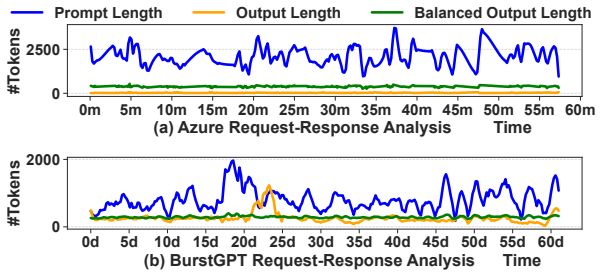

Figure 3: Prompt and output token lengths distribution. The green line indicates the theoretical balanced output length, where decode time equals prefill time.

Table 1: GPU compute, KV cache usage, and inter-token latency when serving Qwen-2.5-14B [6] on two A100 GPUs (G1 and G2) under PD disaggregation and colocation. Request rates are tuned to saturate the GPUs.

Metric		P-8192, D-32		P-2048, D-512		P-219, D-1467	
		Disagg.	Coloc.	Disagg.	Coloc.	Disagg.	Coloc.
MFU (%)	G1	43.24	38.94	30.59	21.26	2.09	13.98
	G2	0.19	38.94	7.93	21.26	14.34	13.98
HBM Usage (%)	G1	4.83	39.67	1.17	94.23	0	95.2
	G2	5.77	40.03	94.83	95.7	96.27	95.33
p50-TBT (ms)		**22.70**	309.68	**47.00**	**46.86**	**50.63**	**47.99**
p99-TBT (ms)		**58.09**	352.93	**65.11**	336.81	**74.47**	162.67
Throughput (rps)		0.83	**1.49**	2.56	**2.83**	1.68	**2.86**
Attainment (%)		**100.00**	1.73	**99.83**	84.09	**99.95**	94.46

measured prefill throughput on the A100 GPU. The balanced line remains relatively flat because prefill processes tokens in parallel with high compute efficiency, so even large variations in prompt require only modest equivalent decode tokens to match the prefill time.

In areas where the yellow line exceeds the green curve, decode demands more GPU time than prefill; otherwise, prefill dominates. The trace exhibits wide swings in both prompt and response volumes. There are extended regions where the response line (yellow) exceeds the balanced curve, indicating decode-heavy periods, followed by regions of prefill dominance. The trace also shows rapid fluctuations between the two types of regions, underscoring the presence of high temporal variance in real-world workloads. Similar dynamics also appear at scale. For instance, Alibaba's Infinite-LLM reports context lengths ranging from a few tokens to more than two million in production [20].

2.4 Issues with Existing Techniques

To precisely diagnose the limitations of existing architectures, we conduct a controlled micro-benchmark on a two-GPU system using three representative request shapes: prefill-heavy (P-8192, D-32), decode-heavy (P-219, D-1467), and balanced (P-2048, D-512). This minimal setup is intentionally chosen to reveal the inherent inflexibility of current designs.

For PD Disaggregation, we use a 1P:1D setup, as it is the only possible setup on two GPUs. It immediately exposes its rigidity: bottleneck stage cannot be provisioned with more

resources. For PD Colocation, we enable chunked prefill with vLLM's default (2048 tokens) to show its coarse-grainularity. In all cases, request rates are pushed to saturation to evaluate performance under a 100 ms TBT SLO.

Analysis of PD Disaggregation. Table 1 shows that PD Disaggregation's strength is its guaranteed latency. By physically isolating stages, it consistently satisfies the 100 ms SLO across all workloads, with P99-TBT remaining below 75 ms. However, this comes at a steep cost to utilization. The rigid 1P:1D partitioning creates severe resource imbalance. For the prefill-heavy workload (P-8192, D-32), the prefill GPU (G1) is compute-bound at 43.2% MFU while the decode GPU (G2) is almost completely idle, delivering a mere 0.19% MFU. Conversely, for the decode-heavy workload (P-219, D-1467), G1 is left idle while G2's memory is saturated. Even in the balanced case, resources are mismatched: G1 underuses memory, and G2 underuses computation. This inability to share work across the rigid boundary leads to significant underutilization and, as a result, lower overall throughput.

Analysis of PD Colocation. In contrast, PD Colocation appears to offer better throughput by achieving balanced resource utilization across both GPUs. For instance, in the prefill-heavy case, it achieves 1.49 rps compared to disaggregation's 0.83 rps. However, this throughput is illusory, as the system completely fails to meet its latency contract. The shared-resource model creates uncontrollable interference. For the long-prompt workload, head-of-line blocking from prefill tasks pushes the P99-TBT to a catastrophic 352.93 ms. The problem persists even in the balanced workload (336.81 ms) and the reasoning workload (162.67 ms), where interference should theoretically be lower. With SLO attainment dropping to as low as 1.73%, colocation effectively trades latency for a higher throughput that is ultimately unusable in a production environment that values user experience.

The Rigidity of Static Partitioning. Together, these findings expose a fundamental dilemma. Colocation offers high potential utilization but no latency guarantees. Disaggregation offers latency guarantees but suffers from low utilization due to rigid resource allocation. While one might suggest tuning these configurations for each workload (e.g., using a smaller chunk size or provisioning a 3P:1D ratio in a larger cluster), our observation highlights a core problem: static partitioning is brittle. Under colocation, chunk-size tuning only amortizes prefill and decode interference coarsely, as prefill and decode workloads remain fundamentally mixed. A system optimally tuned for a prefill-heavy workload is, by definition, sub-optimal the moment the traffic mix shifts towards being decode-heavy. Since real-world service providers must serve dynamic and unpredictable workloads on a fixed and expensive pool of GPUs, they cannot afford to have hardware sit idle. Any static configuration is doomed to inefficiency. Furthermore, real-time changing the parallelism strategies will introduce extra overheads.

This motivates our work. We need to move beyond static, cluster-level provisioning and toward dynamic, request-level adaptation. The goal is to design a system that can create unified resources, dynamically aligning GPU resources with the immediate needs of the workload, thereby achieving the best of both worlds: the high, guaranteed SLO of disaggregation and the high resource utilization of colocation.

3 Libra Overview: Approach and Challenges

Our benchmark results (§2.4) reveal a latency–throughput trade-off in existing serving architectures. In this section, we give an overview of Libra with FPS, a framework designed for efficient handling of dynamic, imbalanced LLM workloads.

3.1 FPS Abstraction

The key design principle in Libra is *Flexible Partition and Scheduling* (FPS). As shown in Figure 4, FPS proposes a new micro-request abstraction that allows partitioning the prefill/decode phases of a serving request at *any* token position. Partitioning and dispatching decisions are centrally made by a global scheduler based on collected latency and resource utilization metrics. Finally, a runtime system local to each GPU server schedules, batches, and executes the assigned micro-requests to maximize throughput under latency SLO.

Micro-request abstraction. Each serving request r can be described by its prompt length P and a decode length D; we estimate the decode length roughly using prior well-studied length-prediction methods [26, 29]. This results in a total logical length of $L = P + D$. We associate each request with a splitting point s between 0 and L, dividing the request into two *micro-requests*: r^{α} (tokens $1 \ldots s$) and r^{β} (tokens $s{+}1 \ldots L$). When s is at the request boundary (0 or L), one of the micro-requests (r^{α} or r^{β}) is empty, i.e., no partitioning. A *micro-request* is a contiguous span of tokens, representing either prefill, decode, or a mixture of both. Unlike PD colocation (no partitioning) or disaggregation (split only at prompt boundary), FPS can split at any token position.

The micro-request abstraction enables FPS to adapt smoothly to workload variations. For example, when the system is underutilized or the prompt is short, FPS may avoid partitioning altogether, similar to a colocated engine by routing the full request to one GPU instance (see request D_{α} in Figure 4). When prefill and decode are balanced under high load, FPS may adopt a PD disaggregated configuration (see requests C_{α} and C_{β}). Beyond these, FPS supports hybrid splits, e.g., merging early prefill with partial decode (request A_{α}), or offloading part of prefill to run alongside decode on another GPU (request B_{β}).

Conceptually, FPS operates within a *generalized* request partitioning space; prior PD colocation and disaggregation are mere special cases within this broader execution space, representing either no partitioning or partitioning at the end of the prompt. The additional partitioning options offer FPS more flexibility in navigating the utilization-latency space while being adaptive to workload dynamism.

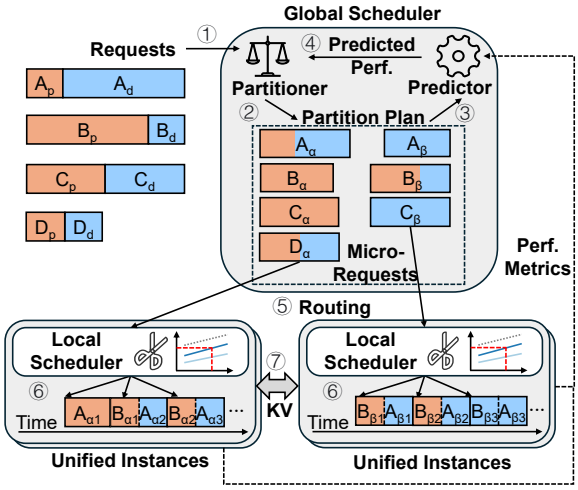

Figure 4: The overall architecture of Libra features unified GPU instances executing partitioned micro-requests, guided by a two-level scheduling for improved SLO attainment and resource utilization. Orange and blue colors denote the prefill and decode stages of each request, respectively.

3.2 Libra's Design Overview

We build Libra, a concrete instance of FPS, to overcome the limitations of both PD disaggregation and colocation, i.e., static PD role assignment leads to resource underutilization, while colocating PD causes uncontrollable interference. Libra addresses this by elevating scheduling decisions to the request level through a two-level scheduling hierarchy.

Global scheduler. All serving requests are first handled by the centralized global scheduler (①). The scheduler has two responsibilities. First, it determines the optimal splitting point s of each request. The scheduler does so by searching a candidate partition ratio $\phi \in [0,1]$ that offers the best predicted performance (②-④), and calculates the splitting point s as $\lceil \phi L \rceil$. Libra divides the sequence into two micro-requests: r^{α} (tokens $1 \ldots s$) and r^{β} (tokens $s{+}1 \ldots L$).

Second, the scheduler routes the partitioned micro-requests to executors based on current system load and latency SLOs (⑤). It runs a lightweight performance predictor, which collects periodic runtime statistics (⑧) from each GPU, including compute/HBM memory utilization, and request queue. These metrics reflect current workload pressure, allowing the predictor to estimate the performance impact of placing r^{α} and r^{β} on different instances. The scheduler selects a placement plan (micro-request to executor) that minimizes load imbalance and maximizes throughput, while meeting the request's SLO.

Unified execution instances. A unified execution instance runs micro-requests dispatched by the global scheduler on its GPUs. Given a set of buffered requests, a local scheduler builds token-level batches via an SLO-aware control mechanism (⑥). The batching protocol enforces two constraints: It should respect the HBM limit of the resident GPUs and it keeps the p99 TBT latency below the SLO. The local scheduler also adapts to changes in workload and observed request latency. It widens the batches and bumps throughput when request workload is moderate; if latency approaches the SLO, it reconfigures per-batch composition to reduce the prefill-decode interference. When the micro-requests of an LLM request span two execution instances, the instances exchange the required KV cache blocks over RDMA (⑦).

3.3 Design Challenges

Although FPS and Libra can adapt on the fly to meet SLOs and boost performance, unlocking their full potential requires overcoming three key system challenges.

Challenge 1: **dynamic request slicing and scheduling complexity.** Each incoming request exposes dozens to thousands of valid split points, and each resulting micro-request can be routed to multiple unified instances. As a result, the scheduling search space for n requests of lengths $l_1, \ldots, l_n$ grows exponentially as $\prod l_i$. The global scheduler must navigate this enormous space within a few milliseconds and with limited foresight: a split that appears balanced now may lead to overload moments later as new traffic arrives. These challenges necessitate fast, predictive cost models to estimate benefits and efficiently identify near-optimal splits while respecting per-request SLOs. This scheduling complexity is the necessary trade-off for achieving the on-the-fly adaptability that static systems lack.

Challenge 2: **fine-grained batch control under SLOs.** Unified instances execute a continuous sequence of batches, each containing a dynamic mix of prefill and decode tokens. This composition varies over time as micro-requests are assigned by the global scheduler, leading to fluctuating workload characteristics that must be handled efficiently at the batch level. Effective batch composition depends on tuning two key factors: the *prefill-to-decode token ratio*, which governs interference between the two stages, and the *context lengths* of decode tokens, which directly impact per-token latency. Additionally, the total number of tokens per batch influences overall GPU utilization, making batching a multi-objective optimization task. To meet latency SLOs while maintaining high throughput, the local scheduler must make fast, accurate decisions when forming each batch. Unlike prior systems that rely on static chunk sizes to loosely bound latency, our setting demands fine-grained control over batch composition in response to changing request mixes, making the problem substantially more complex.

Challenge 3: **frequent cross-instance KV cache transfers.** FPS with micro-requests leads to more frequent and fine-grained KV cache transfers between unified GPU instances compared to coarse-grained PD disaggregation, as it permits splitting at arbitrary points within either the prefill or decode stages. For example, in a reasoning task from our experiments, a micro-request that merges a decode segment with the prefill results in 3× more token cache transfers than standard disaggregation, significantly increasing KV cache volume. Libra must transfer KV cache efficiently to avoid stalling.

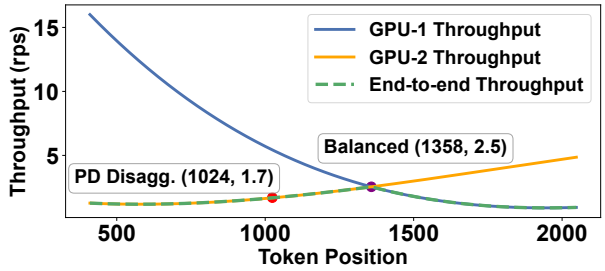

Figure 5: Throughput of Qwen2.5-32B on A100 under different split positions. Each request has 1024-token prompt and output. Position 1024 represents PD disaggregation, while position 1358 represents the optimal split found by Libra.

4 Libra's Design

The primary objective of Libra is to maximize the overall serving goodput given a fixed set of cluster resources (e.g., N GPUs). Here, goodput measures the rate of output tokens generated from requests whose tokens satisfy a per-request latency SLO. Finding the optimal micro-request to GPU schedule is a complex optimization problem. To make the problem tractable, we decompose it into a hierarchical, two-level scheduling strategy.

Libra's scheduling is guided by three key principles.

- **Balancing throughput across stages (§4.1)**: The global scheduler partitions requests to equalize execution time across GPUs, eliminating idle cycles.
- **SLO-aware batch composition (§4.2)**: The local scheduler dynamically adjusts batch composition based on runtime latency feedback.
- **Overlapping KV transfer with compute (§4.3)**: Chunk-based transfers hide communication latency behind ongoing computation.

4.1 Global: Request Partition and Routing

To better guide the design of the partition policy, we first analyze single request partitioning in terms of performance. We conducted a controlled micro-benchmark using a synthetic workload, where each request consists of a fixed 1024-token prompt and a 1024-token output. The system runs on two A100 GPUs. We vary the partition position, which refers to the token index at which the request is split between the two GPUs, and observe its impact on throughput.

In Figure 5, a partition position of 1024 corresponds to vanilla PD disaggregation: the entire prompt is handled by GPU-1, and the entire decode phase by GPU-2. This static partitioning results in highly unbalanced workloads. GPU-1, processes prefilling tasks much faster than GPU-2, which is burdened with the full decode workload. Because the two GPUs operate as a pipeline, the overall system throughput is bottlenecked by the slower of the two—GPU-2 in this case.

We then gradually shift the partition point forward, incrementally assigning more decode tokens to GPU-1. When execution times are unbalanced (e.g., at position 1024), GPU-1 finishes faster and idles waiting for GPU-2, creating stalls. As we rebalance the work by moving the partition point, GPU-1's throughput decreases while GPU-2's increases until they intersect at the optimal point (position 1358). *It is at this intersection—where the workloads on both GPUs are balanced—that the end-to-end system throughput is maximized.* This leads to our key design principle—**Insight 1: Maximizing the throughput of a disaggregated serving system requires balancing the throughput of its stages.**

Algorithm 1: GLOBAL REQUEST SCHEDULER (r)

Input: $r = (P, D, L, K)$ *// prompt, predicted-decode, current server load, maximum step for binary search*
Output: partition ratio ϕ

```
if prefill_clk = 0 ∧ decode_clk = 0 then  // cold start
    return ColdStart(r)
φ ← P/(P+D), lo ← 0, hi ← 1 // starting point
for k = 1 to K do
    (r1, r2) ← Split(r, φ) // total execution time on each server
    T1, T2 ← Predict(r1, r2, L);
    if |T1 − T2| ≤ ε then
        break                  // execution time balanced
    else
        Update(lo, hi, φ)      // binary search update
Commit(r1, r2);
return φ
```

Throughput Equilibrium Constraint. Thus, we impose a throughput equilibrium constraint for any pair of GPUs (g_i, g_j) processing a partitioned request, $|\text{Thpt}(g_i) - \text{Thpt}(g_j)| \leq \varepsilon$, where $\text{Thpt}(g)$ is the predicted request throughput (request/s (rps)), and ε is a small tolerance (e.g., 0.5 rps).

Considering the concrete example from Figure 5, at the unbalanced partition position 1024 (PD disaggregation), GPU-1 achieves 5.6 rps while GPU-2 achieves the 1.7 rps. This creates a "bucket effect" where the end-to-end throughput (1.7 rps) is bounded by the slower component. In contrast, at the balanced position 1358, both GPUs operate at 2.5 rps with synchronized completion times, eliminating idle cycles and achieving higher system throughput. Any deviation from equilibrium wastes resources and reduces utilization.

Load balancing and routing. Balancing each individual request does not guarantee balanced load globally. Concurrent micro-requests may be routed to the same GPU, causing contention. To address this, after determining the partition ratio ϕ, the global scheduler routes micro-requests using a least-loaded policy: r^{α} and r^{β} are assigned to the two instances with the lowest current load, with round-robin dispatch to break ties. The scheduler then adjusts the partition ratio to balance execution time across the selected GPU pair.

Global-level decision and request planner. Based on the above insight, the global scheduler must choose an appro-

priate partition ratio $\phi \in [0,1]$ per request for a stream of requests $\mathcal{R} = \{r_i\}$, to maximize the overall performance. As discussed in §3, the first $\lceil \phi L \rceil$ tokens of the micro-requests α are assigned to the α server, the remainder to the β server.

The algorithm 1 finds the appropriate ratio with a bounded binary search. It starts from an initial ratio $\phi = \frac{P}{P+D}$, which corresponds to pure PD disaggregation (line 3). In each iteration, the scheduler splits the incoming request r into two parts according to ϕ, and uses an analytical latency predictor to estimate the total execution time on each server, denoted by T_1 and T_2 (line 6). Here, T_1 and T_2 represent the predicted time for each server to complete all assigned micro-requests, including the new split segments, under the current load conditions. Predictions use offline-profiled lookup tables with LRU caching, costing only microseconds per probe. The binary search adjusts ϕ to balance the workload by minimizing the difference between T_1 and T_2. The search stops once the absolute difference $|T_1 - T_2|$ is within a small tolerance ε (line 7), indicating near-equal execution times on both servers. To limit overhead, the maximum number of binary search iterations is capped by the parameter K (set to 6 in our setup).

A cold-start path seeds the prefill and decode clocks when the first request arrives (line 2); subsequent requests reuse the existing timing prefix and simulate only the delta. This yields $O(1)$ per-request complexity, while staying within one percent of an offline oracle with perfect future knowledge.

Execution predictor within global scheduler. To gauge the benefit of a partition plan on two selected GPUs, the global scheduler uses a lightweight predictor that analytically estimates the plan's timing. The predictor maintains an input queue of micro-requests and simulates a virtual batch under the same hardware constraints as the runtime: a per-pass batch composition drawn from each instance's recent batching history, at most $N_{\max}$ concurrent requests, and the requirement that every active request advances by at least one token per pass. To preserve accuracy, the predictor continuously calibrates itself with recent requests execution progress and real-time GPU metrics, ensuring its model of system status remains up to date before predicting batch latency. During each virtual pass, the predictor admits requests until the token budget or request limit is reached. It grants one prompt chunk or one decode token per request, then advances time. Advancing deducts granted tokens from remaining work, resets the token budget, and stores a compact batch snapshot including request IDs, outstanding tokens, and timestamps. This cycle repeats until all work is complete, producing a detailed timeline of future GPU activity. We use First-Come-First-Serve to match vLLM setup. The predictor adds negligible overhead via lightweight state snapshots and simple arithmetic.

4.2 Local: SLO-Aware Batch Composition

Complementing the global scheduler, the local scheduler manages batching on each GPU to ensure SLO compliance.

Mix batch's performance analysis. Figure 6 tracks how

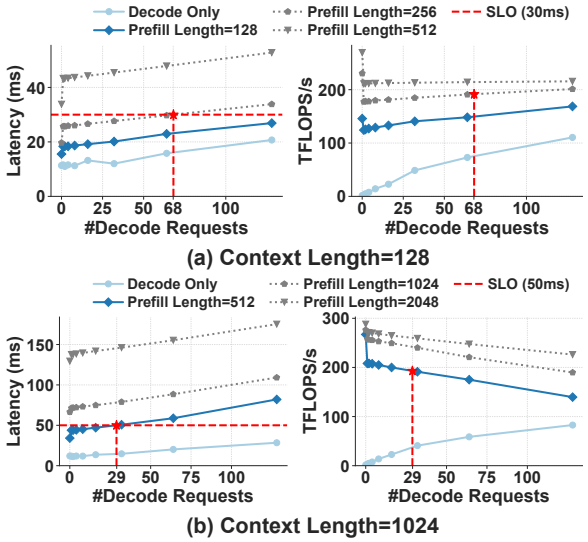

Figure 6: Latency and GPU compute utilization under different batching strategies for Llama-3.1-8B on an A100 GPU.

the mixture of prefill and decode requests in a single batch affects both latency (left-hand figures) and GPU throughput in TFLOPs/s (right-hand figures) on an A100 running Llama-3.1-8B. Two context lengths are examined: a short context of 128 tokens (top row) and a long context of 1024 tokens (bottom row). Here, context length refers to the KV cache length of concurrent decode requests, while prefill length is the token count of a newly admitted prefill request. As they belong to different requests in the same batch, prefill length can exceed decode context length. In every latency figure the red dashed horizontal line marks the latency service-level objective, 30 ms for the short context and 50 ms for the long one. Where that line intersects a latency curve we define the Latency-Constrained Utilization (LCU) point. The x-coordinate of the LCU point reveals the largest number of concurrent decode requests that can be handled without breaching the SLO, while the corresponding throughput is obtained by reading straight up to the matching TFLOPs/s curve in the plot on the right. Because varying the prefill batch size shifts the position of the LCU point, the batch composition directly determines how efficiently real-time inference workloads can be served.

From these results, we further derive the following two key insights, which we leverage in local scheduler.

Insight 2: decode–prefill trade-off between latency and GPU utilization. Decode-only batches consistently meet the 50 ms SLO but are memory-bound, leaving much of the GPU's compute capacity idle and yielding modest TFLOP/s throughput. Introducing a moderate prefill segment (e.g., 512 tokens) shifts work toward compute-bound operations, raising utilization and throughput; however, as the number of concurrent decode requests grows (for example, beyond 29 when decode length is 1024), the end-to-end latency went past the 50 ms target. Thus, higher utilization is achieved at the cost of latency increasing.

Algorithm 2: LOCAL SCHEDULER (P, D, S, B, T)

Input: P, D, S, B, T *// prefill queue, decode queue, target SLO, previous batch, profile table*
Output: next batch B

```
1 RECORD(T, B.PLEN, B.CTX, B.DNUM, B.TIME);
2 B ← D,
    M ← MAXPREFILLALLOWED(T,S,B.ctx,B.dnum);
3 foreach r ∈ P do
      // schedule no more than M tokens
4     t = min(R.TOKEN, M) ;
5     B.add(r,t);
6     M ← M − t ;                          // update budget
7     if M ≤ 0 then
8         break
9 return B
```

Insight 3: batch composition (prefill length (plen), context length (ctx) and decode token number (dnum)) is key performance drivers—but the optimal configuration is dynamic. Larger prefill batches (e.g., 1024 tokens) can boost throughput in light-load scenarios but quickly breach SLOs as batch sizes grow. Longer decode sequences further increase memory contention and delay phase handoffs. Throughput also depends on context length: short contexts (e.g., 128 tokens) exhibit a roofline pattern, rising with more decode requests until the LCU point. And long contexts (e.g., 1024 tokens) see throughput degrade under heavier decode loads. Here the LCU Point offers a concrete metric to evaluate throughput under latency constraints. These patterns suggest that request partitioning and scheduling are crucial for handling diverse prompt lengths and dynamic loads to achieve high GPU utilization without violating SLOs.

Dynamic batch composition. Based on the above insights, we design our local scheduler. After the request is split, the local inference engines need to process them. A naive scheduling policy would simply prefill a fixed batch of tokens and let the decode thread emit one token per request; the batch size then acts as a crude latency knob. In practice this is ineffective, because inference latency depends not only on the batch size but on the full batch composition—prefill length, average context length, and the number of concurrent decodes.

To make scheduling SLO-aware and hardware-efficient, we design a local scheduling algorithm that dynamically composes the batch before each hybrid kernel launch. As illustrated in Algorithm 2, the scheduler updates a profile table with latency statistics from the previously executed batch. Specifically, it records the tuple $(plen, ctx, dnum, time)$, where *time* is the measured latency. This step ensures that the profile table is progressively refined with execution feedback.

The scheduler then constructs the next batch in two stages. First, it includes all decode requests (line 2), which are assumed to be latency-critical and must be processed immediately. Based on the decode portion of the batch, it computes the average context length and decode count. These values are used to consult the profile table and determine the maximum prefill token budget M that would keep the batch latency within the target SLO. Next, the scheduler traverses the prefill queue in arrival order and greedily adds as many requests as possible into the batch without exceeding the token budget M (line 3). Each prefill request r may contribute up to $\min(r.\text{TOKEN}, M)$ tokens (line 4), and the budget M is updated accordingly. The traversal stops once the budget is exhausted. This budget-aware selection maximizes utilization while preserving SLO. This procedure allows Libra to adaptively shape batches based on runtime conditions.

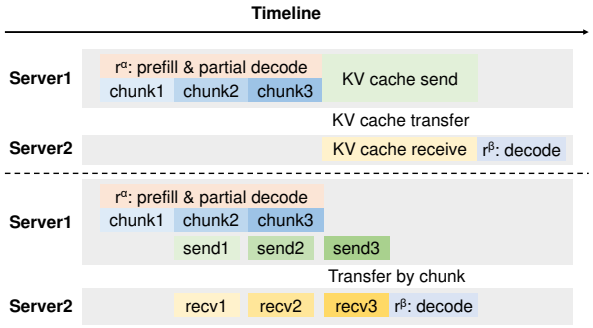

Figure 7: Transfer optimization: original (top) vs. Libra's chunk-based transfer(bottom).

4.3 Chunk-Based KV Transfer

Figure 7 illustrates how Libra transfers the KV cache of r^{α} across servers at chunk granularity. Server1 processes r^{α} in equal-sized chunks, regardless of token type. Once chunk k completes, its KV block is immediately DMA-pushed to Server2, while Server1 continues with chunk $k+1$. This is both safe and efficient because the KV cache is append-only [31]; completed chunks are immutable and can be transferred in parallel without coherence concerns. Transfers are fully offloaded to high-speed RDMA-based libraries like NCCL [22] or Mooncake [24], with lightweight ZeroMQ [44] messages steering placement on the receiver side. This chunk-level transfer overlaps communication with computation, hiding KV transfer costs. In typical NVLink or RDMA configurations, transfer latency is 1–2 orders of magnitude smaller than compute time. Compared to prior layer- or iteration-level approaches [25, 31], our method achieves finer granularity.

4.4 Extending FPS to Multi-Split Inference

Libra's FPS naturally generalizes beyond binary split to accommodate multi-stage inference patterns. For example, a Chain-of-Thought (CoT) request introduces an intermediate "think" stage that, being non-user-facing, has relaxed latency constraints. Libra leverages this by partitioning such a request into three distinct micro-requests: prefill, think, and decode. This allows the scheduler to handle the "think" stage with a different SLO, enabling more aggressive load balancing to improve throughput without compromising the final output latency. This multi-split partitioning is a general principle

that can be extended to more complex workloads. The same principle extends naturally to even more micro-requests when workloads introduce additional stages. We have implemented this multi-split capability, and as demonstrated in §6.7, it yields significant goodput improvements.

5 Implementation and Discussion

We implemented Libra atop vLLM (4K lines of Python), which can be adaptable to other inference engines.

Length prediction discussion. While Libra can leverage output length prediction [26, 29] to improve efficiency, its overall performance do not rely on accurate predictions. The global scheduler bases its decisions on relative execution times across GPU stages rather than exact token counts, which inherently limits the impact of prediction errors. To further guard against underestimation which could cause SLO violations, we apply a conservative safety margin (20 tokens in our setup), while slight overestimation has only negligible efficiency impact. Our sensitivity analysis (Table 5) confirms that Libra consistently maintains its performance targets even with noisy predictions, demonstrating that prediction accuracy is beneficial but never a point of failure.

Offline profiling procedure. To enable accurate SLO-aware latency estimation, we perform offline profiling across key request parameters: prefill length, number of decode requests, and context length. For each combination, we repeatedly measure execution latency on the target GPU to reduce variability, ensuring stable and reliable measurements. The results are stored in a multi-dimensional lookup table, which the scheduler consults at runtime to estimate expected latencies for different batch compositions. This procedure allows the scheduler to make informed batching decisions while keeping latency within SLO constraints. Our profiling approach is also robust against minor measurement noise.

6 Evaluation

6.1 Experimental Setup

Models. We evaluate Libra using three open-source LLM models from the popular Qwen-2.5 series [27], including Qwen-2.5 14B [6], 32B [7] and 72B [8] versions.

Testbed. Our primary evaluation platform consists of two cloud servers, each with four NVIDIA A100 80GB GPUs, 128 CPUs, 1TB RAM, and 4×200 Gbps ConnectX-6 RoCE NICs. NVLink provides 600 GB/s bandwidth between any two GPUs within a server. Additional experiments ran on H100 cloud to validate Libra on newer hardware.

Workloads. We generate request arrival patterns using a Poisson distribution, as in prior work [18, 38, 41]. Request input and output lengths are sampled from multiple real-world datasets, including Azure Code [4], BurstGPT [37], arXiv Summarization [9], and Mini Reasoning [17]. These diverse settings reflect a variety of prompt/output length distributions.

Baselines. We compare Libra against the state-of-the-art LLM serving system vLLM (v0.7.2) in two configurations. **PD Colocation (PD Coloc.)**: We use vLLM's default chunked prefill strategy, with optimal chunk sizes tuned between 256–2048 for each workload. Since Sarathi's [1] Chunked Prefill has already been adopted by vLLM, we do not evaluate Sarathi-Serve separately; **PD Disaggregation (PD Disagg.)**: We extend vLLM's original disaggregation mode (v0) to a more advanced version (v1) under modern scheduling logic. DistServe [41] shares a design very similar to vLLM's PD mode; hence, we omit an independent comparison. We focus on these baselines because they represent common design patterns, and reusing vLLM ensures fair comparisons.

We deploy models using data (DP) and tensor parallelism (TP). For PD Coloc., we use 2 GPUs (DP=2) for 14B, 4 GPUs (TP=2, DP=2) for 32B, and 8 GPUs (TP=4, DP=2) for 72B. For PD Disagg., we use TP within separate prefill and decode instances: we use 1P1D for 14B, 2P2D for 32B, and 4P4D for 72B. Libra uses the same number of GPUs as PD Disagg., allocating TP groups to process r^{α} and r^{β} separately: 1 GPU each for r^{α} and r^{β} in 14B (2 GPUs total), 2 GPUs each in 32B (4 GPUs total), and 4 GPUs each in 72B (8 GPUs total). For the main evaluations, Libra splits each request into two micro-requests, which is the standard setup for these workloads. The three-split generalization is evaluated separately in §6.7.

Metrics. We evaluate Libra using two primary metrics: Goodput [41], defined as the number of tokens generated per second while meeting service-level objectives (SLOs), and serving capacity, defined as the maximum sustainable queries-per-second (QPS) under SLO constraints, following the definition used in Sarathi Serve [1]. We enforce a 100 ms TBT SLO, widely adopted in academia [41, 43] and industry. Such SLO ensures smooth streaming in interactive applications. We also evaluate a stricter 50 ms SLO on H100 GPUs (§6.8).

6.2 Overall Results

Figure 8 compares the goodput of Libra, PD Coloc., and PD Disagg. across various workloads and model scales. Libra consistently outperforms both baselines across all evaluated settings. Specifically, it achieves a maximum goodput improvement of up to 91% over PD Coloc., and up to 61% over PD Disagg.. These gains reflect Libra's load balancing and reduced PD interference.

In BurstGPT, Azure Code, and arXiv Summarization workloads, Libra demonstrates a steady increase in goodput with rising QPS, until reaching saturation where it maintains a plateau. In contrast, PD Coloc. shows degraded performance beyond a certain QPS due to PD interference. This is mitigated in Libra by only introducing mixed PD batches when imbalance arises, thereby reducing the likelihood and severity of interference. Furthermore, the dynamic batch composition strategy helps constrain batch-level delays and maintain TBT within the SLO threshold. PD Disagg., while immune to direct PD interference, lacks scheduling flexibility due to its rigid prefill/decode separation. As a result, it often suffers from server under-utilization in skewed workload.

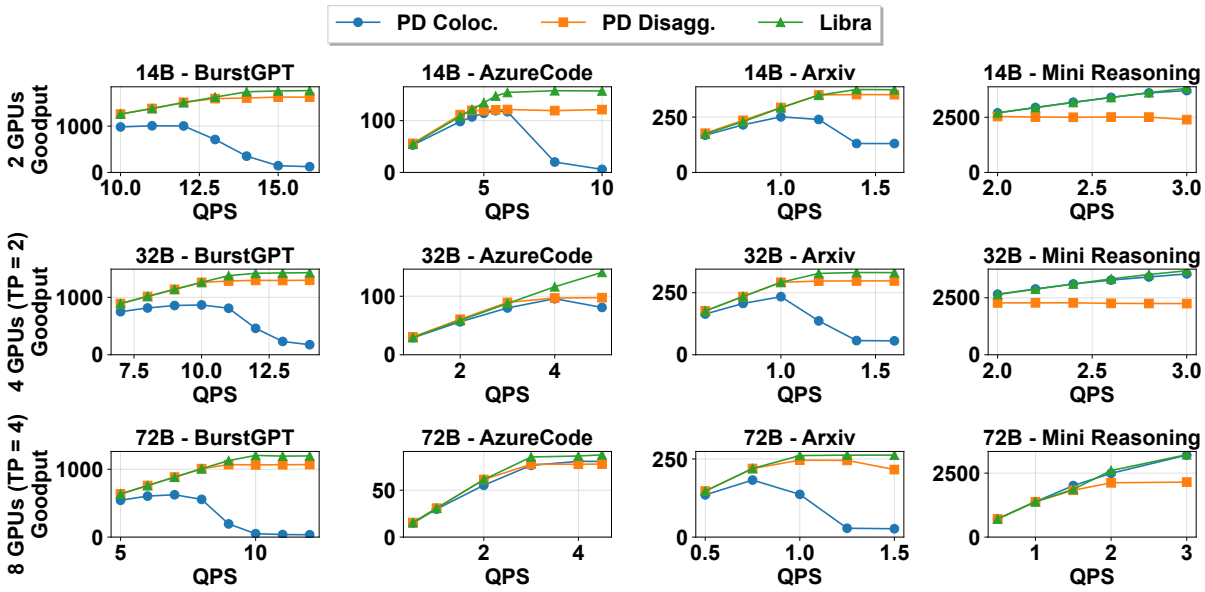

Figure 8: Goodput of Libra, PD Colocation, and PD Disaggregation under a 100 ms TBT SLO. Rows represent model sizes (14B, 32B, 72B); columns represent workloads (BurstGPT, AzureCode, arXiv Summarization, and Mini Reasoning).

Table 2: TTFT (ms) comparison between Libra and PD Disaggregation across different workloads.

TTFT (ms)	BurstGPT	Arxiv	Mini-Reasoning
PD Disagg.	23.8	83.9	0.058
Libra	12.6	68.5	0.088

In Mini Reasoning, with pronounced imbalance from longer generations, Libra's adaptive partitioning improves utilization and goodput. Interestingly, in this particular workload, PD Coloc. also achieves relatively strong performance. This is because the prefill stage accounts for a smaller fraction within the workload, making decoding less sensitive to prefill-induced delays and PD Coloc. becoming a very strong baseline. As model size grows from 14B to 72B, PD Coloc. suffers increasing PD interference, causing sharp goodput drops past peak QPS. In contrast, Libra maintains stable throughput and consistently outperforms PD Disagg., showing strong resilience to scaling and load. We also compare against asymmetric baselines in §6.6, where we show that Libra still outperforms all configurations.

TTFT analysis. Table 2 reports the TTFT for Libra and PD Disaggregation. On prefill-heavy workloads, Libra reduces TTFT by 47% on BurstGPT and 18% on Arxiv, as load balancing reduces queuing delays at prefill instances. On decode-heavy Mini-Reasoning, Libra's TTFT increases slightly (0.08 ms vs. 0.058 ms), but remains well below the 100ms decoding SLO, sufficient for production systems.

6.3 Serving Capacity

Beyond goodput, we also measure serving capacity (the maximum queries per second (QPS) each system can support while keeping the p99 token-by-token latency (TBT) under the 100 ms SLO) in Qwen-2.5-14B model. As shown in Figure 9, Libra consistently achieves the highest serving capacity, averaging 2.37× that of PD Coloc. and 1.37× that of PD Disagg. Its elastic scheduling balances compute and interference, enabling adaptability across diverse workloads.

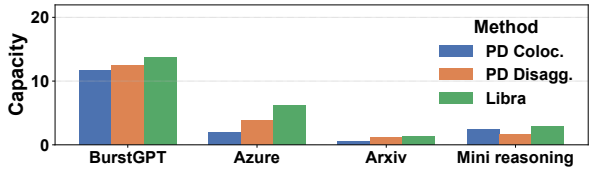

Figure 9: Serving capacity of Libra, PD Colocation, and PD Disaggregation across four workloads using the Qwen-14B.

Libra's capacity improvements align with its goodput gains, as it dynamically balances resource utilization and limits interference. For decode-light workloads (AzureCode), Disaggregation reduces prefill contention but underutilizes decode; Libra avoids both, reaching up to 3× Colocation's QPS. In decode-heavy cases (Mini Reasoning), Colocation benefits from batching, while Libra further improves capacity by tuning batch size and load distribution. Even on long-input (arXiv Summarization) and balanced (BurstGPT) workloads, Libra maintains an edge by flexibly coordinating prefill and decode without underutilization.

6.4 Hybrid Workload

To reflect real-world usage where request patterns span diverse tasks, we construct a hybrid workload by uniformly mixing BurstGPT and Azure Code requests. This combination introduces contrasting prompt and response characteristics,

Table 3: Serving capacity under a hybrid workload (50% BurstGPT + 50% Azure Code) using the Qwen-14B model.

System	PD Coloc.	PD Disagg.	Libra
Serving Capacity (rps)	4.6	5.9	7.4
Goodput (token/s)	316.32	399.31	473.84

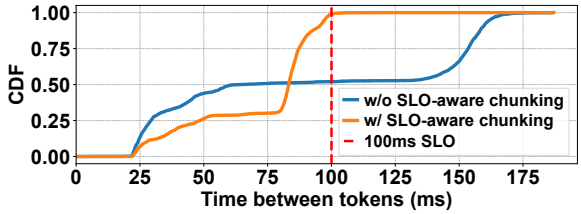

Figure 10: CDF of time-between-tokens (TBT) with and without SLO-aware batching, using Qwen-14B model and Azure-Code workload. QPS is set to serving capacity of Libra.

Table 4: Per-request scheduling overhead under varying QPS when serving Qwen-14B with BurstGPT traces.

QPS	6	8	10	12	14	16
Overhead	0.51%	0.40%	0.29%	0.16%	0.12%	0.09%

making static partitioning inherently unbalanced. As shown in Table 3, Libra achieves 60% higher serving capacity than PD Coloc. and 25% higher than PD Disagg.. In terms of goodput, Libra outperforms PD Coloc. by 49% and PD Disagg. by 20%. PD Coloc. suffers interference from large Azure prompts; PD Disagg. avoids interference but underutilizes resources due to fixed partitioning. In contrast, Libra dynamically adjusts the split ratio per request, balancing compute and memory usage across GPUs. This adaptivity leads to better utilization, stable latency, and higher capacity for Libra.

6.5 Breakdown Analysis

Then we dive into Libra to investigate its internal details.

Benefit of SLO-aware batching. We first evaluate the impact of SLO-aware batching by measuring the CDF of TBT under high-workload pressure conditions. As shown in Figure 10, without SLO-aware batching, PD interference leads to significant tail latency, with TBT exceeding 175 ms in the worst case. Only 52% of tokens are generated within the 100 ms SLO threshold, meaning nearly half the tokens violate the service requirement. By enabling SLO-aware batching, Libra effectively constrains PD interference and tail delay, raising the SLO-attainment to 99%. This demonstrates the importance of fine-grained batch adjustment in ensuring latency predictability under mixed prefill-decode batches.

Scheduling overhead analysis. We evaluate the runtime overhead of Libra 's global scheduler using the Qwen-14B model on a 2-GPU setup (1 for r^{α} and 1 for r^{β}) with BurstGPT traces at varying QPS. Table 4 reports the overhead as a fraction of requests' end-to-end latency. Across all workloads, the per-request overhead remains below 0.51%, corresponding to under 18 ms per request. Although the absolute overhead is roughly constant, its fraction of total end-to-end latency decreases at higher QPS, reflecting the increasing workload and demonstrating the efficiency of the lightweight scheduler.

Table 5: Sensitivity of goodput to prediction errors. The scheduler assumes an output length of 1467 tokens, while actual output lengths are sampled from a normal distribution with varying standard deviation. Prompt length is 219 tokens.

Standard deviation (σ)	0	10	50	100
Goodput (token/s)	3606.93	3591.00	3564.90	3501.85

Table 6: Comparison under different TP configurations for Prefill and Decode instances.

Parallelism	4P1D	1P4D	2P1D	1P2D	1P1D	Libra
Per GPU Goodput	68.8	33.1	73.2	49.8	65.1	86.8

Sensitivity analysis. Then, we evaluate Libra's robustness to output length prediction errors. We conduct a sensitivity analysis where the global scheduler assumes a fixed output length of 1467 tokens. Actual output lengths are drawn from a normal distribution with the same mean (1467) and varying standard deviations ($\sigma = 0, 10, 50, 100$). The prompt length is fixed at 219. In Table 5, Libra maintains high goodput across all levels of variance, with only a 2.9% drop at $\sigma = 100$, showing Libra's tolerance to moderate prediction errors.

Chunk-based KV transfer. Finally, we test chunk-based KV transfer using the Mini-Reasoning task as an example. Compared to a version of Libra without this technique, the chunked approach reduces non-overlapped transfer by 94%.

6.6 Asymmetric Tensor Parallelism

For a more rigorous comparison, we benchmark Libra against TP-asymmetrical baselines, specifically PD Disagg. (a stronger form of disaggregated setup), using a mixed workload scenario with 50% Azure workload (prefill-heavy) and 50% Mini-Reasoning (decode-heavy). Table 6 illustrates that Libra easily achieves higher per-GPU goodput in a basic 1P1D setup compared to PD Disagg. across all tested TP configurations, with improvements spanning 18.6% to 74.2%. This significant uplift highlights Libra's superior ability to squeeze maximum utilization from GPU resources with minimal configuration burden. The underlying limitation of PD Disagg. lies in its static TP assignment: a configuration such as 4P1D optimizes GPU utilization for the prefill-heavy workload but is ineffective for decode-heavy tasks, resulting in underutilized GPUs in a mixed environment. Libra, however, overcomes this limitation via its flexible request splitting mechanism, enabling it to dynamically balance and serve diverse workloads efficiently, thereby maximizing per-GPU goodput to a significant degree.

6.7 Multi Splits For CoT Reasoning

We evaluate Libra's multi-stage partitioning on the Mini-Reasoning (CoT) workload, where intermediate "think" to-

Table 7: Three Split Micro-requests Performance for COT.

	PD Disagg.	Libra (2-split)	Libra (3-split)
Goodput (token/s)	260.01	336.16	366.43

Table 8: H100 results: Goodput comparison against (a) data-parallel baselines and (b) under 50 ms low SLO.

Goodput	Azure Workload			Mini Reasoning		
	Coloc.	Disagg.	Libra	Coloc.	Disagg.	Libra
DP	592.5	587.5	**712.4**	6967.3	6014.2	**7090.0**
50-SLO	259.7	397.2	**608.0**	5696.2	3816.8	**5930.5**

kens have relaxed latency constraints. As shown in Table 6, Libra's default 2-split mode improves goodput by 29% over PD Disagg. but is limited by applying a single strict SLO to all output tokens. By adopting a 3-split strategy, Libra isolates the non-critical "think" tokens, enabling more aggressive load balancing without violating the final output's SLO. While CoT requires predicting both think and visible tokens, Libra only relies on coarse-grained load estimates for scheduling; execution still follows the precise token-level CoT labels. This stage-aware scheduling boosts goodput by another 12% to 366.43 token/s, achieving a 41% total gain over the baseline and demonstrating the power of flexible partitioning for complex inference patterns.

6.8 H100's Results

Due to limited A100 availability, the DP and low-SLO latency experiments ran on H100s, providing validation of Libra's effectiveness on newer hardware.

Data parallelism. As shown in the first data row of Table 8, Libra's dynamic scheduling proves superior to even tuned, asymmetric data-parallel baselines. Against a 2P1D-DP baseline on the prefill-heavy Azure Code workload, Libra improves goodput by 22.8% by using its unified scheduler to prevent the decode replica from being underutilized. On the decode-heavy Mini Reasoning workload, a 1P2D-DP setup suffers from decode GPUs hitting memory pressure while the prefill replica sits idle; Libra rebalances work across all GPUs and improves goodput by 18.0%. These results highlight Libra eliminates both compute underutilization and memory imbalance inherent in static partitioning.

Strict SLO experiment. Libra's performance advantages persist even under a strict 50ms SLO constraint, as shown in last row of Table 8. On the Azure workload, Libra improves goodput by 53.1% over PD Disagg and 134% over PD Coloc. On Mini Reasoning, it delivers 55.4% and 4% gains, showing that even when static baselines already operate near capacity, Libra still extracts further efficiency. These results demonstrate that Libra's dynamic resource allocation is critical for maximizing goodput while adhering to tight latency deadlines, showcasing its robustness in production-like scenarios where static partitioning can struggle.

7 Other Related Work

Request-level scheduling. Latency–throughput tradeoffs are central to LLM serving. Orca [40] improves throughput via continuous batching, while Apparate [11] reduces latency using dynamic early-exit points. Nanoflow [42] overlaps communication and computation within nano-batches for better intra-device parallelism. NIYAMA [14] adapts chunk sizes to support multi-SLO and priority workloads. These techniques offer advanced latency control for diverse scenarios. Libra can incorporate these techniques into its scheduling.

Phase-level scheduling. Apart from DistServe [41], Splitwise [23] also separates prefill and decode across clusters to reduce interference. And Sarathi-Serve [1] introduces chunked prefill with stall-free batching, but uses static chunk sizes and only partitions prompts, limiting its adaptability to dynamic workloads. WindServe [13] uses CUDA-stream-based disaggregation for intra-GPU PD sharing with dynamic prefill dispatch, but operates within the traditional PD architecture and cannot partition requests at arbitrary token boundaries. TaiChi [35] assigns requests to differentiated P-heavy/D-heavy instances for latency shifting, but limits to prompt-boundary split with offline-configured instance ratios and fails to achieve globally balanced load. Other approaches include Shubha [30], which extends phase-level disaggregation with heterogeneous GPU support; MLC-LLM [21], which statically offloads prefill to decode instances using a fixed ratio; Dynamo [3], which routes requests first and may perform prefilling directly on decode instances; Semi-PD [15], which uses SM-level control for intra-GPU sharing; and Adrenaline [19], which offloads decode attention kernels to prefill instances. In contrast, Libra departs from instance-level specialization and prompt-boundary partitions. It operates on unified instances and dynamically partitions each request at *any token position* with global load balancing, enabling robust adaptation to dynamic workloads.

8 Conclusion

We introduced Libra, a scalable LLM serving system that moves beyond static disaggregation and colocation through its micro-request abstraction and two-level scheduler. By dynamically balancing load across a unified GPU pool, Libra combines the high utilization of colocation with the low tail latency of disaggregation. Our results demonstrate superior goodput and capacity, proving that dynamic, request-level scheduling is more efficient for large-scale LLM serving.

Acknowledgments

We thank our shepherd Qizhe Cai and anonymous reviewers for their valuable comments. This work was supported in part by National Key R&D Program of China under Grant No. 2024YFB4505701, the National Natural Science Foundation of China under Grant No.: U25B2020, and the Singapore Ministry of Education, under Academic Research Fund Tier 3 grant MOE-MOET32024-0003.

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] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[3] Ayush Agrawal and contributors. Dynamo: A datacenter scale distributed inference serving framework. https://github.com/ai-dynamo/dynamo, 2025. Accessed September 2025.

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

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

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

[7] Alibaba Cloud. Qwen/qwen2.5-32b-instruct. https://huggingface.co/Qwen/Qwen2.5-32B-Instruct, 2024. "[accessed-April-2025]".

[8] Alibaba Cloud. Qwen/qwen2.5-72b. https://huggingface.co/Qwen/Qwen2.5-72B, 2024. "[accessed-April-2025]".

[9] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Thien Huu Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. A discourse-aware attention model for abstractive summarization of long documents. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 615–621, 2018.

[10] Christopher Culver, Peter Hicks, Mihailo Milenkovic, Sanjif Shanmugavelu, and Tobias Becker. Scientific computing with large language models, 2024.

[11] Yinwei Dai, Rui Pan, Anand Iyer, Kai Li, and Ravi Netravali. Apparate: Rethinking early exits to tame latency-throughput tensions in ml serving. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 607–623, 2024.

[12] Viet-Tung Do, Van-Khanh Hoang, Duy-Hung Nguyen, Shahab Sabahi, Jeff Yang, Hajime Hotta, Minh-Tien Nguyen, and Hung Le. Automatic prompt selection for large language models, 2024.

[13] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. Windserve: Efficient phase-disaggregated llm serving with stream-based dynamic scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1283–1295, New York, NY, USA, 2025. Association for Computing Machinery.

[14] Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. Niyama : Breaking the silos of llm inference serving, 2025.

[15] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, Yun Liang, and Yu Wang. semi-pd: Towards efficient llm serving via phase-wise disaggregated computation and unified storage, 2025.

[16] 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.

[17] KingNish. mini_reasoning_1k: A small-scale dataset for evaluating llm reasoning ability. https://huggingface.co/datasets/KingNish/mini_reasoning_1k, 2024. Accessed: 2025-05-09.

[18] 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.

[19] 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, 2025.

[20] 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.

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

[22] NVIDIA. Nccl - nvidia collective communications library. https://github.com/NVIDIA/nccl, 2024. Accessed: 2025-05-14.

[23] 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.

[24] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 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)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[25] 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.

[26] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Efficient interactive llm serving with proxy model-based sequence length prediction, 2024.

[27] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[28] sglang. SGLang Project, 2024-2025.

[29] Rana Shahout, eran malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. DON't STOP ME NOW: EMBEDDING BASED SCHEDULING FOR LLMS. In *The Thirteenth International Conference on Learning Representations*, 2025.

[30] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. {USHER}: Holistic interference avoidance for resource optimized {ML} inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 947–964, 2024.

[31] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, 2024.

[32] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications, 2022.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] vLLM. vLLM: Easy, fast, and cheap LLM serving for everyone, 2023-2025.

[35] Chao Wang, Pengfei Zuo, Zhangyu Chen, Yunkai Liang, Zhou Yu, and Ming-Chang Yang. Prefill-decode aggregation or disaggregation? unifying both for goodput-optimized llm serving. *arXiv preprint arXiv:2508.01989*, 2025.

[36] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024.

[37] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024.

[38] 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.

[39] Jochen Wulf and Juerg Meierhofer. Exploring the potential of large language models for automation in technical customer service, 2024.

[40] 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.

[41] 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.

[42] 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.

[43] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 592–608, New York, NY, USA, 2025. Association for Computing Machinery.

[44] ZMQ. An open-source universal messaging library. https://zeromq.org/, 2025. Accessed: 2025-05-16.

9 Appendix

9.1 Real-time Workload Replay

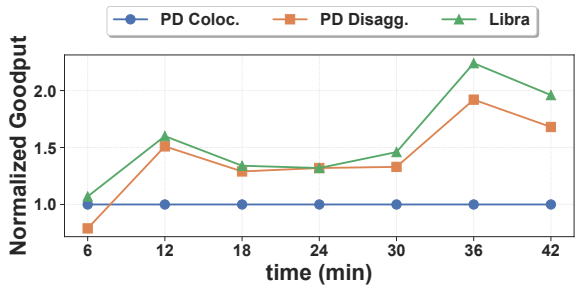

Figure 11: Goodput trends under a real-time workload from the BurstGPT dataset, using the Qwen-14B model.

To evaluate performance under realistic temporal dynamics, we extract a continuous request stream from the BurstGPT dataset, preserving original arrival pattern to simulate a realtime workload. The extracted trace starts at 311th hour and spans 42 minutes, and we measure average goodput every 6 minutes. Figure 11 shows goodput over time for all systems. We observe fluctuations in the measured goodput, which are primarily attributed to variations in input lengths. During intervals with longer prompts, the system typically exhibits reduced goodput, resulting from increased prefill overhead. During the time interval between 12 and 42 minutes, PD Disaggregation outperforms PD Colocation, thanks to its isolation of prefill and decode stages which mitigates interference. Libra, however, further improves goodput by employing SLO-aware batching to limit interference and fine-grained partition to dynamically rebalance load between GPU instances, achieving consistently higher throughput.

In contrast, during the first 6 minutes, requests are more decode-heavy with relatively short prefill phases. Here, Colocation suffers less from interference and temporarily surpasses Disaggregation. Even in these cases, Libra leverages its flexible fine-grained partition mechanism to maintain top-tier goodput across the board.

9.2 Implementation Details

We implemented Libra on top of vLLM with 4K lines of Python code, as an end-to-end distributed LLM serving architecture featuring a two-level scheduler and backend integration interface. The design is non-intrusive and can be extended to various other inference engines. We will open source our code in the near future.

Request scheduler. The global scheduler in Libra is integrated into the front-end proxy of `vLLM`. On arrival, each request is passed through a length predictor, then routed to the scheduler, which chooses a partition ratio ϕ and dispatches the resulting micro-requests to a pair of GPU engines. A lightweight latency predictor (§4.1) embedded in the controller estimates per-engine throughput in real time. To reduce scheduling overhead, we implemented the global scheduler logic in C++ while maintaining full compatibility with the Python-based codebase.