LAPS: A LENGTH-AWARE-PREFILL LLM SERVING SYSTEM

Jianshu She [1] **Zonghang Li** [1] **Hongchao Du** [1] **Shangyu Wu** [1] **Wenhao Zheng** [2] **Eric Xing** [1] **Zhengzhong Liu** [1]
Huaxiu Yao [2] **Jason Xue** [1] **Qirong Ho** [1]

ABSTRACT

Length-**A**ware **P**refill **S**erving (LAPS) identifies and disaggregates requests with different prompt lengths in LLM serving to reduce TTFT latency. While recent systems have decoupled the prefill and decode stages to improve throughput, they still rely on unified scheduling policies that fail to adapt to heterogeneous workload characteristics. We observe that prompt-length variations lead to distinct performance bottlenecks, motivating an adaptive scheduling strategy. LAPS disaggregates multi-turn long-prefill requests from short-prefill ones and introduces a length-aware smart batching mechanism for short-prefill workloads. It adopts a dual-queue design that supports temporal disaggregation on a single prefill instance or spatial disaggregation across multiple instances. For short-prefill batches, a batch waiting window and CUDA Graph-based clustering mitigate interference from heterogeneous computation, reducing batching delay and lowering average latency. In real multi-turn workloads, LAPS reduces prefill latency by over 30% compared to vanilla SGLang under prefill–decode disaggregation, and further decreases SLO violations by 28% in multi-instance deployments with vanilla data-parallel configuration. Compared to the SGLang router with load balancing, it further lowers SLO violations by 12% in multi-GPU settings. Under high concurrency and mixed-request scenarios, LAPS improves request throughput by 35% serving Qwen2.5-32B model for prefill instance, demonstrating its effectiveness in optimizing heterogeneous LLM serving workloads.

1 INTRODUCTION

Modern LLM serving stacks (e.g., vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024b)) combine prefill-decoding (PD) disaggregation (Zhong et al., 2024) with continuous batching to meet low-latency, high-concurrency service-level objectives (SLOs). The Prefill stage (first-token computation) is largely compute-bound, while the decoding stage (auto-regressive generation) is memory-bound. PD disaggregation decouples these two phases across separate instances to avoid cross-phase contention. On the prefill side, concurrent requests are batched to raise GPU utilization. However, we show that this separation and batching are insufficient: even with PD, interference still persists within the prefill stage when long, compute-bound prefills are mixed with short, memory-bound prefills/re-prefills.

Re-prefill denotes the repeated prefill in multi-turn sessions where the model extends an existing context by combining new tokens with cached KV states. It is common in chatbots (Dam et al., 2024), tool-using agents (Wölflein et al., 2025), RAG, speculative decoding (Leviathan et al., 2023), and token routing (She et al., 2025), and is typically memory-bound (dominated by KV-cache reads/writes rather than large GEMMs). Figure 2 illustrates the token length distribution of a real-world trace, LMsys-Chat-1M (Zheng et al., 2024a), which is collected from real human-AI conversations. We observe that most prompts are short (<256 tokens), while long-context requests (>1K tokens) are relatively rare. This indicates that production workloads are dominated by short prefills/re-prefills, which are memory-bound. Mixing them with long, compute-bound prefills in unified batches will also induce compute-memory interference: short prefills/re-prefills wait behind long GEMMs and time-to-first-token (TTFT) spikes, and long prefills lose effective FLOPs due to heavy KV traffic from short jobs. Figure 1 confirms this issue: mixing long and short prefills does significantly increase long-prefill latency, and this contention becomes more severe as concurrency rises.

Although prior systems have recognized the resource contention between compute-bound and memory-bound workloads, they focus on coordinating prefill and decode: (1) decode-priority schedulers(Kwon et al., 2023) prioritize the decode phase to minimize per-token latency; (2) prefill-first schedulers prioritize the prefill phase and use continuous batching (Yu et al., 2022) to improve throughput; (3) stall-free chunked prefill (Agrawal et al., 2023) splits long

[*]Equal contribution [1]Mohamed bin Zayed University of Artificial Intelligence [2]The University of North Carolina at Chapel Hill. Correspondence to: Jianshu She <jianshu.she@mbzuai.ac.ae>, Qirong Ho <Qirong.Ho@mbzuai.ac.ae>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

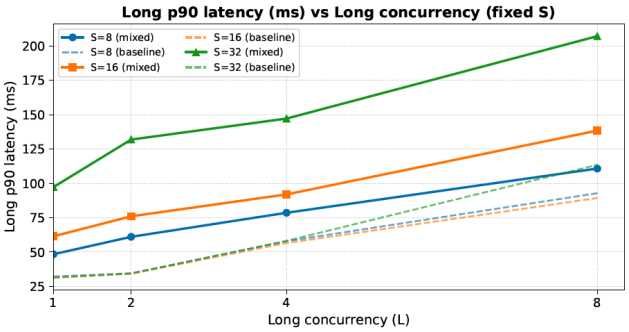

Figure 1. P90 TTFT of long-prefill requests under varying concurrency levels for long requests under fixed short requests. The long-prefill requests have more than 1K tokens, while the short ones have fewer than 64 tokens. We concurrently run them on a single H200 GPU and serve by Qwen2.5-32B (Qwen et al., 2025).The dashed lines (baseline) indicate latency for long-prefill-only workloads, while the solid lines (mixed) correspond to batches with both short and long requests.

prefill into chunks and interleaves them with decode, so long prefills don't stall ongoing decodes; and (4) PD disaggregation (Zhong et al., 2024; Jin et al., 2024; Hu et al., 2024a; Strati et al., 2024), which dominates in modern serving systems, runs prefill and decode on separate instances. These advances alleviate cross-phase contention but implicitly assume all prefills are compute-bound long sequences, overlooking that short (e.g., $<$256 tokens), memory-bound prefills/re-prefills could dominate real-world multi-turn serving workloads.

To address these challenges, we propose LAPS, a length-aware LLM serving system that explicitly disaggregates and optimizes long-prefill and short-prefill workloads within the prefill stage. LAPS maintains two separate prefill pools at runtime and performs batch disaggregation to isolate long and short prefill requests, completely eliminating their mutual interference. For short-prefill requests, LAPS introduces a dynamic waiting window in the scheduler and bucketizes requests by input length, making them more uniform and enabling larger batch sizes. This reduces batch-launch overhead and, combined with CUDA Graph-based execution, further accelerates processing and improves throughput. For SLO-serving scenarios, we design an SLO-aware scheduler that balances the trade-off between the waiting window and throughput.

In multi-instance deployments, the scheduler dynamically adjusts each instance's workload type based on real-time load, achieving load balancing across spatially disaggregated prefill instances. This adaptive strategy resembles resource allocation problems in deep learning systems, like in (Qiao et al., 2021).

In addition, LLM serving systems typically support three modes: mix and PD temporal/spatial disaggregation. Beyond these, our LAPS introduces a fourth mode:

- *Mix:* Decode requests are inserted into prefill batches (without disaggregation);
- *PD temporal disaggregation:* Prefill and decode batches run sequentially on the same instance;
- *PD spatial disaggregation:* Prefill and decode batches run on separate instances.
- *Prefill batch temporal/spatial disaggregation (ours):* Enabling disaggregation within the prefill stage (rather than between prefill and decode), separating long- and short-prefill batches temporally on a single instance or spatially across instances.

It is worth noting that our LAPS remains compatible with existing PD disaggregation architectures. Overall, our main contributions are summarized as follows:

1. **Empirical characterization.** We analyze intra-prefill interference between long and short requests in multi-turn workloads, exposing compute-memory contention caused by current batching strategies.
2. **Length-aware disaggregation.** We design a request-level temporal/spatial prefill disaggregation architecture that isolates long and short prefills to eliminate interference.
3. **Adaptive scheduling.** We introduce a dynamic bucket-based batching policy with a waiting window and load balancing across prefill instances, improving throughput and reducing SLO violations.

2 Background and Motivations

In this section, we model the token-length conditions under which prefill and re-prefill become memory-bound, how compute-bound long prefills and memory-bound short (re-)prefills interfere with each other, and what causes this long/short mixing.

2.1 Compute-Memory Boundary for (Re-)Prefills

The prefill and re-prefill phases have different latency behaviors. In re-prefill, the model processes new prompt tokens while also attending to historical tokens. This increases both computing and memory overhead, and shifts the token-length boundary (critical point) L_m at which (re-)prefills transition from compute-bound to memory-bound. We now formulate a unified latency model to find the token-length boundaries L_m^{prefill} and $L_m^{\text{re-prefill}}$ for prefill and re-prefill phases, respectively. We will use this model to show

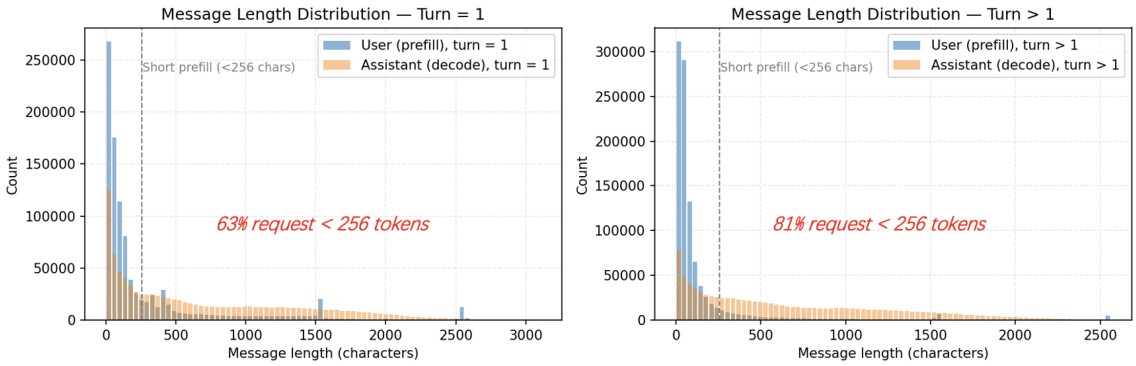

Figure 2. The token length distribution of multi-turn dialogues in the real LMsys-Chat-1M dataset. The left plot shows the prompt length in the first turn (including the system prompt by default), where approximately 63% of requests contain fewer than 256 tokens. In subsequent turns, the proportion of prompts shorter than 256 tokens increases to an average of 81%.

that **both prefill and re-prefill remain memory-bound for shorter fill lengths**.

Let L be the number of new tokens in this turn, H be the number of historical tokens, and $T(L,H) = T_{\text{comp}}(L,H) + T_{\text{mem}}(L,H)$ be the total latency. The compute term reflects incremental causal attention and FFN:

$$T_{\text{comp}}(L,H) \approx \alpha L(L+2H) + \beta L,$$

where α, β are per-token costs for attention and FFN compute, respectively. The memory term models the time for KV read/write I/O:

$$T_{\text{mem}}(L,H) \approx \gamma_w L + \gamma_r H,$$

where γ_w and γ_r are per-token KV write/read times.

Prefills In the first-turn prefill, there is no history ($H = 0$), so $T_{\text{comp}}(L,0) \approx \alpha L^2 + \beta L$ and $T_{\text{mem}}(L,0) \approx \gamma_w L$. The boundary can be obtained by equating these two contributions, yielding:

$$L_m^{\text{prefill}} = \max\left(0, \frac{\gamma_w - \beta}{\alpha}\right).$$

If $\gamma_w \leq \beta$, prefills are always compute-bound; otherwise, memory access dominates for small $L < L_m^{\text{prefill}}$.

Re-prefills. Similarly, for re-prefills with $H > 0$,

$$T_{\text{comp}}(L,H) \approx \alpha L^2 + (2\alpha H + \beta)L,$$

$$T_{\text{mem}}(L,H) \approx \gamma_w L + \gamma_r H,$$

so the boundary is given by:

$$L_m^{\text{re-prefill}} = \max\left(0, \frac{-(2\alpha H+\beta-\gamma_w)+\sqrt{(2\alpha H+\beta-\gamma_w)^2+4\alpha\gamma_r H}}{2\alpha}\right).$$

For any fixed $H > 0$, re-prefill is memory-bound for small $L < L_m^{\text{re-prefill}}$ because when $L \to 0$, $T_{\text{mem}}(L,H) \to \gamma_r H > 0$ while $T_{\text{comp}}(L,H) \to 0$. As H increases, the $L_m^{\text{re-prefill}}$ boundary grows until a saturation point: $L_m^{\text{re-prefill}} \to \frac{\gamma_r}{2\alpha}$ for large $H \gg |\beta - \gamma_w|/(2\alpha)$. Thus, with long histories, re-prefills remain memory-bound up to a constant number of new tokens, after which the $2\alpha HL$ and αL^2 terms render the phase compute-bound.

Fitting at runtime Compute and memory latency can be modeled as quadratic and linear functions of (L,H), respectively. We collect runtime samples $(T_{\text{comp}}, T_{\text{mem}}, L, H)$ to fit these two curves and obtain $\alpha, \beta, \gamma_w, \gamma_r$, and then calculate the boundaries L_m^{prefill} and $L_m^{\text{re-prefill}}$.

Roofline model We also use the arithmetic intensity and roofline model to characterize the transition between memory- and compute-bound workloads in the prefill stage. The arithmetic intensity of prefill computation increases approximately linearly with the prompt length L, since longer sequences proportionally increase the ratio of arithmetic operations to memory access. The compute-memory boundary occurs when the arithmetic intensity $AI(L)$ reaches the hardware roofline slope $AI^* = P_{\text{peak}}/B_{\text{mem}}$, where P_{peak} and B_{mem} denote the peak compute throughput and sustained memory bandwidth of the GPU.

Empirical profiling across advanced hardware generations (A100, H100, and H200) and LLMs ranging from 7B to 32B parameters shows that this transition typically occurs between 150 and 512 tokens (Yuan et al., 2024; Zhong et al., 2024), depending on model architecture, kernel implementation, and batch configuration. Prefills shorter than this range tend to be memory-bound and limited by KV-cache I/O, while longer ones are dominated by GEMM throughput.

Case study We use empirical measurements of Qwen2.5-7B on a single H200 GPU as a case study. Figure 3 and Figure 4 illustrates the positions of FlashAttention and GEMM

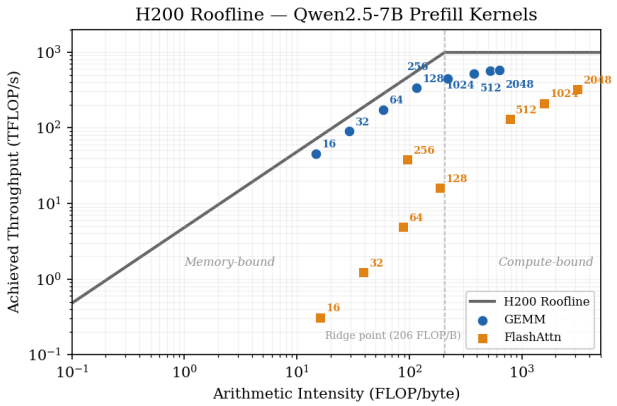

Figure 3. Roofline analysis of GEMM and FlashAttention kernels for single-request prefill on H200.

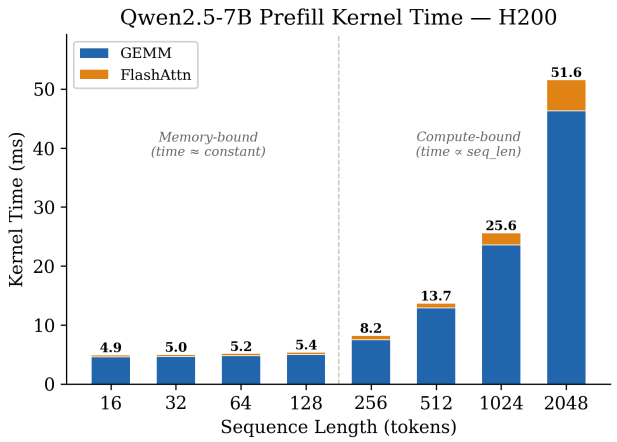

Figure 4. Absolute kernel time for a single prefill forward pass.

kernels on the H200 roofline model, together with their execution time contributions.

We profile Qwen2.5-7B on a single H200 GPU (in Figure 3, BF16 peak: 989 TFLOPS, HBM3e bandwidth: 4.8 TB/s). Each point represents the aggregated kernel time at a given sequence length (16–2048 tokens).

GEMM kernels (blue circles) transition from memory-bound to compute-bound as sequence length crosses the ridge point (206 FLOP/byte), corresponding to 256 tokens. At short sequences ($\leq$128), GEMM arithmetic intensity is low (14–115 FLOP/byte) and performance is limited by HBM bandwidth. At longer sequences ($\geq$256), arithmetic intensity exceeds the ridge point, and achieved throughput saturates at 520–577 TFLOPS (55% of peak).

FlashAttention kernels (orange squares) are strongly memory-bound at short sequences due to limited $O(S^2)$ compute, but rapidly shift toward the compute-bound regime at longer sequences (reaching up to 3122 FLOP/byte at $S = 2048$).

In Figure 4, Kernel time exhibits two regimes: it remains nearly constant (5 ms) for short sequences ($\leq$128) due to memory-bound weight loading, but grows linearly beyond 256 tokens as execution becomes compute-bound, explaining why mixing short and long prefills degrades short-request TTFT.

2.2 Exploring Interference Between Long-Short Prefills/Re-prefills

We model the interference between long-prefill and short-prefill (or re-prefill) requests using a standard M/G/1 (Markovian/General/1-server queue) first-come, first-served (FCFS) queuing model (Meini, 1998). Unlike prior PD scheduling analyses that focused on prefill-decode contention, we examine the *intra-prefill* interference caused by mixing compute-bound and memory-bound jobs within continuous batching.

In this model, each request passes through two service stations: a compute station (for GEMM-dominated operations) and a memory station (for KV-cache I/O). Short-prefill and re-prefill jobs are typically memory-bound, while long-prefill jobs are compute-bound. Let the aggregate arrival rate be λ and utilization $\rho < 1$. Denote by p the fraction of short jobs in the workload. The service time at the memory station is $S_m = \gamma_w L + \gamma_r H$, and at the compute station $S_c = \alpha L^2 + (2\alpha H + \beta)L$.

Using the Pollaczek-Khinchine (P-K) result (Neuts, 1986) for M/G/1 queues, the mean waiting time is

$$W = \frac{\lambda\, \mathbb{E}[S^2]}{2(1-\rho)}.$$

When jobs of different lengths are batched together, the variance in service times inflates waiting time for all requests, introducing a **head-of-line (HoL) blocking penalty**:

$$\Delta W_{\text{HoL}} = \frac{\lambda p(1-p)}{2(1-\rho)}(S_\ell - S_s)^2 > 0.$$

This term grows with higher concurrency and service heterogeneity, explaining the observed latency increase in mixed long/short-prefill workloads (shown in Figures 1)

Furthermore, long prefills hurt short (re-)prefills more. Every class sees the same queuing term W, so normalized latency is $R_i/S_i = 1 + W/S_i$. Given $S_s < S_\ell$, the relative increase is larger for short jobs because $W/S_s > W/S_\ell$. This convoy effect explains why short-prefill latency grows faster as long-prefill concurrency increases, which is a clear symptom of **bandwidth contention**.

Case Study: Kernel-Level Interference in Mixed Batches. To quantify the compute waste induced by mixing heteroge-

Category	Prefill / Decode Features	Typical Tasks
SPSD	Short prompt and short output	Token-level routing, classification, short chat replies, tool selection
SPLD	Short prompt, long output	Code generation, creative writing, long summarization, Chain-of-Thought
LPSD	Long context, short output	Document QA, retrieval-based answering, chat reply, sentiment classification
LPLD	Long context and long output	Long-form summarization, report writing, autonomous agents

Table 1. Task classification by prefill and decode characteristics. SPSD: short-prefill, short-decode; SPLD: short-prefill, long-decode; LPSD: long-prefill, short-decode; LPLD: long-prefill, long-decode.

neous sequence lengths, we profile Qwen2.5-7B and 32B on a single H200 GPU using HuggingFace padded batching.

We consider three configurations: (A) *short-only*, 8×64 tokens; (B) *long-only*, 1×2048 tokens; and (C) *mixed*, $7\times64 + 1\times2048$, resulting in a padded shape of $(8, 2048)$ with only 2,496 useful tokens out of 16,384. We measure forward-pass latency (100 iterations, `cudaEvent`) and decompose GPU time using the PyTorch CUDA profiler.

Table 2 shows that mixing induces substantial inefficiency: 85% of GEMM and 87% of attention computation are wasted on padding, inflating short-request latency by 29–31$\times$ compared to the short-only batch. Table 3 further reveals that this slowdown is dominated by GEMM (36–39$\times$) but propagates across all kernel categories.

These results demonstrate that length heterogeneity introduces structural compute waste, which directly degrades short-request TTFT.

2.3 Uncovering the Sources of Long-Short Prefill Mixing

General-purpose LLM services must handle a wide spectrum of tasks. As shown in Table 1, daily chat and creative ideation are typical short-prompt tasks, while speculative decoding and token routing produce high-frequency, short re-prefills (Chatterji et al., 2025). In contrast, long-document QA and autonomous agent workflows correspond to long-context prefills. In practice, these streams interleave over time, leading to long-short mixing within prefill workloads.

Most existing systems schedule requests in an FCFS fashion, packing them into unified batches. Many deploy multi-queue variants: continuous or rolling batching (e.g., vLLM (Kwon et al., 2023), TGI (HuggingFace, 2024)) treats prefill and decode as distinct phases, applying FCFS-style admission under token/KV limits and optional priorities or aging policies. With chunked prefill (e.g., Sarathi-Serve (Agrawal et al., 2023)), vLLM prioritizes decode and may co-batch prefills with decode. In-flight batching (e.g., TensorRT-LLM (Corporation, 2023)) runs distinct context and generation engines, each maintaining its own ready queue and often prioritizing generation. Moreover, many LLM gateways expose service tiers that prioritize certain request classes while enforcing token-based budgets using fixed or sliding windows (e.g., OpenAI (OpenAI, 2024), Anthropic (Anthropic, 2024), Envoy (Proxy, 2024), Kong (Inc., 2024b), APISIX (Foundation, 2024), and Cloudflare (Inc., 2024a)).

These systems indeed adopt multi-queue designs, but most queues are phase-oriented (prefill vs. decode) or SLA-based. Consequently, long and short (re-)prefills still end up co-batched even under multi-queue scheduling. Long-prefill requests have longer residence times, and schedulers backfill every few milliseconds, so newly arriving short (re-)prefills are co-admitted into the same batch. Larger admission windows or micro-batches further raise the odds of co-admission, while speculative decoding and token routing inject frequent short jobs alongside ongoing long prefills.

The most related line of work is *length bucketing*, which groups requests by predicted sequence length into size-homogeneous buckets to reduce padding and improve throughput (e.g., Multi-Bin Batching (Guldogan et al., 2024), BucketServe (Zheng et al., 2025)). However, these methods only optimize intra-batch length variance; they do not disaggregate prefills versus re-prefills nor address the compute-memory interference we identify.

3 LENGTH-AWARE-PREFILL SERVE (LAPS)

We develop LAPS to mitigate the interference between long-prefill and short-prefill requests in multi-turn LLM serving. The interference stems from two major factors: (1) their heterogeneous computation characteristics, and (2) head-of-line blocking caused by unified batching. Section 3.1 introduces the strategies we design to optimize high-concurrency short-prefill workloads, while Section 3.2 shows our queue- and instance-level disaggregation mechanism that isolates these request types to reduce interference. LAPS is built upon the prefill instance in the PD disaggregation architecture and extends it with a finer-grained disaggregation design within the prefill stage.

3.1 Short Prefill Optimization

During auto-regressive serving, the majority of end-to-end latency typically arises from the *decode* stage. Conse-

Table 2. Compute waste and latency for padded batch configurations. Config C mixes 7 short (64-token) and 1 long (2048-token) request, padding all to length 2048.

Config	Padded Shape	GEMM Waste	Attn Waste	7B Latency	32B Latency
A: Short-only (8×64)	(8, 64)	0%	0%	19.4 ms	73.7 ms
B: Long-only (1×2048)	(1, 2048)	0%	0%	65.1 ms	270.0 ms
C: Mixed (7×64+1×2048)	(8, 2048)	84.8%	87.4%	567.1 ms	2289.3 ms
TTFT Penalty (C/A)				**29.3**×	**31.0**×

Table 3. GPU kernel time breakdown by category for Qwen2.5-7B and 32B. C/A ratio shows the inflation factor of the mixed batch relative to the short-only batch.

	Qwen2.5-7B				Qwen2.5-32B			
Category	**A**	**B**	**C**	**C/A**	**A**	**B**	**C**	**C/A**
GEMM	11.2	44.7	437.3	38.9×	51.6	209.7	1891.3	36.6×
Activation	4.6	14.0	107.9	23.4×	13.8	48.8	353.7	25.7×
Memory	0.5	1.4	9.1	18.5×	1.7	4.7	30.5	18.4×
Reduce	0.5	1.0	3.8	7.1×	1.6	3.3	11.7	7.3×
Total (ms)	**16.9**	**61.1**	**558.6**	**33.0**×	**68.7**	**266.6**	**2287.8**	**33.3**×

quently, most existing optimization efforts (e.g., PD disaggregation (Zhong et al., 2024), CUDA Graph acceleration (Harish & Narayanan, 2007), and router-based load balancing across decoding instances (Hu et al., 2024b; Jain et al., 2025; Stripelis et al., 2024; Jitkrittum et al., 2025)) have been designed for the decode phase. However, as the diversity of LLM workloads grows (e.g., agent decision-making, chain-of-thought reasoning, and multi-turn task planning), the prefill stage has become an increasingly significant bottleneck, yet its optimization potential remains largely overlooked. Despite this trend, optimization for short and multi-turn prefills remains unexplored.

In the *decode* phase, serving systems widely adopt CUDA Graphs because token-by-token computation is highly repetitive. Each decoding step runs nearly identical kernels with stable batch shapes, while frequent small launches make CPU dispatch overhead non-negligible. As decoding adds one token per step and keeps a fixed graph structure, CUDA Graphs effectively eliminate CPU overhead and reduce latency, improving throughput and responsiveness. In contrast, the *prefill* stage performs full-sequence embedding and attention, where input lengths and batch compositions vary greatly across requests. These dynamics make tensor shapes unstable, preventing CUDA Graph reuse. Prefill is also dominated by large attention GEMMs, making graph capture expensive and rarely amortized. Hence, mainstream serving systems avoid CUDA Graphs in prefill and instead rely on conventional kernel launches or fused-kernel optimizations.

In multi-turn dialogues, each user message triggers a *re-prefill* step that encodes new tokens on top of the cached KV states. Unlike the initial long-prompt prefill, re-prefill excludes the system prompt and contains only new user inputs, resulting in shorter and more uniform sequences. This stable shape pattern matches CUDA Graph's fixed-structure requirement. In practice, most re-prefill segments have only a few dozen tokens, allowing high graph reuse through padding or bucketization (e.g., lengths 8, 16, 32, 64). Compared with the highly dynamic long-prefill, this "short-prefill" regime incurs much lower graph-construction cost and delivers greater performance gains.

Graph capture and bucketization. From the characteristics of intensive *re-prefill* workloads, speculative decoding and token-level routing—although not conventional PD inference—generate numerous short re-prefill requests. Drawing inspiration from EAGLE-2's speculative decoding optimization (Li et al., 2024; 2025), LAPS pre-defines a grid of *power-of-two* prompt-length-batch-size buckets (e.g., $L \in \{8, 16, 32, 64, 128, 256\}$ and $B \in \{1, 2, 4, 8, 16, 32, 64\}$) (Gao et al., 2023). At system initialization, a CUDA Graph is captured for each bucket under the assumption of fixed operator topology and variable memory addresses. During inference, each short-prefill request is padded to the nearest bucket and grouped with others sharing the same (L, B) configuration, thereby maximizing graph reuse with negligible memory overhead.

Graph-aware memory-based batching. To maximize the benefit of CUDA Graphs for short-prefill workloads, LAPS optimizes batching with two goals: (i) reducing Graph launch frequency and (ii) increasing the reuse rate of large-batch Graphs. These are achieved through modestly extended waiting windows and graph fusion. Under high concurrency, slightly delaying batch formation allows more short-prefill requests to accumulate, improving overall efficiency when the saved launch overhead outweighs

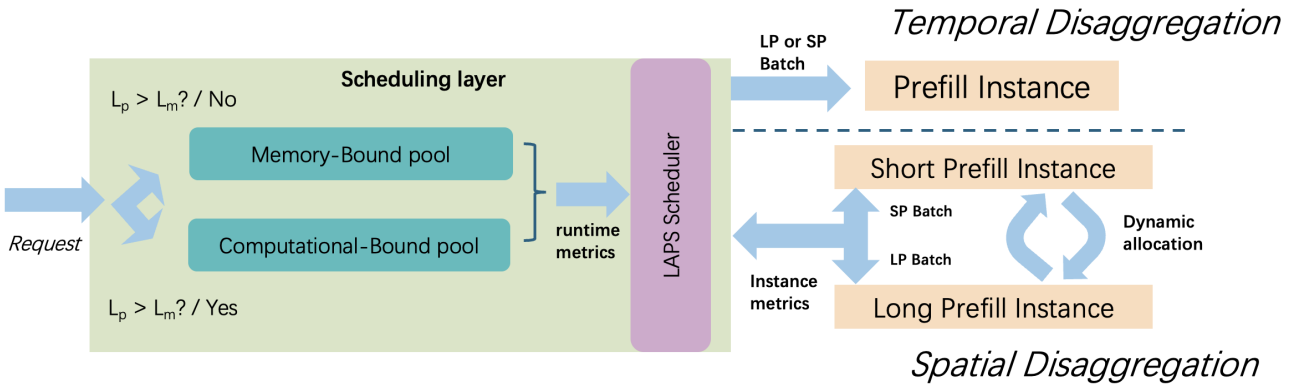

Figure 5. Resource utilization during multi-turn LLM inference. Long-context requests saturate tensor cores during prefill (compute-bound), while short, frequent requests and re-prefill stages are memory-bound with high HBM usage—illustrating the interference between compute- and memory-bound workloads in shared serving systems.

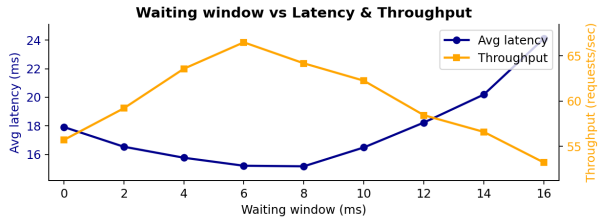

Figure 6. Average latency and throughput curves over varying waiting windows. The larger the waiting window, the more short-prefill requests will be batched. The serving system runs on an H200 GPU and a 14B model, with 64-way concurrency for short-prefill requests (prompt length less than 256 tokens).

the waiting cost. Figure 6 shows the latency-throughput trade-off under different window settings.

While current serving systems typically adopt a memory-constrained batching policy (i.e., aggregating requests until total tokens reach the GPU memory limit), LAPS enhances this approach with *graph awareness*. During short-prefill batching, requests are grouped under the memory budget and aligned to the nearest captured Graph shape.

Our Adaptive Wait-Depth (AWD) scheduler, shown in Algorithm 1, maintains two adaptive thresholds: a waiting window W (the maximum time to wait before dispatch) and a target depth D (the desired batch size aligned to a captured CUDA Graph shape). During each scheduling round, AWD accumulates short-prefill requests until either the waiting window W expires or the target depth D is reached. Requests are greedily grouped by input length to minimize padding, and dispatched early if any request is close to violating its deadline. Before dispatch, the batch is matched to the nearest captured CUDA Graph configuration to maximize graph reuse; otherwise, the standard prefill kernel is used. After each dispatch, W and D are dynamically updated based on the observed fill time and actual batch size for the next scheduling round.

Algorithm 1 AWD: Adaptive-Wait-Depth Batching (Short-Prefill)

Inputs: captured shapes $\mathcal{H}$ (depth, mem), budget M, slack threshold σ, bounds $[W_{\min}, W_{\max}]$, service est. $\widehat{S}$

$W \leftarrow \mathrm{clip}\big(\min_i(\mathrm{DDL}_i - t - \widehat{S}), [W_{\min}, W_{\max}]\big)$ $D \leftarrow \max_{G \in \mathcal{H}:\ \mathrm{mem}(G) \le M} \mathrm{depth}(G)$
while *server running* **do**
  start timer; $B \leftarrow \varnothing$ **while** elapsed $< W$ ***and*** $\mathrm{depth}(B) < D$ **do**
    if $\min_{i \in B \cup \{next\}}(\mathrm{DDL}_i - (t + \widehat{S})) \le \sigma$ **then**
      break ▷ SLA
    add next short request (bucket-first; fit mem)
  $G^\star \leftarrow \textsc{NearestGraph}(B, \mathcal{H}, M)$ ▷ nearest captured shape
  if $G^\star$ *exists* **then**
    pad B to $G^\star$
  else
    use standard prefill kernel
  dispatch B; $d \leftarrow \mathrm{depth}(B)$; $\tau \leftarrow$ time to reach d **if** $d \ge D$ **then**
    $W \leftarrow \mathrm{clip}(\tau, [W_{\min}, W_{\max}])$
  else
    $D \leftarrow d$

3.2 Long-Short Prefill Disaggregation

To fundamentally eliminate the interference between long- and short-prefill requests discussed above, we adopt a design philosophy inspired by PD disaggregation, which further disaggregates long-prefill (LP) and short-prefill (SP) requests. However, unlike PD disaggregation, where the prefill and decode stages exhibit strong temporal dependencies and KV-cache transfers, the two types of tasks are merely mutually exclusive in our LP/SP disaggregation, resulting

in fewer constraints in the scheduling objectives. Consequently, practical PD instance scheduling must account for the capacity coordination between the prefill and decode cluster, as well as the effective interconnect bandwidth when designing resource allocation strategies, but our design provides a larger design space for scheduling strategies that can adapt to the physical compute resources and workload characteristics. To this end, LAPS implements two complementary schedulers: a temporal disaggregation scheduler for single-instance prefill execution, and a spatial disaggregation scheduler for multi-instance prefill coordination. Figure 5 presents the system overview.

Disaggregating prefill execution eliminates direct interference between compute-bound long-prefill and memory-bound short-prefill tasks; however, static separation alone cannot accommodate dynamic workload variations. In real-world deployments, the ratio of long to short requests fluctuates over time, and requests within each category exhibit heterogeneous lengths and deadlines. To address this, LAPS introduces a hierarchical scheduling layer: a temporal disaggregation scheduler is employed within each single prefill instance to manage intra-instance prioritization, while a spatial disaggregation scheduler operates across multiple prefill instances to coordinate inter-instance resource allocation.

It is worth noting that the disaggregation design further amplifies the benefits of CUDA Graphs for short-prefill workloads. In mixed long/short prefill instances, a unified queue containing both request types limits the ability of short-prefill requests to form large, graph-aligned batches. In contrast, by maintaining two independent queues under the disaggregated design, LAPS can determine at request arrival whether CUDA Graph execution should be applied, thereby reducing batching delay and minimizing shape heterogeneity. In mixed queues, the large length disparity between long and short requests leads to excessive padding, lowering the Graph shape hit rate and GPU memory efficiency. After LP/SP separation, requests within each instance exhibit a more concentrated length distribution, improving both Graph reuse and throughput. As a result, the system can achieve higher CUDA Graph reuse and significantly reduce padding overhead.

Mutual exclusion. Prefill execution is *disaggregated by length* at the instance level: each instance exclusively executes one type of prefill task, either short prefill (memory-bound) or long prefill (compute-bound). This mutual exclusion ensures that within an instance, GPU resources are never shared between the two classes, completely avoiding interference arising from scheduling strategies and heterogeneous computational characteristics. All requests are first classified by prompt length L_p using the boundary point L_m. Each class maintains an independent queue: a short queue Q_s and a long queue Q_l.

In real-world scenarios, inference tasks can be categorized into two major types depending on whether an individual request carries a strict TTFT requirement:

(a) SLA-constrained mode: Each request i has an absolute deadline DDL_i, and the scheduler jointly considers *SLA urgency* and *CUDA Graph efficiency*. At the beginning of each decision epoch t, we compute two candidate waiting windows and choose the tighter one:

$$W(t) = \mathrm{clip}\Big(\min\{W_{\mathrm{SLA}}(t),\, W_{\mathrm{GR}}(t)\},\, [W_{\min}, W_{\max}]\Big).$$

The SLA window

$$W_{\mathrm{SLA}}(t) = \max\Big\{0,\, \min_{i\in\mathcal{Q}_s(t)} (\mathrm{DDL}_i - t - \widehat{S}) - \delta\Big\}$$

represents the last safe time to wait before any pending short-prefill request would violate its deadline after one prefill step of duration $\widehat{S}$ (with a small safety margin δ). The Graph window

$$W_{\mathrm{GR}}(t) \approx \frac{\max\{0,\, D - \mathrm{depth}(B)\}}{\max\{\hat{r}_s,\, \varepsilon\}}$$

is the expected time to reach the target batch depth D aligned to the nearest captured CUDA Graph shape, under the estimated short-request arrival rate $\hat{r}_s$. During batching, if the smallest batch slack $\min_{i\in B\cup\{\mathrm{next}\}}(\mathrm{DDL}_i - (t + \widehat{S})) \le \sigma$ or the head-of-line wait exceeds $T_{\max}$, we dispatch immediately. Thus, SLA pressure shortens the waiting window when deadlines are tight, whereas under low SLA pressure, the scheduler may wait up to W_{GR} to aggregate a larger batch and improve CUDA Graph reuse. Long-prefill dispatch continues to advance a single request by fixed-size chunks C_l, and each instance remains exclusive to either short or long mode.

(b) Deadline-free mode: For offline tasks like dataset distillation (Lei & Tao, 2023), each request does not have a preset deadline, and the policy reduces to *token-max* under the same feasibility constraints. The scheduler forms large, shape-similar short-prefill batches to fill the nearest captured CUDA Graph bucket (admit when $\mathrm{tok}(B) \ge M_s$), while long-prefill dispatches a single request with large fixed-size chunks C_l to sustain high arithmetic intensity and maximize throughput.

Temporal disaggregation mode for single instance. LAPS adopts a temporal disaggregation mode, where each GPU instance is dedicated exclusively to either short- or long-prefill execution. Two global queues Q_s and Q_l are maintained for short and long requests, respectively, and each instance pulls tasks only from its own queue. Scheduling decisions within each instance follow the policies described in the previous section: *SLA-first* (near-deadline priority) when deadlines are active, and *token-max* (CUDA Graph aggregation) when no deadline is preset. This exclusive-per-class

Algorithm 2 Lightweight Instance-Pressure Controller

Input : Total N; current (n_s, n_l); control period Δt; cool-down T_{cool}; hysteresis τ; min allocation $n^{\min}$; weights (α, β, γ); robust aggregator $A(\cdot)$

$t_{\text{last}} \leftarrow -\infty$ **while** *server running* **do**
  sleep Δt ▷ `collect per-instance signals for both pools`
  foreach *instance* k *in SHORT pool* **do**
    measure q_k, e_k, u_k; $\psi_k \leftarrow \alpha\, q_k + \beta\, e_k - \gamma\, u_k$;
  foreach *instance* k *in LONG pool* **do**
    measure q_k, e_k, u_k; $\psi_k \leftarrow \alpha\, q_k + \beta\, e_k - \gamma\, u_k$;
  ▷ `robust pool pressures` $(P90)$
  $P_s \leftarrow A(\{\psi_k : k \in \text{SHORT}\})$ $P_l \leftarrow A(\{\psi_k : k \in \text{LONG}\})$ **if** $\text{now} - t_{\text{last}} < T_{\text{cool}}$ **then**
    continue
  ▷ `single-step hill-climb with hysteresis and safeguards`
  if $P_s > (1+\tau)\, P_l$ ***and*** $n_l > n^{\min}$ **then**
    migrate one instance: $n_s \leftarrow n_s + 1$; $n_l \leftarrow n_l - 1$; $t_{\text{last}} \leftarrow \text{now}$;
  else if $P_l > (1+\tau)\, P_s$ ***and*** $n_s > n^{\min}$ **then**
    migrate one instance: $n_l \leftarrow n_l + 1$; $n_s \leftarrow n_s - 1$; $t_{\text{last}} \leftarrow \text{now}$;

execution avoids long-short interference and ensures stable prefill latency under both modes.

Spatial disaggregation mode for multiple instances. In the multi-instance setting, LAPS employs a controller to dynamically balance short- and long-prefill workloads across N GPU instances (see Algorithm 2). Two independent instance pools are maintained: n_s short-prefill instances and $n_l = N - n_s$ long-prefill instances. At each control interval, the controller monitors the queue backlog, SLA deviation, and GPU utilization of each instance to estimate its load pressure. It then compares the overall pressures of the two pools and, after a cool-down period, migrates at most one instance between them when the imbalance exceeds a threshold. This simple feedback control stabilizes P99 latency, prevents oscillation, and keeps GPU utilization high with negligible overhead.

4 EXPERIMENTS

In both online LLM serving and offline LLM tasks (e.g., dataset distillation), the system must handle highly concurrent requests with heterogeneous task types.

We implement and deploy LAPS on an NVIDIA H200 GPU as well as on an 8×H200 multi-GPU cluster. We build it upon SGLang by extending ∼2K lines of code. We evaluate the prototype system under both single- and multi-GPU settings using a variety of workload patterns:

1. Online task: High-concurrency multi-turn conversations with long/short prompts;
2. Offline task: Full dataset distillation without deadline constraints on single requests.

Our evaluations focus on multi-turn conversational workloads. We use LMsys-Chat-1M (Zheng et al., 2024a) and ShareGPT (Zheng et al., 2023) as our datasets, which consist of large-scale, real-world human-assistant conversations collected from ChatGPT and LMsys platforms.

Metrics. We collect several key metrics for the prefill stage, including average and P90 Time-to-First-Token (TTFT), average request per second (RPS), and SLO violation rate (Wang et al., 2024). Note that in our context, as LAPS specifically disaggregates and optimizes the prefill stage, all latency metrics (unless otherwise specified as end-to-end) strictly refer to TTFT.

Baselines. We compare LAPS against SGLang and vLLM, both are state-of-the-art LLM serving systems. *SGLang* is a widely adopted serving system in both academia and industry; it implements continuous batching to improve throughput and radix-attention (Zheng et al., 2024b) to mitigate memory fragmentation during KV-cache allocation. However, neither SGLang nor vLLM supports CUDA Graph during the prefill phase, and their batching policies rely solely on available memory capacity, so they cannot adjust their batching strategies according to the workload characteristics.

4.1 Numerical Results and Analysis

Figure 7 compares the performance of LAPS with SGLang (with PD disaggregation) and its two partial variants (LAPS with CUDA Graph only and LAPS with Disaggregation only) under a sustained client load with varying concurrency levels (from 1 to 64). The requests are drawn from real multi-turn conversations in the ShareGPT-4 dataset. We evaluate three models, Qwen2.5-7/14/32B, under both the single-instance (temporal disaggregation) and 8-instance (spatial disaggregation) settings.

LAPS consistently outperforms SGLang and its two partial variants across all key metrics: RPS, average latency, and P90 latency. The benefits of LAPS's scheduling mechanism and CUDA Graph optimization become more pronounced under high concurrency. Specifically, LAPS achieves up to 20% and 33% higher RPS than the baseline in the single-prefill instance and 8-prefill instance settings, while reducing average latency by 20% respectively.

We evaluate the effectiveness of LAPS under different workload types in Figure 8. On the Alibaba-NLP/ZeroSearch dataset (Sun et al., 2025), which represents an agent-like,

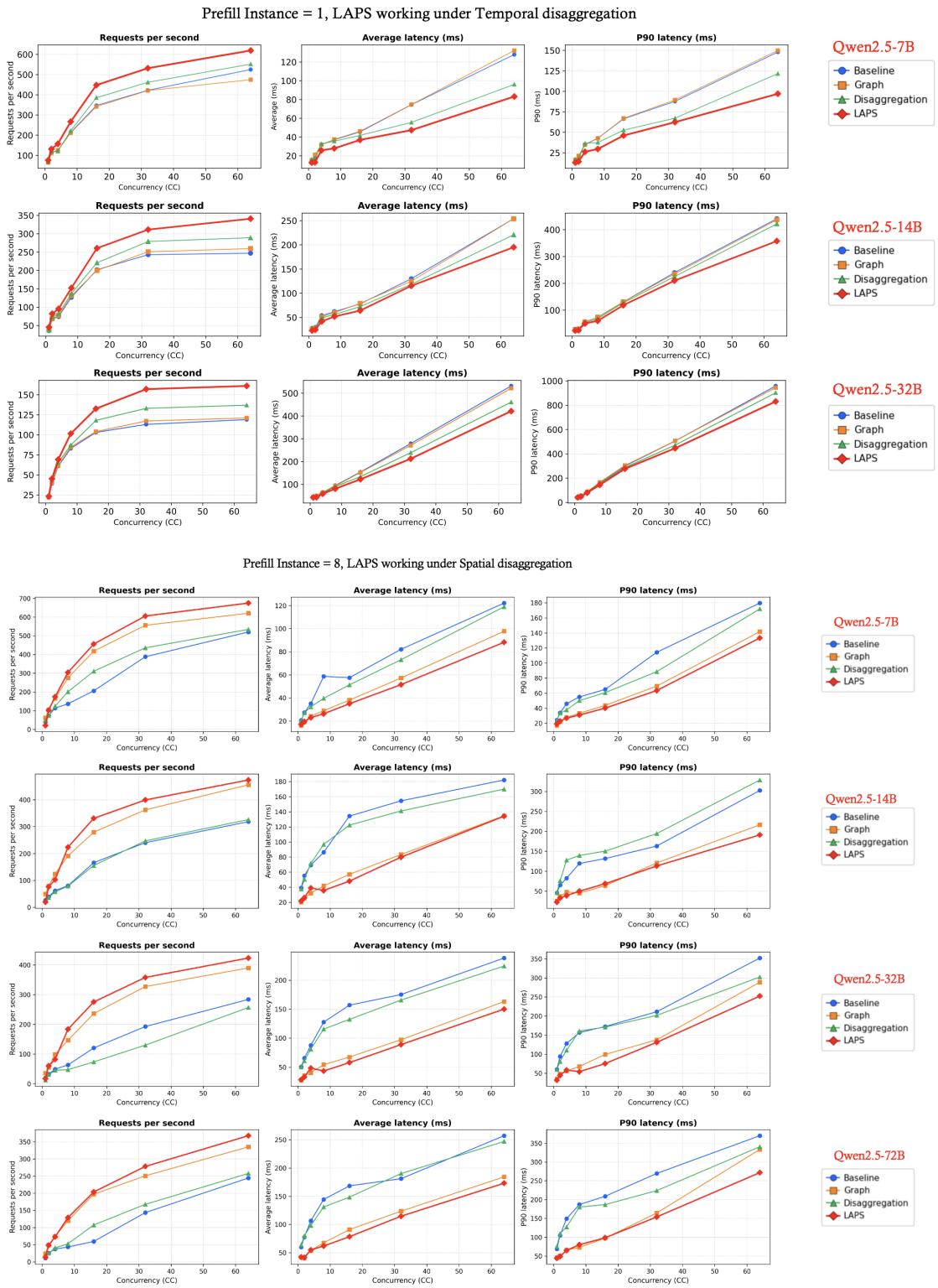

Figure 7. Comparison of LAPS and Vanilla SGLang on one H200 node, processing LMSYS-chat-1M dataset. The top three lines of figures correspond to the *temporal disaggregation* setting (with prefill instance of 1), while the bottom four lines of figures correspond to the *spatial disaggregation* setting (with prefill instance of 8). We report RPS, average latency, and P90 latency under four configurations: *Vanilla SGLang PD disaggregation (blue line)*, *LAPS (only CUDA Graphs enabled, orange line)*, *LAPS (only disaggregation enabled, green line)*, and *Full LAPS (red line)*. More results can be found in Appendix B.

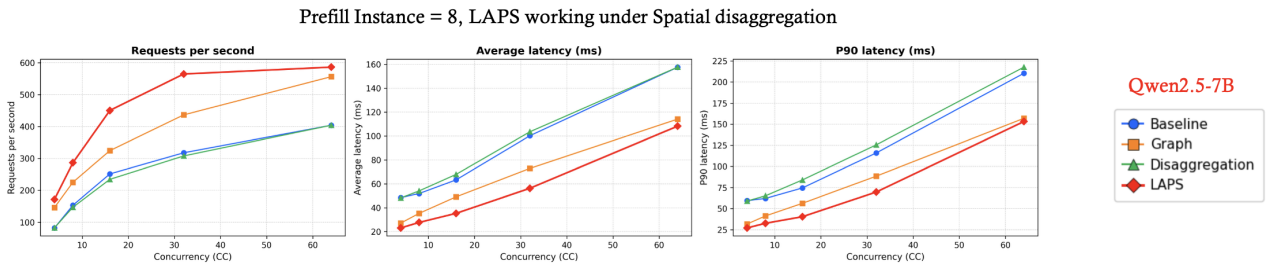

Figure 8. Comparison of LAPS and vanilla SGLang on a single H200 node using the Alibaba-NLP/ZeroSearch (Sun et al., 2025) dataset using Qwen2.5-7B model.

search-augmented workload, 82.9% of requests have prefill lengths shorter than 256 tokens. Despite the highly skewed length distribution, LAPS continues to provide substantial performance improvements. More results related to other model families and architecture (such as MoE model) can be found in Appendix B.

It is worth noting that, in some configurations, enabling CUDA Graphs alone yields limited improvements and can even degrade throughput, as the overhead of graph eligibility checking and graph launching becomes non-negligible. Enabling disaggregation, however, allows the system to dynamically adjust the waiting window size and form larger batches, thereby amplifying the effective performance gains from CUDA Graph execution and making its scheduling and launch overhead negligible.

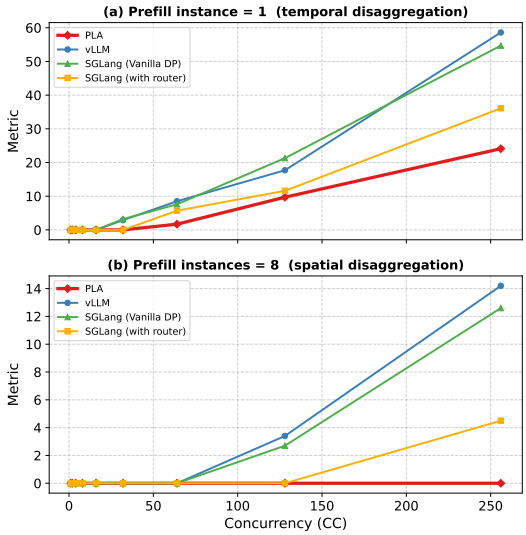

Figure 9. SLO violation rate under varying client concurrency levels using the LMsys-Chat-1M dataset. Results are shown for *LAPS*, *SGLang (PD disaggregation)*, *SGLang (PD disaggregation with router)*, and *vLLM (PD disaggregation)* under two settings: (top) single-instance (temporal disaggregation) and (bottom) 8-instance (spatial disaggregation).

In Figure 9, we use the LMsys-Chat-1M dataset and assume that request arrivals follow a Poisson process with an average arrival rate λ, while each request's service time follows an empirical distribution measured from model execution. We set the TTFT SLO to 0.4s and vary the client-side concurrency levels to observe the actual SLO violation rate. *SGLang* supports data parallel (DP) serving based on a router that dispatches requests to different workers using either round-robin or load-balancing strategies; however, the router is unaware of the SLOs of individual requests. As shown in the figure, within a single instance, *LAPS* reduces the SLO violation rate by approximately 10% compared with *SGLang (PD disaggregation) with router*, and by around 30% compared with *Vanilla DP*. Under the 8-instance spatial disaggregation setting, *LAPS* achieves zero SLO violations, whereas *SGLang* with router still exhibits a 4.7% violation rate.

In Figure 10, we evaluate the compatibility of LAPS under non-PD-disaggregated settings by mixing prefill and decode requests at different concurrency levels. In both single-instance and multi-instance configurations, the request-per-second (RPS) of prefill requests decreases under the *Mix with Decode* condition, indicating that LAPS can fully exploit the throughput benefits of CUDA Graphs for short prefill requests only within the PD-disaggregated architecture. When mixed with decode workloads, the lack of continuous batching introduces additional inter-batch latency, resulting in degraded overall performance.

As shown in Table 4, we evaluate the distillation task where each request has no strict deadline (i.e., the waiting window can be relatively large). Under this setting, with four prefill instances and a decoding length of 1K tokens, LAPS achieves about an 8% reduction in time consumption compared to SGLang (Vanilla PD disaggregation).

4.2 Cost analysis of CUDA Graph

In LAPS deployment, CUDA Graphs are captured into memory during initialization. Each graph is bound to a fixed kernel configuration and cannot adapt to dynamic kernel sizes, so multiple graphs must be captured to cover different token

Table 4. End-to-end time of PD-disaggregated serving with 4 prefill and 4 decode instances, distilled on two dialogue datasets.

System Setting	LMSys	ShareGPT
4 P (SGLang) + 4 D	18,963s	25,132s
4 P (LAPS) + 4 D	17,572s	23,041s
Improvement	*7.3%*	*8.3%*

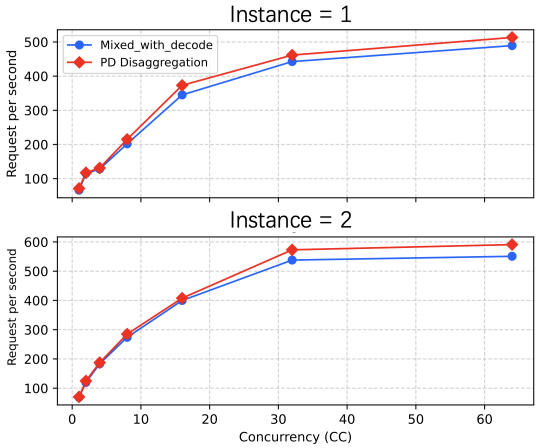

Figure 10. Comparison of prefill throughput between PD disaggregation and Mixed with Decode across different concurrency levels under single- and 2-instance settings.

lengths and batch sizes. Each prefill step introduces lookup and selection overhead, and thus, the number of graphs must be limited to balance memory usage and performance. We measure single-graph sizes of 228 MB, 240 MB, and 277 MB for the 7B, 14B, and 32B models, showing that graph size is largely insensitive to model scale. When the system is initialized for the first time, it needs to capture kernels and the KV-cache operations layer by layer, which introduces a certain startup overhead. Experiments show that capturing a single prefill graph incurs an initialization overhead of approximately 8-12 seconds.

Graph Matching Strategy In CUDA graph-based batch prefill, requests are padded to a uniform captured shape. The choice between a single graph (memory-first) and multiple tighter-fitting graphs (latency-first) depends on the compute–memory regime. For short, memory-bound prefills, latency is dominated by HBM weight loading and is largely insensitive to padding, making fewer graph launches preferable. For long, compute-bound prefills, GEMM time scales with padded token count, so padding waste directly increases latency, favoring finer-grained graph partitioning. Mixed-length batches represent the worst case for a single-graph strategy, as short requests incur both wasted FLOPs and inflated KV-cache allocation proportional to the length gap.

Implications for LAPS This observation motivates LAPS's dual-queue design, which separates short and long prefills prior to batching. By ensuring length-homogeneous batches, LAPS minimizes padding within each CUDA graph and makes even simple memory-first matching effective. Without such separation, mixed batches introduce structural inefficiency: short requests are padded to the longest sequence, wasting compute in the compute-bound regime and KV-cache memory across all regimes. LAPS thus reduces graph matching from a complex bin-packing problem over heterogeneous lengths to a simple nearest-bucket lookup over homogeneous ones.

4.3 Discussion

In the memory-bound regime, padding introduces negligible compute overhead, as latency is dominated by HBM weight loading and is largely insensitive to sequence length. As shown in our case study (Section 2), for Qwen2.5-7B on H200 this regime corresponds to prefills shorter than ∼200 tokens (ridge point at ∼206 FLOP/byte). In this regime, batching short prefills is particularly effective: grouping requests reduces kernel launch overhead and improves hardware utilization with minimal additional cost.

In contrast, once prefill becomes compute-bound, latency scales with token count and padding waste directly translates into increased computation. Therefore, mixing heterogeneous lengths is inefficient in this regime. This motivates LAPS to batch short prefills aggressively while separating long prefills to avoid compute waste.

5 Conclusion

In this paper, we propose *LAPS*, a prefill-length-aware LLM serving system built on the PD disaggregation paradigm to optimize heterogeneous multi-turn conversational workloads. By separating long- and short-prefill requests, LAPS eliminates compute–memory interference in the prefill stage. Its adaptive scheduler (AWD) and CUDA Graph–based execution improve batching efficiency and reduce short-prefill latency. Supporting both temporal and spatial disaggregation, LAPS scales across single- and multi-prefill-instance deployments. Experiments on real-world datasets show that LAPS achieves higher throughput and lower latency than state-of-the-art frameworks (e.g., SGLang and vLLM under PD disaggregation), demonstrating its effectiveness under high concurrency.

REFERENCES

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills, 2023. URL https://arxiv.org/abs/2308.16369.

Anthropic. https://www.anthropic.com/api, 2024. Accessed: 2025-10-30.

Chatterji, A., Cunningham, T., Deming, D. J., Hitzig, Z., Ong, C., Shan, C. Y., and Wadman, K. How people use chatgpt. Technical report, National Bureau of Economic Research, 2025.

Corporation, N., 2023. URL https://github.com/NVIDIA/TensorRT-LLM. Apache-2.0 License; accessed 2025-10-30.

Dam, S. K., Hong, C. S., Qiao, Y., and Zhang, C. A complete survey on llm-based ai chatbots, 2024. URL https://arxiv.org/abs/2406.16937.

Foundation, A. S. https://apisix.apache.org/blog/2025/02/24/apisix-ai-gateway-features/, 2024. Accessed: 2025-10-30.

Gao, H., Qiu, B., Wang, Y., Yu, S., Xu, Y., and Wang, X. Tbdb: Token bucket-based dynamic batching for resource scheduling supporting neural network inference in intelligent consumer electronics. *IEEE Transactions on Consumer Electronics*, 70(1):1134–1144, 2023.

Guldogan, O., Kunde, J., Lee, K., and Pedarsani, R. Multi-bin batching for increasing llm inference throughput. *arXiv preprint arXiv:2412.04504*, 2024.

Harish, P. and Narayanan, P. J. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pp. 197–208. Springer, 2007.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024a. URL https://arxiv.org/abs/2401.11181.

Hu, Q. J., Bieker, J., Li, X., Jiang, N., Keigwin, B., Ranganath, G., Keutzer, K., and Upadhyay, S. K. Routerbench: A benchmark for multi-llm routing system, 2024b. URL https://arxiv.org/abs/2403.12031.

HuggingFace. Text generation inference documentation. https://huggingface.co/docs/text-generation-inference/en/index, 2024.

Inc., C. https://developers.cloudflare.com/api/, 2024a. Accessed: 2025-10-30.

Inc., K. https://docs.konghq.com/hub/kong-inc/rate-limiting/, 2024b. Accessed: 2025-10-30.

Jain, K., Parayil, A., Mallick, A., Choukse, E., Qin, X., Zhang, J., Íñigo Goiri, Wang, R., Bansal, C., Rühle, V., Kulkarni, A., Kofsky, S., and Rajmohan, S. Intelligent router for llm workloads: Improving performance through workload-aware load balancing, 2025. URL https://arxiv.org/abs/2408.13510.

Jin, Y., Wang, T., Lin, H., Song, M., Li, P., Ma, Y., Shan, Y., Yuan, Z., Li, C., Sun, Y., Wu, T., Chu, X., Huan, R., Ma, L., You, X., Zhou, W., Ye, Y., Liu, W., Xu, X., Zhang, Y., Dong, T., Zhu, J., Wang, Z., Ju, X., Song, J., Cheng, H., Li, X., Ding, J., Guo, H., and Zhang, Z. P/d-serve: Serving disaggregated large language model at scale, 2024. URL https://arxiv.org/abs/2408.08147.

Jitkrittum, W., Narasimhan, H., Rawat, A. S., Juneja, J., Wang, C., Wang, Z., Go, A., Lee, C.-Y., Shenoy, P., Panigrahy, R., et al. Universal model routing for efficient llm inference. *arXiv preprint arXiv:2502.08773*, 2025.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Lei, S. and Tao, D. A comprehensive survey of dataset distillation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(1):17–32, 2023.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding, 2023. URL https://arxiv.org/abs/2211.17192.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-2: Faster inference of language models with dynamic draft trees, 2024. URL https://arxiv.org/abs/2406.16858.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle-3: Scaling up inference acceleration of large language models via training-time test, 2025. URL https://arxiv.org/abs/2503.01840.

Meini, B. Solving m/g/l type markov chains: recent advances and applications. *Stochastic Models*, 14(1-2): 479–496, 1998.

Neuts, M. F. Generalizations of the pollaczek-khinchin integral equation in the theory of queues. *Advances in applied probability*, 18(4):952–990, 1986.

OpenAI. https://openai.com/api/pricing, 2024. Accessed: 2025-10-30.

Proxy, E. Envoy gateway: Rate limiting and token-based access control. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rate_limit_filter, 2024. Accessed: 2025-10-30.

Qiao, A., Choe, S. K., Subramanya, S. J., Neiswanger, W., Ho, Q., Zhang, H., Ganger, G. R., and Xing, E. P. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

Qwen, :, Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Tang, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., and Qiu, Z. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

She, J., Zheng, W., Liu, Z., Wang, H., Xing, E., Yao, H., and Ho, Q. Token level routing inference system for edge devices, 2025. URL https://arxiv.org/abs/2504.07878.

Strati, F., Mcallister, S., Phanishayee, A., Tarnawski, J., and Klimovic, A. Déjàvu: Kv-cache streaming for fast, fault-tolerant generative llm serving, 2024. URL https://arxiv.org/abs/2403.01876.

Stripelis, D., Hu, Z., Zhang, J., Xu, Z., Shah, A. D., Jin, H., Yao, Y., Avestimehr, S., and He, C. Tensoropera router: A multi-model router for efficient llm inference. *arXiv preprint arXiv:2408.12320*, 2024.

Sun, H., Qiao, Z., Guo, J., Fan, X., Hou, Y., Jiang, Y., Xie, P., Zhang, Y., Huang, F., and Zhou, J. Zerosearch: Incentivize the search capability of llms without searching, 2025. URL https://arxiv.org/abs/2505.04588.

Wang, Z., Li, S., Zhou, Y., Li, X., Gu, R., Cam-Tu, N., Tian, C., and Zhong, S. Revisiting slo and goodput metrics in llm serving. *arXiv preprint arXiv:2410.14257*, 2024.

Wölflein, G., Ferber, D., Truhn, D., Arandjelović, O., and Kather, J. N. Llm agents making agent tools, 2025. URL https://arxiv.org/abs/2502.11705.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Yuan, Z., Shang, Y., Zhou, Y., Dong, Z., Zhou, Z., Xue, C., Wu, B., Li, Z., Gu, Q., Lee, Y. J., Yan, Y., Chen, B., Sun, G., and Keutzer, K. Llm inference unveiled: Survey and roofline model insights, 2024. URL https://arxiv.org/abs/2402.16363.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., and Stoica, I. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., Gonzalez, J. E., Stoica, I., and Zhang, H. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2024a. URL https://arxiv.org/abs/2309.11998.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024b. URL https://arxiv.org/abs/2312.07104.

Zheng, W., Xu, M., Song, S., and Ye, K. Bucketserve: Bucket-based dynamic batching for smart and efficient llm inference serving. *arXiv preprint arXiv:2507.17120*, 2025.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024. URL https://arxiv.org/abs/2401.09670.

A ARTIFACT APPENDIX

A.1 Abstract

LAPS (Original paper title is : LAPS: A Prefill-Length-Aware LLM Serving System) is a prefill-oriented scheduling framework for prefill-decode (PD) disaggregated LLM serving, implemented on top of SGLang. The artifact includes the full source code, benchmark scripts, and result data needed to reproduce the key experiments in the paper. Specifically, the artifact demonstrates: (1) dual-queue scheduling that separates short and long prefill requests to eliminate head-of-line blocking, (2) a batch waiting window that accumulates same-length requests for higher GPU utilization, and (3) dynamic GPU allocation that adjusts the prefill-decode split at runtime based on queue pressure. The main benchmark sweeps concurrency levels from 1 to 64 on Qwen2.5 model, from 7b to 72b with 8*NVIDIA H200 GPUs under PD disaggregation, measuring prefill throughput (req/s) and P90 TTFT (ms).

A.1.1 Project URL

https://github.com/Jianshu-She/LAPS

For Quick Artifact Evaluation To provide the best convinence for the evaluation, we create a script in https://github.com/Jianshu-She/LAPS/tree/main/bench_laps_prefill_throughput to automatically run all of the experiments to reproduce the results we reported in the paper.

A.1.2 Minimal Hardware Requirement

Best Minimal requirement: Nvidia-H200 *8 (InfiniBand)

A.1.3 Best Minimal Software Requirement

LAPS was built on the SGLang v0.3.6 and currently update to v0.4.6.post5, all the software requriments are the same as SGLang v0.4.6.post.

A.1.4 How LAPS support the Paper

A comprehensive summary of all code modifications and implementation details is available in our repository at https://github.com/Jianshu-She/LAPS/blob/main/docs/code_changes.md. These features directly correspond to the system design presented in Section 3 of the LAPS paper.

Development History Notably, LAPS was originally developed on top of SGLang v0.3.6, a version that lacked native support for Piecewise CUDA graphs—a feature we custom-built for LAPS. While the upstream SGLang team has since introduced their own Piecewise CUDA graph support for prefill, their implementation details differ fundamentally from ours. Under the experimental settings described in our paper, the current version of LAPS continues to outperform vanilla SGLang. In fact, due to recent codebase refinements, the overall throughput reproduced by this artifact may even slightly exceed the results originally reported in the paper.

A.1.5 How to verify the result in Paper

Quick Eval See https://github.com/Jianshu-She/LAPS/tree/main/bench_laps_prefill_throughput for instruction about quick eval.

Further Function Details about LAPS The instructions of launching LAPS under different model are reported in https://github.com/Jianshu-She/LAPS/blob/main/docs/laps_scheduler.md.

A.1.6 Expected Outcomes

The scripts should generate the results that reported in the README.md https://github.com/Jianshu-She/LAPS/blob/main/README.md in the LAPS repo.

A.2 Artifact check-list (meta-information)

- **Algorithm:** LLM inference, PD-disaggregation
- **Program:** LAPS v1.0, built on SGLang v0.4.6.post5; Python (SGLang runtime, router); CUDA (attention kernels via FlashAttention
- **Compilation:** pip install from source; no special compilation beyond standard SGLang build (requires PyTorch, FlashAttention, Triton)
- **Transformations:** NA
- **Binary:** NA
- **Data set:** LMSYS-Chat-1M
- **Run-time environment:** Linux (Ubuntu 20.04+), Python 3.10+, PyTorch 2.4+, CUDA 12.4+, Mooncake KV transfer backend
- **Hardware:** NVIDIA H200 GPUs * 8 (minimum 2 GPUs for PD disaggregation benchmarks) with IB
- **Run-time state:** NA
- **Execution:** Automated via shell scripts;
- **Metrics:** Prefill throughput (req/s), mean TTFT, P90 TTFT
- **Output:** JSON files with per-run metrics; summary table (text) with throughput and TTFT across concurrency levels
- **Experiments:** 3 different setting: Vanilla SGLang with PD disaggregation; SGLang with prefill cuda graph; LAPS
- **How much disk space required (approximately)?:** 200GB

- **How much time is needed to prepare workflow (approximately)?:** 30 mins
- **How much time is needed to complete experiments (approximately)?:** 1 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Data licenses (if publicly available)?:** LMSYS-Chat-1M is publicly available under CC-BY-NC-4.0
- **Workflow framework used?:** Bash scripts + Python asyncio benchmark client
- **Archived (provide DOI)?:** https://github.com/Jianshu-She/LAPS

A.3 Description

A.3.1 How delivered

Obligatory

The artifact is delivered as a public GitHub repository: https://github.com/Jianshu-She/LAPS

The repository is a fork of SGLang v0.4.6.post5 with LAPS-specific modifications (LAPS v1.0). The `main` branch contains all features merged together. The benchmark dataset is included at `data/lmsys_chat_10k.jsonl`. Please see the repo README.md for details.

A.3.2 Hardware dependencies

- 8× NVIDIA H200 GPUs (or A100/H100 with sufficient memory for Qwen2.5-32B/72B)
- InfiniBand or RoCE network interface for Mooncake KV cache transfer backend (e.g., `mlx5_0`)
- ≥200 GB host memory

A.3.3 Software dependencies

- Linux (tested on Ubuntu 20.04 with kernel 5.15+)
- Python 3.10+
- CUDA 12.4+ and cuDNN 9+
- PyTorch 2.4+
- FlashAttention 2.6+
- Mooncake transfer engine (for PD disaggregation KV cache transfer)
- SGLang dependencies (installed via `pip install -e ".[all]"`)

A.3.4 Data sets

URL : https://github.com/Jianshu-She/LAPS/tree/main/data

The benchmark uses a subset of the LMSYS-Chat-1M dataset (10,000 human prompts) stored as a JSONL file. Each line contains a `"text"` field with a real user prompt. The dataset is used to drive prefill-only workloads (`max_new_tokens=1`) to isolate prefill throughput.

A.4 Installation

Obligatory

Quick Installation

For quick installation, using the install script: https://github.com/Jianshu-She/LAPS/blob/main/install.sh

Manually Installation

1. Clone the repository:

```
git clone https://github.com/Jianshu-She/LAPS.git
cd LAPS
```

2. Create a conda environment and install dependencies:

```
conda create -n laps python=3.12
conda activate laps
pip install -e ".[all]"
```

3. Install the Mooncake transfer backend following SGLang's PD disaggregation documentation.

4. Download the model:

```
huggingface-cli download Qwen/Qwen2.5-32B
```

A.5 Experiment workflow

See LAPS repo: https://github.com/Jianshu-She/LAPS/tree/main/bench_laps_prefill_throughput for quick evaluation and experiment workflow details.

A.6 Evaluation and expected result

Obligatory

The primary claim validated by this artifact is that LAPS improves prefill throughput and reduces TTFT compared to vanilla SGLang under PD disaggregation.

Expected results for the concurrency sweep (Qwen2.5-72B, 8× H200):

Table 5. Request Throughput (req/s) for Qwen2.5-72B-Instruct with TP=4 and one prefill instance (4×H200).

Setting	cc=4	cc=8	cc=16	cc=32	cc=64
Vanilla SGLang	40.8	53.5	108.5	167.5	255.4
Prefill Disaggregation	24.3	42.3	60.4	90.8	203.7
LAPS	**57.6**	**129.4**	**200.7**	**270.4**	**369.1**

More results can be found in LAPS repo: https://github.com/Jianshu-She/LAPS/tree/main/bench_laps_prefill_throughput

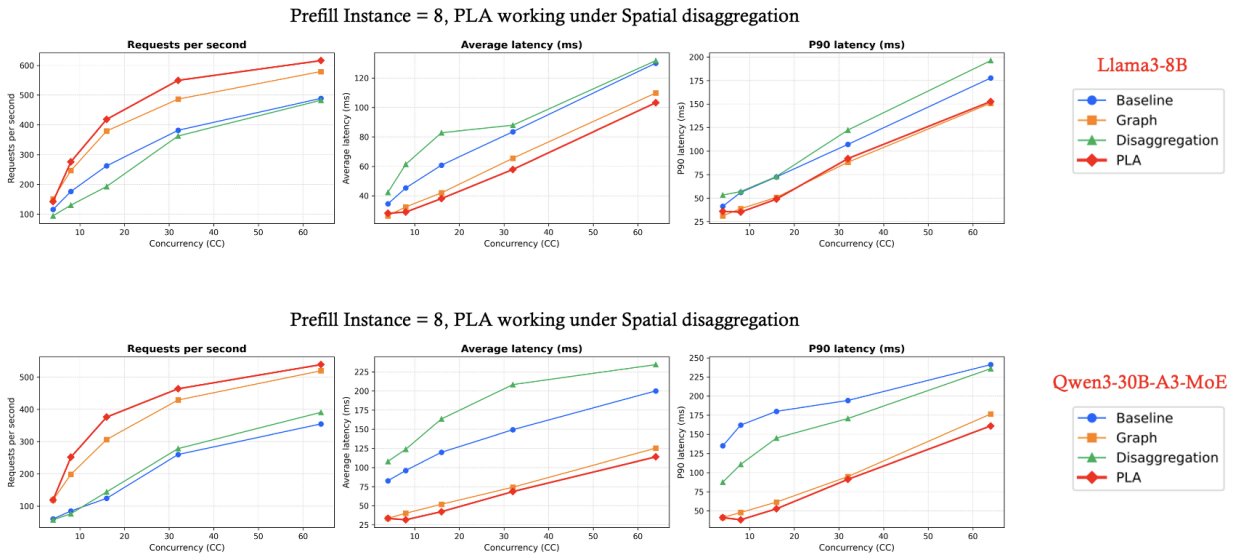

Figure 11. Comparison of LAPS and vanilla SGLang on a single H200 node using the LMSYS-Chat-1M dataset across different model families and architectures.

Table 6. Mean Time-to-First-Token (TTFT) in milliseconds.

Setting	**cc=4**	**cc=8**	**cc=16**	**cc=32**	**cc=64**
Vanilla SGLang	97.9	149.4	147.3	190.6	249.6
Prefill Disaggregation	164.7	189.0	264.8	351.9	313.0
LAPS	**69.4**	**61.7**	**79.5**	**117.9**	**172.6**

B RESULTS APPENDIX

We include additional results across diverse model architectures and workloads in this section. Figure 11 presents results for Llama-3.3-8B and Qwen3-30B-A3B (MoE). Across request rates ranging from 4 to 64 requests per second, both average latency and P90 latency demonstrate the effectiveness of LAPS across different model architectures.