PLA-Serve

A Prefill-Length-Aware LLM Serving System

Jianshu She · Zonghang Li · Hongchao Du · Shangyu Wu · Wenhao Zheng
Eric Xing · Zhengzhong Liu · Huaxiu Yao · Jason Xue · Qirong Ho

Mohamed bin Zayed University of AI · UNC Chapel Hill

MLSys 2026 ·Seattle

Mohamed bin Zayed University of Artificial Intelligence

The University of North Carolina at Chapel Hill

Wechat:

Linkin:

New title: "LAPS: A Length-Aware-Prefill LLM Serving System"

Email: jianshushe@gmail.com

Background: Modern LLM Serving Stack

LLM inference splits into two phases with fundamentally different computational characteristics:

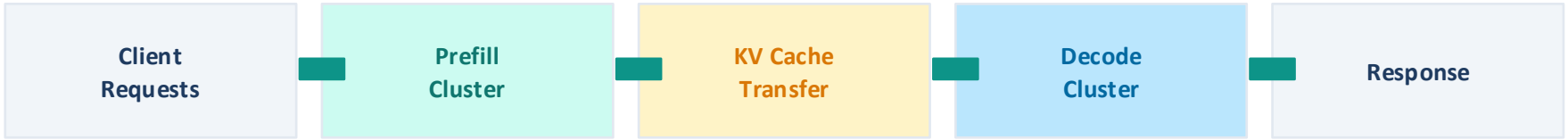
Prefill Stage

- Processes entire prompt at once
- Compute-bound (large GEMMs)
- Produces KV cache → determines TTFT
- Append new prefill tokens to current KV-cache (re-prefill)

Decode Stage

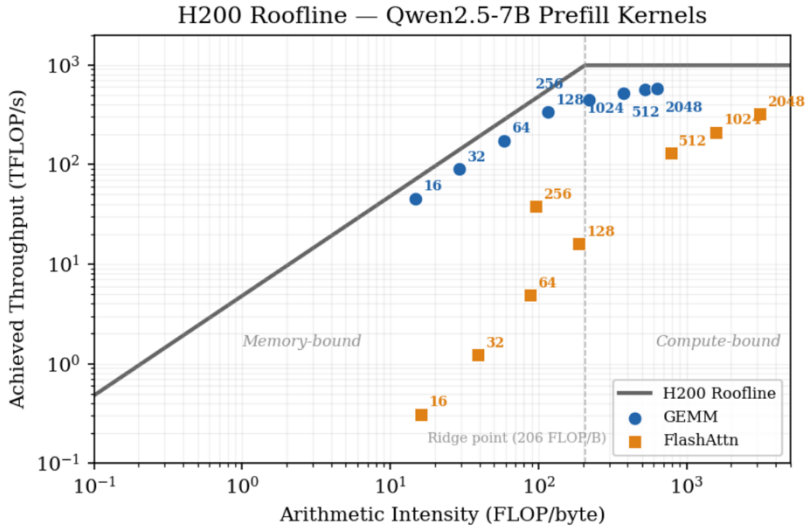
- Generates tokens one-by-one
- Memory-bound (KV cache I/O)
- Autoregressive → determines throughput

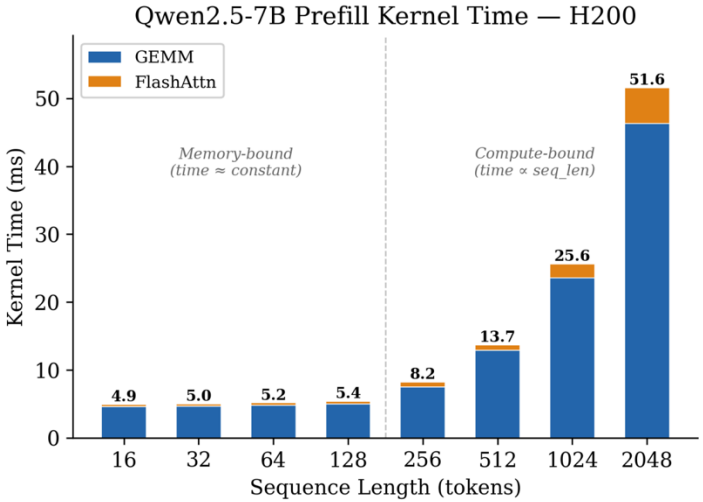
PD Disaggregation (current SOTA) — runs prefill & decode on separate GPU instances:



Characteristic of Prefill Request

Analysis of Prefill request on H200 GPU: Roofline Model vs Kernel Time

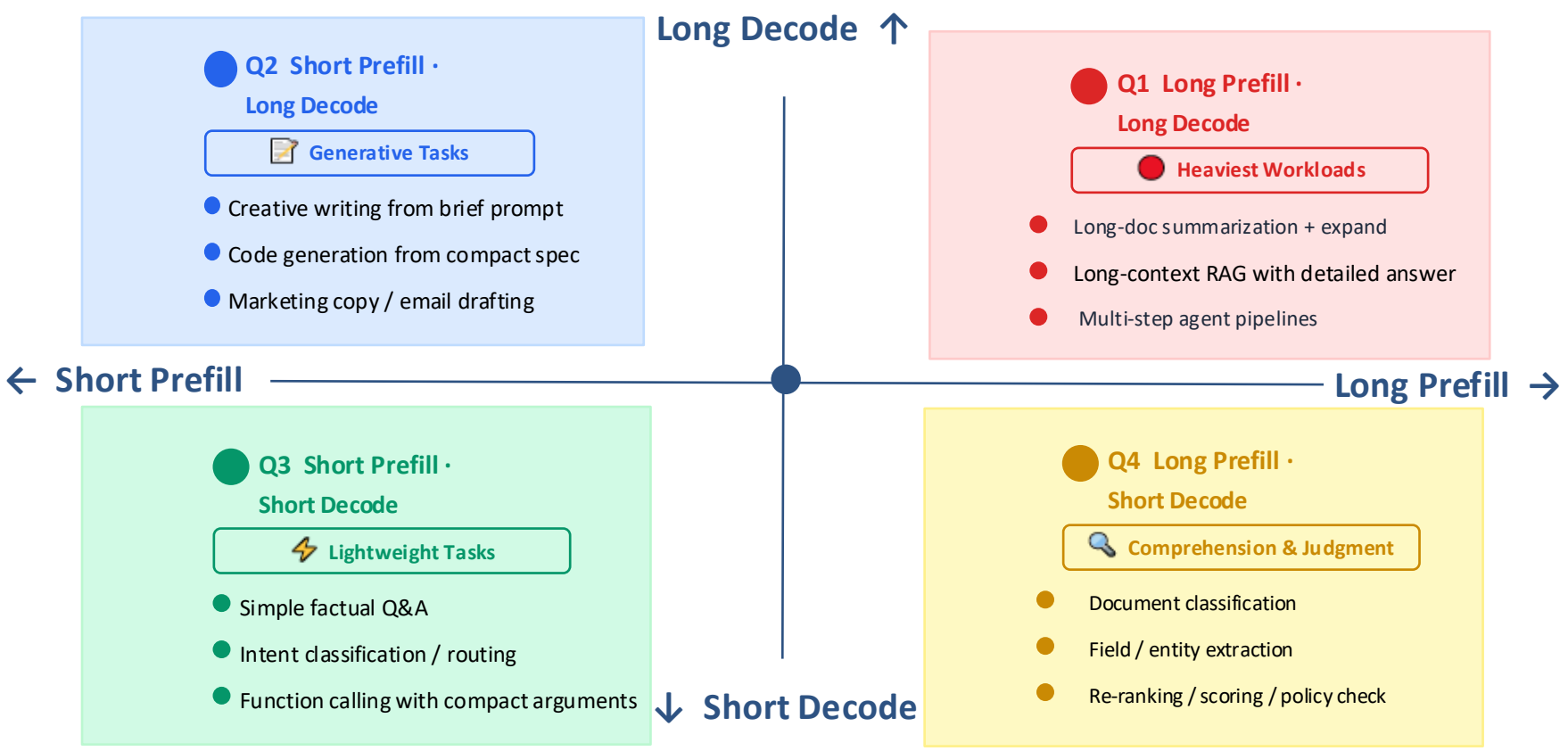




Re-Prefill in multi-turn task will have lower arithmetic intensity due to the large amount of KV-cache history !

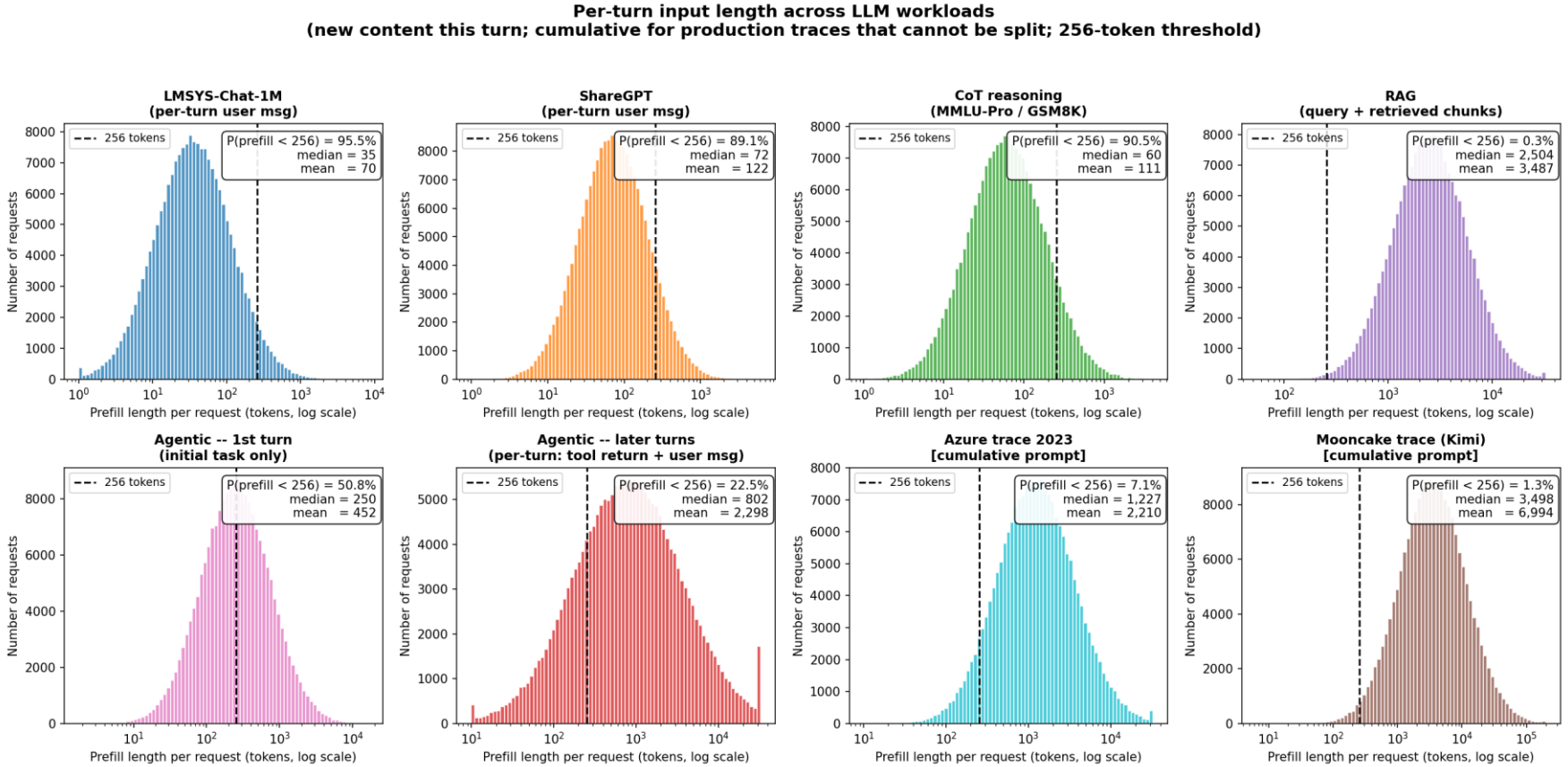
Background: Real-World LLM Serving Workload

Motivation



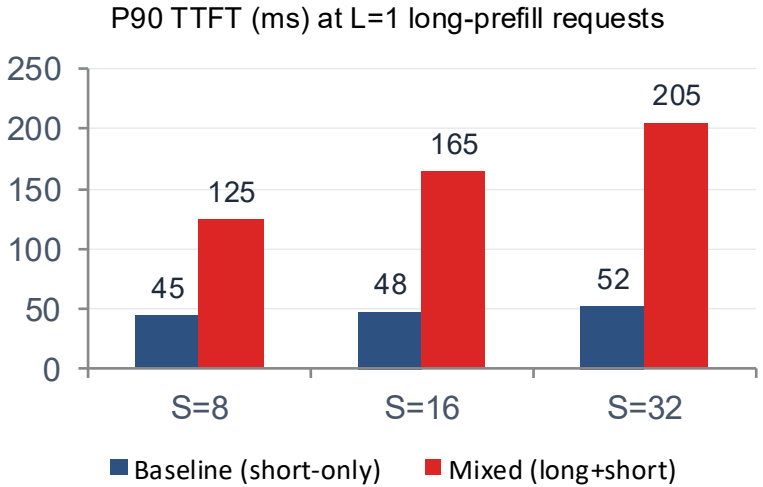
Background: Real-World LLM Serving Workload

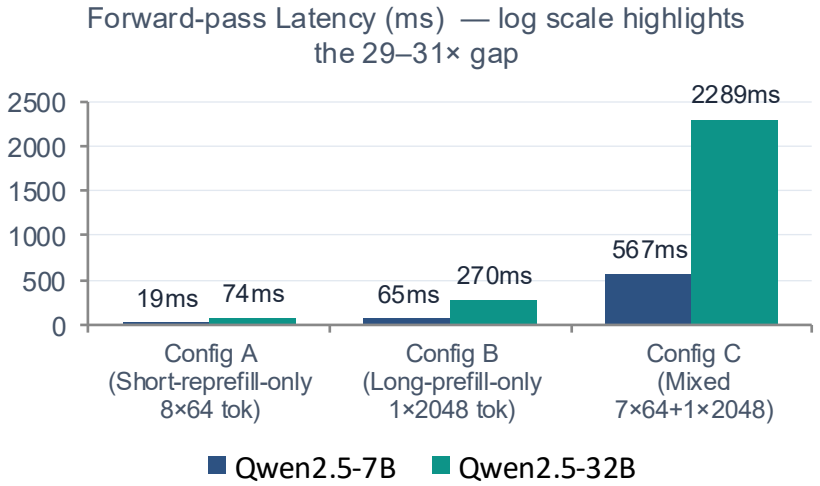
Motivation



Motivation: Intra Prefill Task interference

When Long Prefill (LP) and Short Prefill (SP) Task mixed within the same batch to process:





1. Head-of-Line Blocking 2. Compute/Memory bound interference

Theoretical Analysis: Latency Model & Scheduling Interference

1 Head-of-Line Blocking (M/G/1 Queue)

When a long job sits at the head, short jobs must wait:

Extra waiting time imposed on all jobs:

$$\Delta W = \lambda \cdot p(1-p) \cdot (S_l - S_s)^2 \;/\; 2(1-\rho) \;>\; 0$$

- λ — arrival rate of requests
- p — fraction of long jobs in the mix
- S_l, S_s — service time: long vs short jobs
- ρ — server utilization ($\rho < 1$)

$(S_l - S_s)^2$ grows with heterogeneity → penalty explodes as mix gets more diverse

2 Prefill Latency Decomposition

T (L, H) has 2 part: T_compute & T_memory

L = new request tokens | *H = history KV tokens*

T_compute (L, H)

$$\approx \alpha L(L+2H) + \beta L$$

$\alpha \cdot L(L+2H)$ — attention (quadratic)
$\beta \cdot L$ — FFN (linear)

T_memory (L, H)

$$\approx \gamma_w \cdot L + \gamma_r \cdot H$$

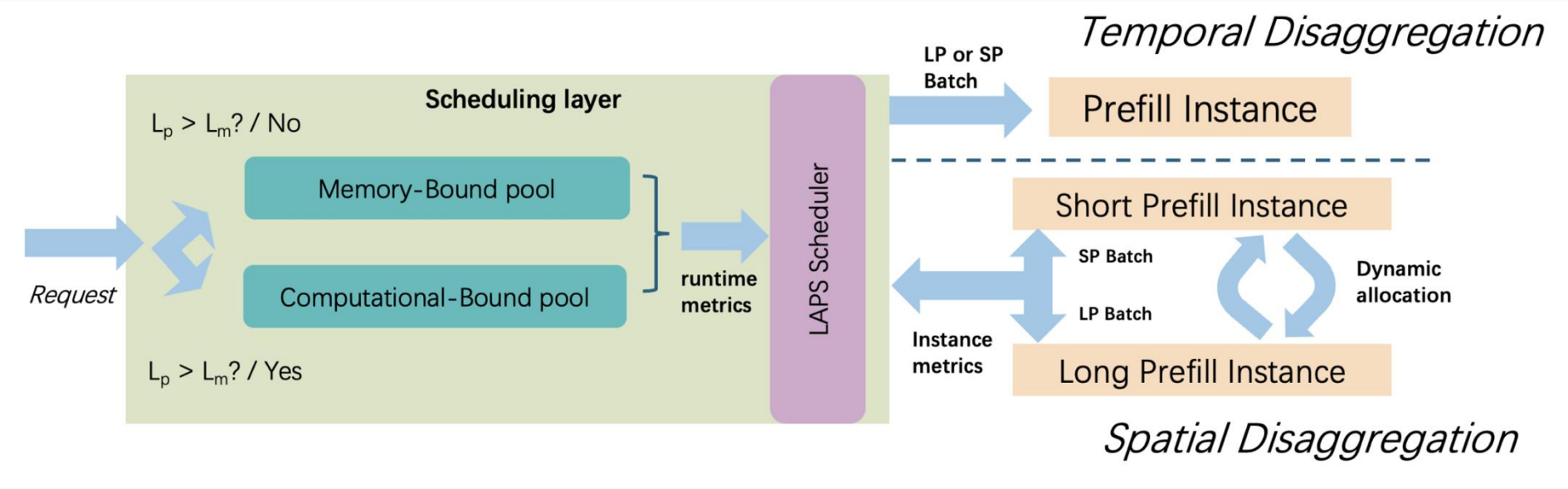
$\gamma_w \cdot L$ — Write KV · $\gamma_r \cdot H$ — Read history

First-turn vs Re-prefill

H=0 (first turn)	$= \gamma_w \cdot L$
H>>0 (re-prefill)	$= \gamma_w \cdot L + \gamma_r \cdot H$

Boundary $L_m = (\gamma_w - \beta) / \alpha \approx 256$ tokens (H200)

Our Solution: Length-Aware-Prefill LLM Serving System



- *Profiling for heterogenous GPU metrics to define the threshold of Short Prefill*
- *Maintaining 2 type of Queue*
- *Scheduling request on temporal/ spatial disaggregation mode*

Innovation 1: Prefill Disaggregation — Dynamic Instance Balancing

N GPU instances split into two pools. A lightweight controller watches each instance every Δt seconds and rebalances when needed.

How Pressure is Measured

For each GPU instance, monitor 3 signals:

Queue Backlog
Long queue → GPU struggling to keep up with arrival rate

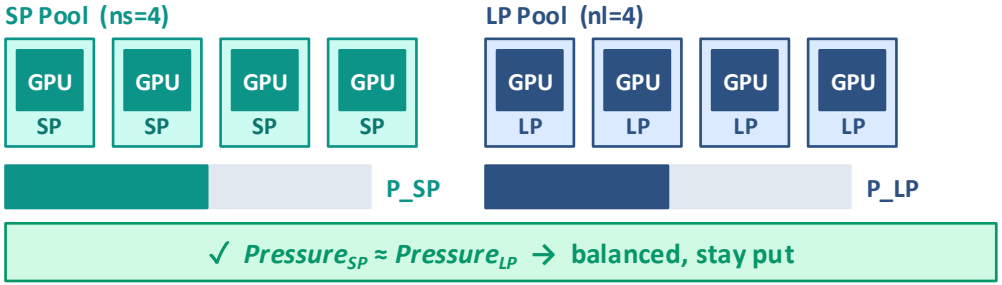
SLA Deviation
Requests close to / past their TTFT deadline

GPU Utilization
High utilization is good → reduces pressure score

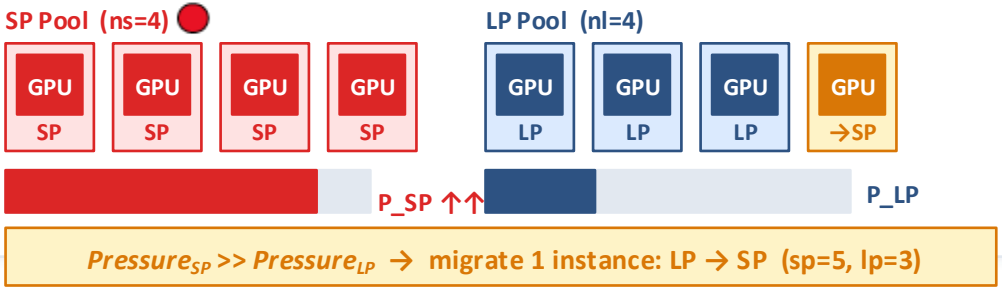
```
Pressure = Queue↑ + SLA miss↑ - GPU util↑
```

Pool Comparison → Migration Decision

Example A — Balanced, no migration needed:



Example B — SP overloaded, migrate one GPU:



Innovation 2: CUDA Graph Bucketization for Batched Short Prefills

Why not regular prefill use CUDA graph?

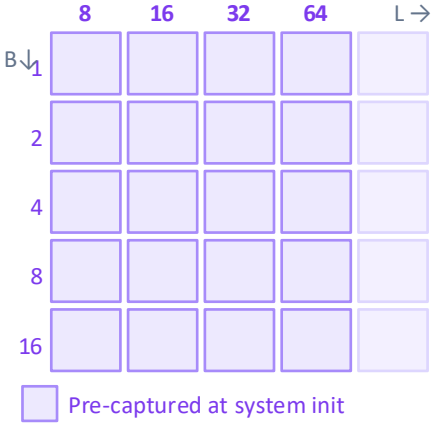
Prefill length varies every request → tensor shapes change → Attention kernel cannot be captured.

Why short prefill works:

Behaves like decode (memory-bound) → stable compact shapes → perfect for CUDA Graph.

Bucket Grid (L × B)

One CUDA Graph captured per cell (Allowed to dynamically change based on hit frequency):



init

① CAPTURE (one-time init)

1 Fill dummy tensors

Allocate input shape with dummy values

2 Full forward pass

Run prefill: GEMM → FlashAttn → FFN through all layers

3 Store by key (L, B)

graph_store[(32, 4)] = graph
Ready for instant replay

inference

② REPLAY (every request)

Incoming requests

e.g. 29 tok, 47 tok, 13 tok, 58 tok

Lookup nearest bucket

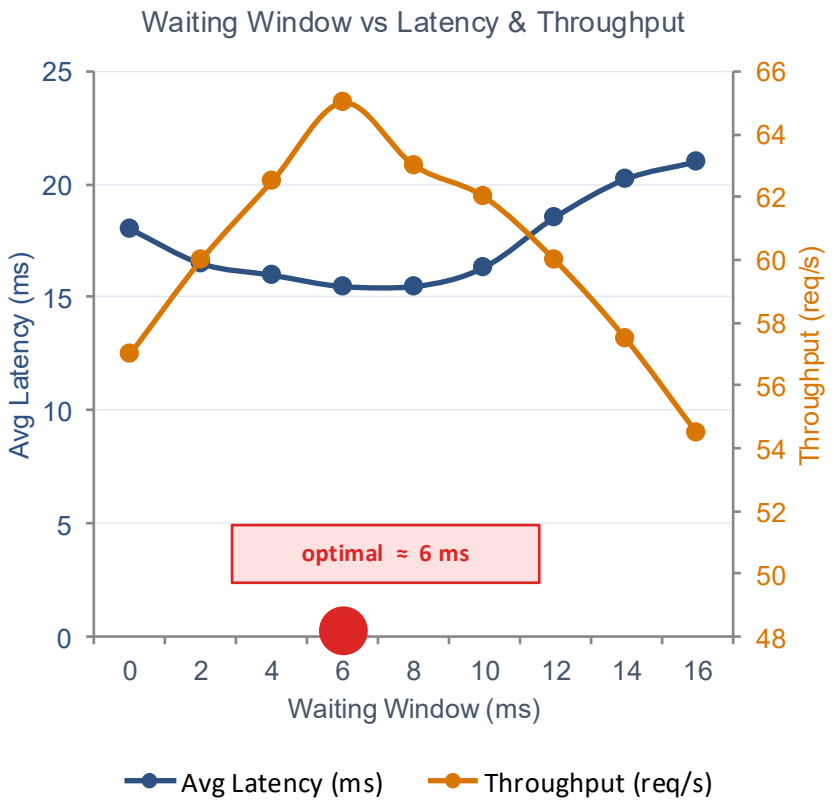
29→32, 47→64, 13→16, 58→64

Group by bucket shape

Memory first strategy:
batch (2×64) + batch (1×32) + batch (1×16)
minimal padding, length-homogeneous

Latency first strategy:
batch (4×64), same length

Innovation 3: Adaptive Wait-Depth (AWD) Scheduler



How AWD Works

Two adaptive thresholds, chosen every scheduling round:

W_{SLA} — SLA Safety Window

Latest moment we can still dispatch without any request missing its deadline

W_{GR} — Graph Fill Window

Expected time for enough requests to arrive and fill a CUDA Graph bucket

$$W = clip(min\{ W_{SLA} , W_{GR} \})$$

- Wait up to W, accumulate requests until batch depth D is reached
- If any SLA slack ≤ σ → dispatch immediately without waiting
- After each dispatch, W and D update from observed arrival rate

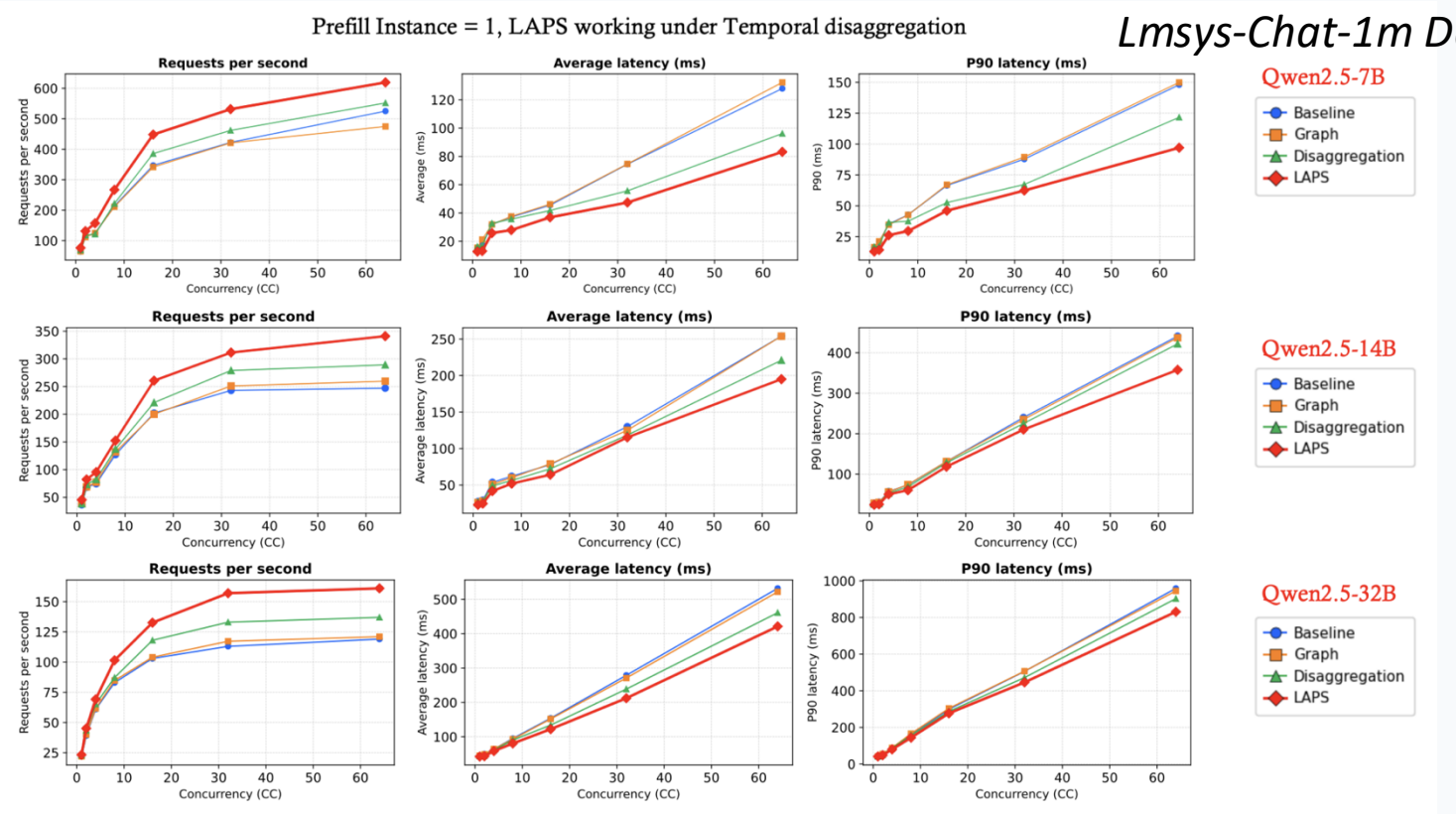
LAPS: Introducing the 4th Serving Mode

LLM serving systems support 3 existing modes. LAPS introduces a 4th:

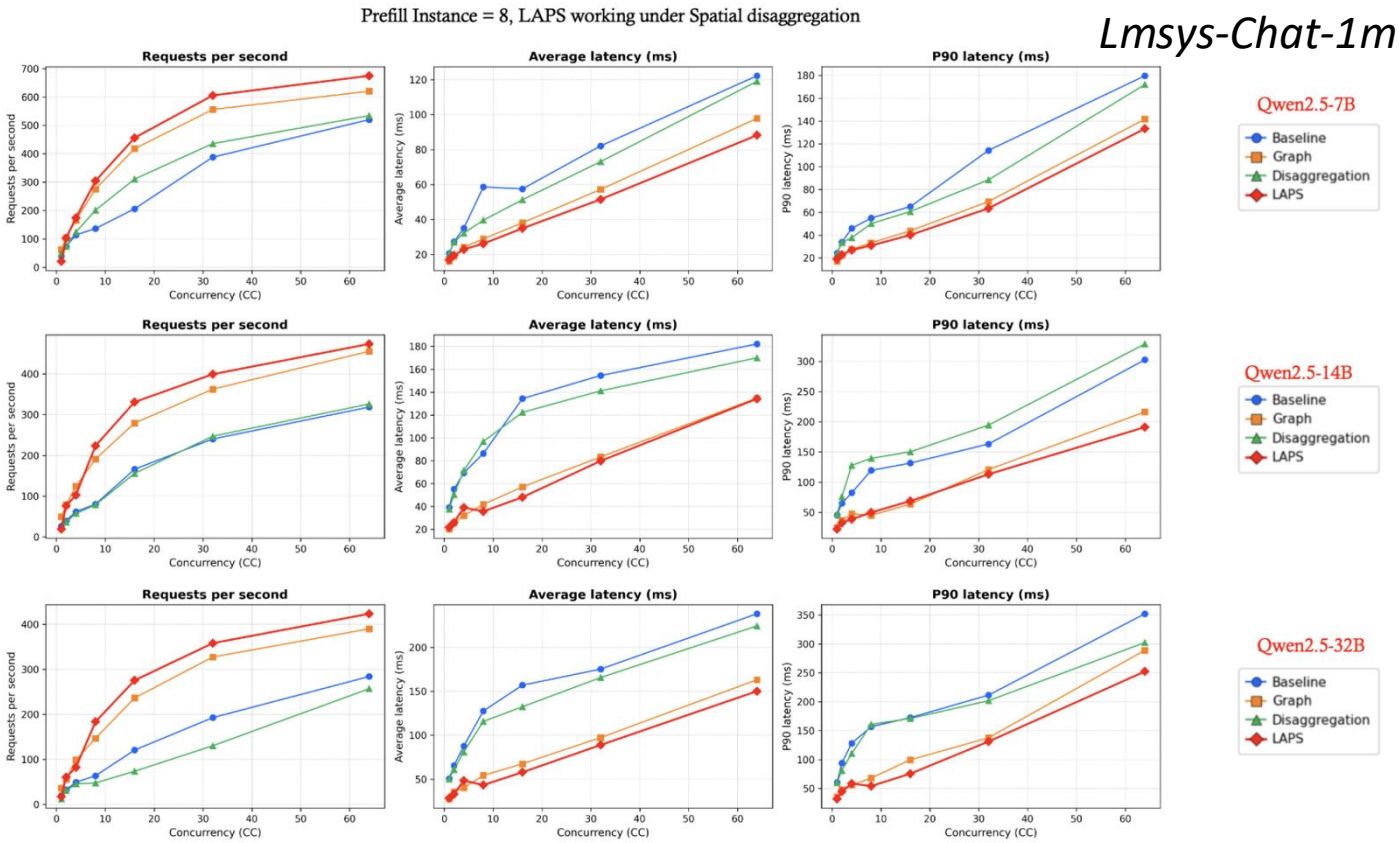
1	**Mix Mode** Decode requests inserted into prefill batches. No disaggregation.	Existing
2	**PD Temporal Disagg.** Prefill and decode run sequentially on the same instance.	Existing
3	**PD Spatial Disagg.** Prefill and decode run on separate instances. Current SOTA.	Existing
4	**Prefill Batch Disagg. ★** Separate long- and short-prefill inside the prefill stage. Temporal or spatial.	New!

★ LAPS is fully compatible with existing PD disaggregation — it operates within the prefill cluster, no interface change.

Results: Request Throughput — Single Prefill Instance (Temporal)



Results: Request Throughput — 8 Prefill Instance (Spatial)



Lmsys-Chat-1m Dataset

Discussion: Two Design Trade-offs in LAPS

D1 KV Cache Slot Waste from Bucket Padding

To use a CUDA Graph, each request is padded to the nearest bucket boundary. The KV cache is allocated for the full padded length — unused slots are wasted GPU memory.

KV Cache Allocation per Request:

req: 29 tok → bucket 32

3 slots wasted

req: 47 tok → bucket 64

17 slots wasted

req: 13 tok → bucket 16

3 slots wasted

useful token wasted KV slot

Mitigation: Memory-First vs Latency-First

Use 2×32 + 2×64 instead of 1×(4×64) → less waste, but 2 kernel launches.
Tunable: latency-critical → latency-first; memory-tight → memory-first.

D2 Bursty KV Cache Transfer under PD Disaggregation

AWD holds requests during the waiting window, then dispatches a full batch at once. All requests finish prefill simultaneously → their KV caches all transfer to decode at the same moment.

KV Transfer Pattern over Time:

Without LAPS (request-by-request):

→ steady, uniform KV transfers to decode instances

With LAPS (batched after waiting window W):

→ all KV caches transferred at once after window expires

Impact on Network:

✓ With InfiniBand: high bandwidth absorbs bursts — no new bottleneck, full benefit preserved.

✗ Without InfiniBand: burst saturates the network → new bottleneck. Larger W = bigger batch = higher peak bandwidth demand.

Thank You!

Wechat:

Linkin:

Questions?

Email: jianshushe@gmail.com