

Sarathi-Serve: Chunked Prefills & Stall-Free Batching

Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve
Agrawal, Kedia, Panwar, Mohan, Kwatra, Gulavani, Tumanov, Ramjee · Microsoft Research India + Georgia Tech · OSDI '24 · arXiv:2403.02310 · Day 2

Where this sits relative to Day 1. PagedAttention solved *memory* — it let you fit a batch of 128. Sarathi-Serve shows that under a real latency SLO you **cannot actually use that batch**, because the scheduler freezes every decode whenever a prefill arrives. This paper fixes the *scheduler*. Read together they're the two halves of one answer: memory management gets you capacity, scheduling gets you capacity *you can ship*.

Problem

Prefill and decode are opposite workloads sharing one GPU

	Prefill	Decode
Work per iter	All N prompt tokens, in parallel	1 token per request
Bottleneck	Compute-bound (low bandwidth util)	Memory-bound (low FLOPs util)
Batching helps?	No — throughput saturates at batch size 1	Yes — roughly linear in batch size
Latency metric	TTFT (time-to-first-token), once per request	TBT (time-between-tokens), every token

Put them in the same batch naively and one of them suffers. Every existing scheduler picked a side:

- Decode-prioritizing** (FasterTransformer, Triton — request-level batching): admit no new prefill until every running request finishes decoding. Great TBT. Terrible throughput — the batch drains as requests complete and you keep running at shrinking batch size, plus padding waste.
- Prefill-prioritizing** (Orca, vLLM — iteration-level batching): eagerly admit prefills the moment KV memory frees up, because a bigger batch makes the *next* decodes cheaper. Great throughput. But a prefill iteration over a 7,000-token prompt takes *seconds*, and every running request's token stream freezes for that entire time.

Generation stalls — the named failure

A **generation stall** is a P99 TBT spike caused by scheduling a prefill between two decode iterations of an in-flight request. On Yi-34B / 2×A100 with the arxiv-summarisation trace, vLLM stalls lasted **several seconds**. The user watching tokens stream sees the response hang.

The single most quotable result in the paper. They ran vLLM at max batch size 32, 64, and 128 under a strict TBT SLO. **Capacity was essentially identical across all three.** PagedAttention hands you a batch of 128; the prefill-prioritizing scheduler makes it unusable. Memory was never the binding constraint once you attach an SLO — *scheduling was*.

Second problem: pipeline bubbles that micro-batching doesn't fix

Cross-node deployments need pipeline parallelism (TP's two all-reduces per layer are ruinous over commodity ethernet — measured TBT for Falcon-180B TP-8 across nodes is **~2x worse** than TP4-PP2). Conventional wisdom says inference is forward-only, so micro-batching eliminates bubbles. False: bubbles come from **runtime variance between micro-batches**, and LLM iterations vary wildly.

- PB1** — consecutive micro-batches have different prefill token counts.
 - PB2** — a prefill micro-batch followed by a decode micro-batch (or vice versa).
 - PB3** — even decode-only batches differ, because attention cost scales with each request's accumulated context length.
- Falcon-180B: one 4K-token prompt **≈ 1150 ms**; a decode-only batch of 32 **≈ 200 ms**. Interleave them and you burn a **~950 ms bubble**.

Intuition

The free lunch: arithmetic intensity slack. Iteration time is $T = \max(T_{\text{math}}, T_{\text{mem}})$. A decode batch has $T_{\text{math}} \ll T_{\text{mem}}$ — you're waiting on weight fetches from HBM, and the tensor cores are idle. That idle compute is **free**. You can add prefill tokens to a decode batch and the iteration takes almost the same wall-clock time, right up until the batch crosses into the compute-bound regime.

Two supporting measurements make this concrete:

- Linear layers are >80% of runtime**, even at long sequence lengths where attention's quadratic term is biting. So reason about linear layers, not attention.
- Linear-layer time is **flat from ~128 to ~512 tokens** (dominated by fetching weights from HBM), then rises linearly. Theory says the knee is ~200 tokens on A100; in practice it's ~500-600 at higher TP degrees because of fixed overheads. **Tokens below the knee are nearly free.**

So: don't dump a whole 7,000-token prefill into a decode batch — that overshoots the knee by 10x. **Chop the prefill into chunks sized to reach the knee and stop.** The chunk size becomes a **token budget τ** , and τ is the knob that converts a

latency SLO into a scheduling parameter: cap tokens per iteration $\Rightarrow$ cap iteration time $\Rightarrow$ cap TBT. Crucially, iteration time becomes **independent of prompt length**.

The second-order win was almost accidental: if every iteration carries $\approx \tau$ tokens, every micro-batch takes $\approx$ the same time, so PB1/PB2/PB3 all collapse. **Uniform batches are the bubble fix**. That side effect is worth 5.6 $\times$ on Falcon-180B — larger than the primary result.

Solution

1. Chunked prefills

Split a prompt into near-equal compute-sized chunks and compute its prefill across several iterations. Chunk i 's attention reads the KV of chunks $1..i-1$, which is already in the paged KV cache — this is a correctness-preserving decomposition, not an approximation.

2. Stall-free batching (the actual contribution)

A strict priority order inside every iteration, capped by τ :

1. **All running decodes go in first.** 1 token each. They are never delayed by a prefill — this is the whole point.
2. **Then in-flight prefill chunks** from partially-completed prompts, into leftover budget.
3. **Then, and only then, new requests** — chunked to fit whatever budget remains.

Contrast with Orca, which also builds hybrid batches but admits the *entire* prefill, so a long prompt still blows the iteration up. Hybrid batching without chunking does not solve this.

3. Choosing the token budget τ — four competing forces

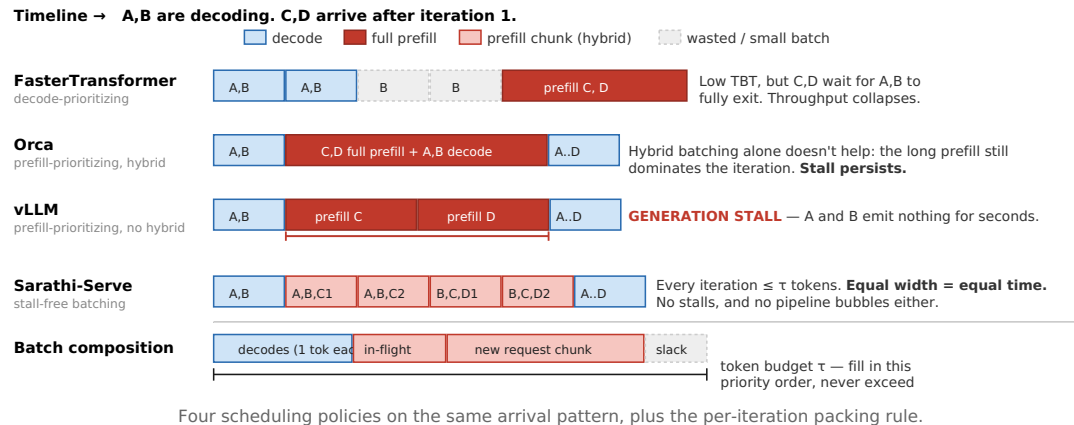
Force	Pushes τ	Detail
TBT SLO	Down	Fewer tokens $\Rightarrow$ shorter iteration $\Rightarrow$ lower P99 TBT.
Chunking overhead	Up	Chunk i re-reads KV of all prior chunks: split into N chunks and chunk 1's KV is read $N-1$ times. Extra HBM reads, same FLOPs. Measured $\leq$ ~25% at chunk 512, negligible at 2048.
Tile quantization	To multiples of tile size	GPUs tile matmuls across thread blocks. Dimensions not divisible by tile size waste whole blocks. Chunk 257 measured up to 32% slower than 256.
Pipeline bubbles	Down	Bigger chunks $\Rightarrow$ more inter-batch runtime variance $\Rightarrow$ more bubbles.

Resolved by **one-time offline profiling**: sweep iteration time vs token count, pick the largest tile-aligned τ that meets the SLO. The paper uses the Vidur simulator. In the evaluation: **$\tau = 512$ under strict SLO, $\tau = 2048$ under relaxed.**

4. Implementation

Built *on top of* vLLM — it inherits PagedAttention rather than replacing it. Added paged chunked-prefill support via FlashAttention-2 and FlashInfer kernels, plus pipeline parallelism and telemetry. NCCL for both TP and PP communication.

Solution Diagram



Code

The entire paper is ~25 lines of scheduler. Be able to write this cold.

```
def build_batch(running, queue, tau, kv):
    """One iteration of stall-free batching. tau = token budget from TBT SLO."""
    batch, n = [], 0

    # (1) ALL running decodes first. Never preempted by a prefill. This is the paper.
    for r in running:
        if r.prefill_done:
            batch.append((r, r.next_token)); n += 1      # 1 token each

    # (2) Then in-flight prefills – finish what you started before admitting more.
    for r in running:
        if not r.prefill_done:
            c = min(tau - n, r.prompt_len - r.prefilled)  # next chunk
            if c > 0:
                batch.append((r, r.prompt[r.prefilled : r.prefilled + c])); n += c

    # (3) Only now admit new work, into leftover budget.
    while queue and n < tau:
        r = queue.peek()
        if not kv.can_allocate(r): break                # PagedAttention still governs memory
        c = min(tau - n, r.prompt_len)
        if c == 0: break
        queue.pop()
        batch.append((r, r.prompt[:c])); n += c

    return batch    # n <= tau (+decodes) => iteration time bounded => TBT bounded
```

```
# --- Deriving tau offline. Do NOT hand-pick it; profile the step curve. ---
def compute_token_budget(tbt_slo_ms, model, parallel_cfg):
    best = 0
    # tile-aligned candidates ONLY: 256, 512, 768, 1024, 1536, 2048 ...
    # 257 can be ~32% slower than 256 due to tile quantization.
    for n in [256, 512, 768, 1024, 1536, 2048, 3072, 4096]:
        t = profile_iteration_time(model, parallel_cfg, n_tokens=n)
        if t <= tbt_slo_ms:
            best = n                                # take the LARGEST that fits
    return best

# Paper's values: tau = 512 (strict SLO), tau = 2048 (relaxed).
```

Note the ordering subtlety an interviewer may probe: decodes are packed unconditionally, *before* the budget check. So a batch of 600 running decodes can exceed $\tau = 512$ on its own. τ throttles **prefill admission**, not total batch size — the max-batch-size limit is a separate knob. Getting this right shows you read the algorithm, not the abstract.

Results

Setup	Capacity gain
Mistral-7B, 1×A100, sharegpt, strict SLO — vs vLLM	2.6×
Yi-34B, 2×A100 (TP2), sharegpt, strict SLO — vs vLLM / vs Orca	3.7× / 4.0×
LLaMA2-70B, 8×A40 (TP4-PP2) — vs Orca	6.31×
Falcon-180B, 2 nodes (TP4-PP2), end-to-end	5.6×
Falcon-180B strict SLO — vs vLLM TP8 / vs vLLM hybrid-parallel	4.3× / 3.6×
Chunking overhead on raw prefill time (Yi-34B, chunk 512 / 2048)	$\leq \sim 25\%$ / $\sim 0\%$

Ablation (Yi-34B, $\tau=1024$) — both halves are required:

Configuration	P50 TTFT (s)	P99 TBT (s)
Hybrid batching only ($\approx$ Orca)	0.53	0.68 ← stalls remain
Chunked prefills only	1.04 ← TTFT doubles	0.17
Sarathi-Serve (both)	0.76	0.14

Chunking alone hurts TTFT (a prompt now waits several iterations to finish prefilling). Hybrid batching alone hurts TBT (long prefills still dominate). Composed, each covers the other's regression.

Likely Interview Questions + Answers

Q1. Why does batching help decode throughput almost linearly but do nothing for prefill?

Decode fetches the full weight matrix from HBM to produce one token per request — the fetch cost is fixed and amortizes across the batch, so throughput scales until you hit the compute knee. Prefill already processes hundreds or thousands of tokens in parallel, so it is *already* compute-saturated at batch size 1; adding requests just queues more FLOPs. State it as arithmetic intensity: decode is far left of the roofline knee, prefill is already past it.

Q2. vLLM's PagedAttention lets you batch 128 requests. Why doesn't that translate into throughput?

Because capacity is defined under a latency SLO, and vLLM's prefill-prioritizing scheduler blows the P99 TBT budget long before it saturates the GPU. The paper measured vLLM at batch 32, 64, and 128 and got essentially the same capacity under a strict SLO. Memory management removed one bottleneck and exposed the next one — scheduling. Good answer to give whenever someone asks "we already added capacity, why is p99 flat?"

Q3. Define TTFT, TBT, and capacity. Which does chunked prefill trade away?

TTFT = arrival → first output token; measured once per request; dominated by prefill cost and queueing delay. TBT = interval between consecutive output tokens; measured on *every* token, so it's the one you take at P99. Capacity = max sustainable QPS while meeting a latency SLO (with bounded queueing — the paper caps median scheduling delay at 2s so "capacity" isn't gamed by unbounded queues). Chunked prefill trades **TTFT** for TBT: a prompt now takes several iterations to prefill. The ablation shows it in isolation (1.04s vs 0.53s), and hybrid batching claws most of it back (0.76s).

Q4. What exactly is the overhead of chunking, and why is it small?

Chunk *i*'s attention must read the KV of chunks 1..*i*−1. Split into *N* chunks and chunk 1's KV is re-read *N*−1 times — quadratic extra HBM traffic, identical FLOPs. It stays cheap because prefill attention remains compute-bound even at small chunk sizes, so the extra reads hide under the math. Measured: $\leq \sim 25\%$ on total prefill time at chunk 512, effectively zero at 2048. The honest caveat: this grows with prompt length, and they filtered traces at 8K–16K tokens, so don't extrapolate to 128K context.

Q5. How do you pick the token budget τ ?

Offline profiling, not intuition. Sweep iteration time against token count for that exact model + parallelism config, then take the **largest tile-aligned** τ whose iteration time fits the TBT SLO. Four forces: SLO pushes τ down; chunking overhead pushes it up; tile quantization forces it onto multiples of the matmul tile size; pipeline parallelism pushes it down to keep micro-batches uniform. The paper used $\tau=512$ strict, $\tau=2048$ relaxed, and used a simulator (Vidur) to search. τ is static in this work — making it adapt to live workload is explicitly left as future work, which is a fine thing to propose in an interview.

Q6. Why is a chunk size of 257 a bug?

Tile quantization. GPUs partition matmuls into fixed-size tiles across thread blocks; every block does the same amount of work whether or not it's full. A dimension one past a tile boundary allocates an entire extra row of thread blocks doing almost nothing. Measured up to **32% slower than 256**. Generalizes: any hand-tuned batch/sequence dimension should be snapped to a power of two or a multiple of the tile size.

Q7. Micro-batching is supposed to eliminate pipeline bubbles in a forward-only workload. Why doesn't it here?

Micro-batching hides bubbles only when micro-batches take *equal* time. LLM inference iterations don't: PB1 from differing prefill token counts, PB2 from a prefill stage followed by a decode stage, PB3 from decode-only batches whose attention cost depends on accumulated context length. Falcon-180B: 4K prompt $\approx 1150\text{ms}$ vs decode batch of 32 $\approx 200\text{ms}$ → $\sim 950\text{ms}$ bubble. Sarathi-Serve fixes it as a side effect: capping every iteration at τ tokens makes runtimes near-uniform. Worth $5.6\times$ end-to-end — the largest single number in the paper.

Q8. Chunked prefill vs. prefill/decode disaggregation (DistServe, Splitwise). When would you pick each?

Disaggregation runs prefill and decode on *separate replicas*, eliminating interference completely and giving the best possible TTFT (prefills run unchunked, at full efficiency). Its costs: you must ship the whole KV cache across the network on every phase transition, which needs high-bandwidth interconnect; and prefill replicas' GPU memory sits largely unused since only decode replicas hold KV. Chunked prefill keeps everything on one replica — no migration, no interconnect requirement, better memory utilization — at the price of slightly slower prefills and some TTFT. **Rule of thumb:** commodity networking or memory-constrained → chunked prefill; NVLink/InfiniBand-class fabric with hard TTFT SLOs and very long prompts → disaggregation. Note the paper explicitly declines to benchmark against disaggregation, so don't claim a winner.

Q9. Design: a chatbot at 100 QPS, median prompt 1.7K tokens, P99 TBT ≤ 100 ms. Walk me through it.

(1) **Size a replica:** KV bytes/token $\times$ context $\times$ target concurrency vs HBM minus weights — the Day-1 formula. (2) **Profile the step curve** for your model+TP config; find the largest tile-aligned token count whose iteration time is ≤ 100 ms. That is τ . (3) **Scheduler:** stall-free ordering — decodes, then in-flight chunks, then admissions. (4) **Parallelism:** TP within a node (NVLink); go PP only across nodes, and lower τ if you do, to keep micro-batches uniform. (5) **Capacity:** measured QPS per replica at that τ under the SLO → replica count → horizontal scale behind a router. (6) **Watch:** P99 TBT and P50 TTFT together — τ trades one for the other, so an alert on only one will let you regress the other silently. (7) **Layer on:** prefix caching for the system prompt, GQA/quantized KV to shrink the cache. Naming the two-metric monitoring trap is what makes this answer land.

Q10. What breaks if you set τ too small? Too large?

Too small: prefills fragment into many chunks $\rightarrow$ quadratic KV re-reads, kernel-launch overhead, batches sitting below the arithmetic-intensity knee so you waste compute, and TTFT climbs. Too large: iteration time exceeds the TBT SLO (generation stalls return), and under PP, inter-batch variance grows and bubbles come back. It's a genuine U-curve, which is exactly why the paper profiles rather than picks.

What's not covered

Deliberately deferred by the authors

- **No quantitative comparison against disaggregation.** They discuss SplitWise / DistServe / TetriInfer qualitatively and explicitly leave the head-to-head to future work. Any claim about which wins is yours, not theirs.
- **τ is static.** Set once by offline profiling; no adaptation to live prompt-length distribution or load. They name dynamic τ as future work — a strong thing to propose unprompted.

Scope limits worth naming

- **SLO is defined on P99 TBT only.** TTFT is reported but never constrained in the optimization — and TTFT is exactly what chunking degrades. A production system needs both in the objective.
- **Scheduling is still FCFS.** No priorities, no fairness across tenants, no preemption for head-of-line blocking. They point at Sheng et al. (fairness) and FastServe (preemptive) as complementary.
- **Long context is untested.** Traces filtered at 8K / 16K tokens. Chunking's re-read overhead grows with prompt length, so the $\leq 25\%$ figure should not be extrapolated to 128K+.
- **No interaction with prefix caching.** Chunk boundaries vs. cached-prefix boundaries is a real engineering problem the paper doesn't touch.
- **No multi-turn KV reuse.** Each conversation round is an independent request — the same gap as vLLM. (APIServe, cited here, adopts chunked prefills specifically to fix this.)
- **Hardware-specific knee.** The whole design rests on where the memory/compute crossover sits. That point moves with FLOPS:bandwidth ratio, so τ must be re-profiled per GPU generation. Evaluated only on A100/A40 — H100 and beyond shift it.

Orthogonal, untouched

- Memory management itself — this *assumes* PagedAttention underneath. KV exhaustion and preemption remain vLLM's problem.
- Speculative decoding (attacks the sequential-decode bottleneck directly, and interacts non-trivially with token budgets), quantization, MoE, LoRA multi-tenancy.
- Cluster level: no request routing, replica placement, or autoscaling. This is a single-replica scheduler.

Where to go next: DistServe / Splitwise (the disaggregation alternative — the natural Day 3, so you can argue both sides of Q8) $\rightarrow$ Orca (the iteration-level batching baseline both papers build on) $\rightarrow$ Vidur (simulation-driven config search, which is how τ actually gets chosen in practice) $\rightarrow$ FlashAttention-2 (the kernel making chunked paged prefill viable).