

Adaptive Scheduling for High-Concurrency AI Inference Systems

Saksham Rai

UC San Diego

¹Received: 27/05/2026; Accepted: 28/06/2026; Published: 11/07/2026

Abstract

High-concurrency serving of large language models (LLMs) must balance throughput against per-request latency. Batching raises accelerator utilization but can delay the first token, stall streaming output, and exhaust key-value (KV) cache memory when prompt and output lengths vary. This paper proposes a hierarchical adaptive scheduler and evaluates its replica-level mechanisms, namely deadline-ordered admission, prefill chunks sized from predicted iteration time, and memory-aware admission, against vLLM and Sarathi-Serve scheduling. The evaluation uses the Vidur simulator with A100 GPU profiles of Llama-2-7B and request lengths from two Azure production traces. Near saturation, the adaptive scheduler meets the latency service-level objective (SLO) for 82.1% of conversation requests at 5 requests/s, compared with 65.4% for Sarathi-Serve and 34.9% for vLLM, and raises goodput by 26% over Sarathi-Serve; on a coding workload at 6 requests/s, the corresponding figures are 77.5%, 51.2%, and 24.7%. Most of the gain comes from deadline ordering. The maximum load sustainable at 90% SLO attainment improves by only 5%, p99 time-to-first-token does not improve, and memory-aware admission removes recomputation but slightly lowers SLO attainment. Adaptive scheduling therefore helps most by deciding which requests to serve near saturation, not by raising peak capacity.

Keywords: *AI inference; adaptive scheduling; continuous batching; KV cache; latency; throughput; service-level objectives*

1 Introduction

LLM serving now mixes short interactive prompts, long-context jobs, and background work on the same accelerators. Output lengths are unknown at admission, and each request passes through two phases with different costs. Prefill processes the whole prompt in one compute-heavy pass; decode then generates one token per step and reads a growing key-value (KV) cache (Vaswani et al., 2017). Filling the largest possible batch raises tokens per second but can increase time-to-first-token (TTFT), time-between-tokens (TBT), and tail latency.

Earlier serving systems showed that batching, replication, and predictable execution make tail latency controllable (Crankshaw et al., 2017; Gujarati et al., 2020). Autoregressive models break the predictability assumption, so the scheduler must act at token granularity while managing scarce KV memory. This paper makes three contributions. It frames the scheduling objectives and the signals that expose them (Section 2) and reviews how current systems address them (Section 3). It proposes a hierarchical adaptive scheduler (Section 4). Finally, it implements the scheduler's replica-level mechanisms in the Vidur simulator (Agrawal et al., 2024b) and compares them with vLLM and Sarathi-Serve on production-derived workloads (Section 5).

¹ ¹ How to cite the article: Rai S.; Adaptive Scheduling for High-Concurrency AI Inference Systems; *International Journal of Advances in Engineering Research*, July 2026, Vol 32 Issue 1; pg: 15-23

2 Scheduling Requirements and Design Tensions

Scheduling quality has several dimensions (Table 1). Throughput counts completed tokens or requests per second. TTFT measures queueing plus prompt processing, TBT measures how smoothly tokens stream, and p95 or p99 latency reveals queue build-up. Cost per token is a constraint, not an afterthought. These goals conflict. Waiting to form larger batches improves efficiency but delays interactive requests. Admitting a long prompt raises utilization, but its prefill can stall running decodes. Aggressive preemption protects urgent work but wastes compute when evicted KV state must be rebuilt.

The scheduler also acts on partial information. Prompt length is known, but output length, tool calls, and cancellations are not. Arrival rates shift, requests may share prefixes, and multi-GPU deployments add communication and placement costs. Scheduling is therefore a closed-loop control problem, not a one-time choice of batch size.

Table 1. Principal scheduling objectives and observable signals

Objective	Operational signal	Primary control	Failure symptom
Responsiveness	TTFT, queue age, deadline slack	Admission, priority, prefill chunk	Slow first response
Streaming quality	TBT and decode lag	Decode reservation, stall limit	Bursty token delivery
Throughput	Tokens/s, batch occupancy	Continuous batching, token budget	Idle accelerator cycles
Memory safety	KV blocks, fragmentation, predicted growth	Admission, paging, preemption	OOM or thrashing
Fairness	Normalized service deficit	Weighted virtual time, aging	Starvation by long or premium jobs
Cost	GPU-seconds/token, energy, replica load	Routing, autoscaling, model placement	Low utilization or excess replicas

3 Evolution of High-Concurrency Inference Scheduling

Orca introduced iteration-level scheduling, which adds and removes sequences at every generation step instead of waiting for the whole batch to finish (Yu et al., 2022). vLLM’s PagedAttention stores the KV cache in fixed-size blocks, which reduces fragmentation and allows larger batches (Kwon et al., 2023). Together, these two systems made continuous batching the default for LLM serving.

Later work addressed interference between prefill and decode. Sarathi-Serve splits long prompts into chunks and runs them alongside decodes, so that no iteration stalls streaming output (Agrawal et al., 2024a); DeepSpeed-FastGen’s Dynamic SplitFuse follows the same idea (Holmes et al., 2024). DistServe instead places prefill and decode on separate GPUs, which removes interference at the cost of KV transfer and separate provisioning (Zhong et al., 2024).

At cluster level, Llumnix migrates KV state between instances to balance load (Sun et al., 2024), AlpaServe multiplexes several models over shared devices (Li et al., 2023), and SGLang reuses shared prompt prefixes across requests (Zheng et al., 2024). Sheng et al. (2024) define fair-share cost functions so that schedulers do not favor short requests, and recent queueing analyses derive throughput-optimal and KV-constrained policies under simplified assumptions (Jaillet et al., 2025; Li et al., 2025). Table 2 summarizes these mechanisms.

Table 2. Representative mechanisms in adaptive inference serving

System	Year	Scheduling contribution	Remaining limitation
Clockwork	2020	Predictable centralized admission and execution	Designed before autoregressive KV growth
Orca	2022	Iteration-level scheduling and selective batching	Limited memory-management flexibility
vLLM	2023	Paged KV cache and continuous batching	Baseline policies can struggle under mixed SLOs
AlpaServe	2023	Cross-model statistical multiplexing	Placement assumes workload estimates
Sarathi-Serve	2024	Chunked prefill and stall-free batching	Chunk tuning varies by model and hardware
DistServe	2024	Prefill-decode disaggregation	KV transfer and provisioning overhead
Llumnix	2024	Instance-level migration and load balancing	Migration consumes network bandwidth
SGLang	2024	Prefix-aware runtime scheduling	Benefits depend on workload structure
Queueing models	2025	Stability and KV-constrained online guarantees	Simplifying assumptions need production validation

4 Proposed Adaptive Scheduler

The scheduler is hierarchical. When a request arrives, an admission layer classifies its SLO, estimates its prompt cost, checks KV headroom, looks for shared prefixes, and assigns a virtual TTFT deadline. Instead of predicting exact output lengths, it keeps a running estimate of the expected decode length per model and tenant. A request whose predicted memory use would exceed a safety margin waits or is sent to another replica.

At each iteration, a token planner forms the batch under three budgets: compute tokens, KV blocks, and deadline risk. Running decodes always receive a slot, which bounds TBT. Prompts are processed in chunks whose size is chosen from the predicted iteration time, so the chunk grows when few decodes are running and shrinks as decode load rises. Waiting requests are ordered by deadline. Because a deadline is fixed at arrival, waiting requests age and cannot starve.

A device controller monitors memory bandwidth, utilization, and KV fragmentation, and chooses between swapping, recomputation, and migration based on measured cost. A cluster controller routes requests by predicted completion time rather than queue length, taking into account model placement, prefix locality, and KV-transfer time. When prefill and decode run on separate pools, the pools scale independently, and backpressure keeps prefill from creating more sequences than the decode pool can serve.

Adaptation should be conservative. Thresholds are updated every few seconds from exponentially weighted statistics, with bounded step sizes and hysteresis to avoid oscillation, while hard limits on memory, queue age, and TBT stay fixed. The scheduler should also log the reason for each decision, such as blocked memory, deadline promotion, cache-local routing, or aging, to support debugging.

5 Evaluation

5.1 Setup

The replica-level mechanisms of Section 4 were implemented in Vidur, an open-source LLM inference simulator that predicts operator execution times from GPU profiling data and reports per-request and per-iteration metrics (Agrawal et al., 2024b). The cluster-level controller was not evaluated. All runs simulate Llama-2-7B (Touvron et al., 2023) on one NVIDIA A100 GPU, using Vidur’s published profiles. Prompt and output lengths are sampled from two Azure production traces released with Splitwise (Patel et al., 2024). The conversation trace contains 19,366 requests with a median prompt of 1,020 tokens and a median output of 129 tokens. The coding trace contains 8,819 requests with a median prompt of 1,469 tokens and a median output of 13 tokens. Requests are capped at 4,096 tokens, the model’s context length, by truncating long prompts. Arrivals are Poisson, and each run serves 1,000 requests and is repeated with three random seeds.

Three schedulers are compared, all with 16-token KV blocks and a cap of 128 concurrent requests. vLLM uses its default policy, which runs waiting prompts in prefill-only iterations ahead of running decodes. Sarathi-Serve uses chunked prefill with a fixed budget of 512 tokens per iteration. The adaptive scheduler extends Sarathi-Serve with three mechanisms. First, waiting requests are served in order of a virtual TTFT deadline of 0.5 s plus 0.5 ms per prompt token. Second, the prefill chunk is the largest size between 512 and 2,048 tokens whose predicted iteration time, including all running decodes, stays below 90 ms. Third, a new prompt is admitted only if the free KV blocks also cover half of the expected decode growth of the running requests, where expected decode length is tracked with an exponentially weighted moving average. A request meets the SLO if its TTFT is at most 1 s and its mean TBT is at most 100 ms, and goodput is the number of requests per second that meet the SLO (Zhong et al., 2024). To fit the available compute, Vidur’s execution-time predictors were trained with a single random-forest configuration instead of a full grid search; their mean absolute percentage error on the profiling data was below 1% for the attention operators.

5.2 Results

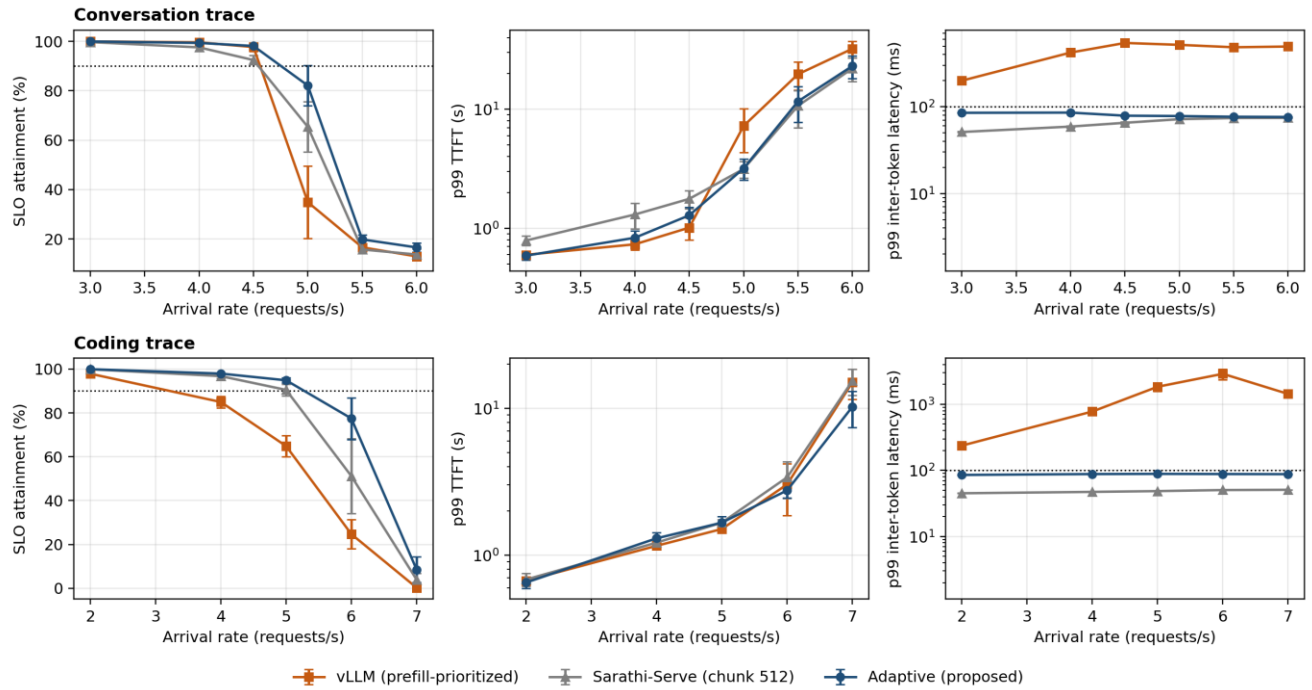


Fig. 1: SLO attainment, p99 TTFT, and p99 inter-token latency against arrival rate for the conversation trace (top) and the coding trace (bottom). Points show the mean over three seeds, and error bars show one standard deviation. Dotted lines mark 90% SLO attainment and the 100 ms TBT target.

Table 3. Latency, SLO attainment, and goodput near saturation (mean $\pm$ s.d. over three seeds)

Workload	Scheduler	TTFT p50 (ms)	TTFT p99 (s)	ITL p50 (ms)	ITL p99 (ms)	SLO attainment (%)	Goodput (req/s)	Through put (tokens/s)
Conversation, 5 req/s	vLLM	2176 $\pm$ 1377	7.24 $\pm$ 2.90	45 $\pm$ 0	517 $\pm$ 21	34.9 $\pm$ 14.7	1.69 $\pm$ 0.71	6718 $\pm$ 19
Conversation, 5 req/s	Sarathi-Serve	657 $\pm$ 149	3.15 $\pm$ 0.49	59 $\pm$ 4	72 $\pm$ 0	65.4 $\pm$ 10.2	3.25 $\pm$ 0.46	6891 $\pm$ 109
Conversation, 5 req/s	Adaptive (proposed)	360 $\pm$ 109	3.18 $\pm$ 0.65	58 $\pm$ 10	78 $\pm$ 0	82.1 $\pm$ 8.2	4.08 $\pm$ 0.34	6894 $\pm$ 110
Coding, 6 req/s	vLLM	771 $\pm$ 370	3.03 $\pm$ 1.17	29 $\pm$ 5	2885 $\pm$ 520	24.7 $\pm$ 6.6	1.50 $\pm$ 0.39	10586 $\pm$ 79
Coding, 6 req/s	Sarathi-Serve	1126 $\pm$ 527	3.38 $\pm$ 0.95	44 $\pm$ 0	50 $\pm$ 0	51.2 $\pm$ 17.1	3.15 $\pm$ 0.99	10737 $\pm$ 217
Coding, 6 req/s	Adaptive (proposed)	420 $\pm$ 94	2.76 $\pm$ 0.31	65 $\pm$ 1	88 $\pm$ 0	77.5 $\pm$ 9.5	4.80 $\pm$ 0.50	10765 $\pm$ 262

Note. ITL is inter-token latency, measured per generated token and including stalls. SLO: TTFT ≤ 1 s and mean TBT ≤ 100 ms per request. Simulated with Vidur (Llama-2-7B, one A100 GPU).

Prefill stalls dominate vLLM's tail latency. Under vLLM, a prompt that arrives while decodes are running is processed in a prefill-only iteration, so every running request waits for it. At the loads in Table 3, vLLM's p99 inter-token latency (ITL) is 517 ms on the conversation trace and 2.9 s on the coding trace, against 50–88 ms for the two chunked schedulers. Fig. 2 shows the mechanism: within a 10-s window, vLLM's prefill-only iterations last up to 345 ms, while every Sarathi-Serve and adaptive iteration mixes prefill and decode tokens and stays below 80 ms.

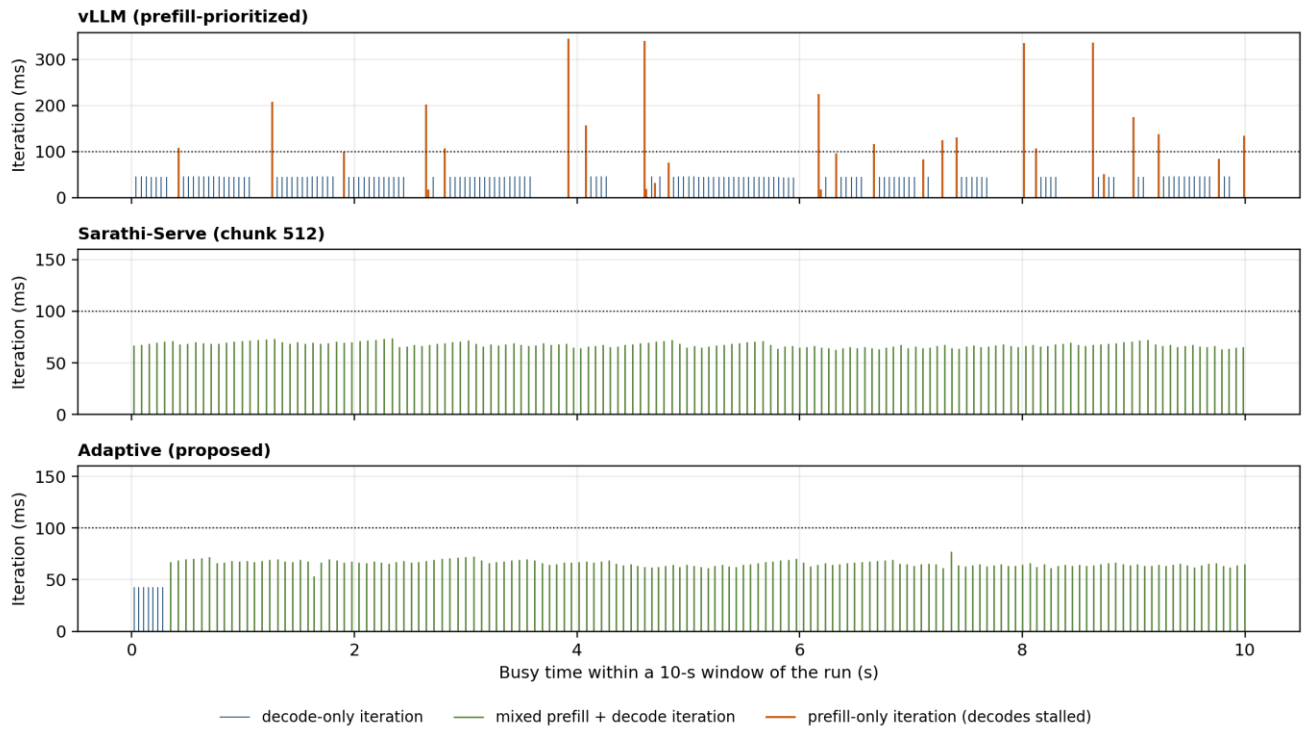


Fig. 2: Iteration-level trace on the conversation trace at 5 requests/s (seed 0). Each vertical line is one iteration, and its height is the iteration's execution time. Orange lines are prefill-only iterations during which running decodes stall; the dotted line marks the 100 ms TBT target.

Near saturation, the adaptive scheduler raises SLO attainment and goodput. At 5 requests/s on the conversation trace, it meets the SLO for 82.1% of requests, compared with 65.4% for Sarathi-Serve and 34.9% for vLLM, and its goodput is 4.08 requests/s against 3.25 and 1.69. On the coding trace at 6 requests/s, attainment is 77.5%, 51.2%, and 24.7%. Token throughput is almost identical for the two chunked schedulers (6,894 and 6,891 tokens/s), so the gain comes from serving the right requests first, not from doing more work. Median TTFT falls from 657 ms to 360 ms on the conversation trace and from 1,126 ms to 420 ms on the coding trace.

The gains are limited to a narrow load range. Interpolating Fig. 1, the highest arrival rate at which attainment stays at or above 90% rises from 4.54 to 4.75 requests/s on the conversation trace (+4.6%) and from 5.01 to 5.28 requests/s on the coding trace (+5.3%). The p99 TTFT does not improve (3.18 s against 3.15 s at 5 requests/s), because deadline ordering favors short prompts while long prompts still wait. The adaptive scheduler's p99 ITL is 6–38 ms higher than Sarathi-Serve's, because it uses larger chunks when decode load is low, but it stays below the 100 ms target. Beyond saturation, all three schedulers collapse below 20% attainment on the conversation trace at 5.5 requests/s; no replica-level policy prevents queue growth once arrivals exceed capacity, which is why admission control and routing at cluster level are needed.

5.3 Ablation and Memory Pressure

Table 4. Ablation of the adaptive scheduler (conversation trace, 5 requests/s)

Variant	SLO attainment (%)	Goodput (req/s)	TTFT p99 (s)	ITL p99 (ms)
Adaptive (all mechanisms)	82.1 ± 8.2	4.08 ± 0.34	3.18 ± 0.65	78
– Deadline ordering (FIFO admission)	71.9 ± 10.9	3.57 ± 0.48	2.90 ± 0.38	78

Variant	SLO attainment (%)	Goodput (req/s)	TTFT p99 (s)	ITL p99 (ms)
– Adaptive chunk (fixed 512 tokens)	77.3 $\pm$ 8.4	3.84 $\pm$ 0.36	3.44 $\pm$ 0.47	71
– Memory-aware admission	84.3 $\pm$ 6.6	4.19 $\pm$ 0.26	3.03 $\pm$ 0.58	78
Sarathi-Serve (reference)	65.4 $\pm$ 10.2	3.25 $\pm$ 0.46	3.15 $\pm$ 0.49	72

Note. Each variant removes one mechanism from the full scheduler. Mean $\pm$ s.d. over three seeds.

Removing deadline ordering lowers attainment by 10.2 points, and fixing the chunk at 512 tokens lowers it by 4.8 points; these two mechanisms account for the gain over Sarathi-Serve (Table 4). Memory-aware admission brings no benefit when KV memory is plentiful: without it, attainment is slightly higher (84.3%), because the reservation delays some admissions without preventing any restarts.

Table 5. Memory pressure (conversation trace, 2.5 requests/s, KV cache limited to 24,000 tokens)

Scheduler	SLO attainment (%)	Restarts per 1,000 requests	TTFT p99 (s)	ITL p99 (ms)
vLLM	92.4 $\pm$ 2.9	7.3 $\pm$ 4.6	2.52 $\pm$ 0.63	142
Sarathi-Serve	96.2 $\pm$ 0.8	4.7 $\pm$ 2.9	1.85 $\pm$ 0.23	48
Adaptive (all mechanisms)	95.1 $\pm$ 1.7	0.0 $\pm$ 0.0	2.45 $\pm$ 0.75	84
Adaptive without memory-aware admission	96.5 $\pm$ 1.7	6.3 $\pm$ 4.0	2.03 $\pm$ 0.62	84

Note. A restart discards a preempted request's KV cache and recomputes its prefill. The KV cache is limited to 1,500 blocks of 16 tokens.

To test memory-aware admission where it should matter, the KV cache was limited to 1,500 blocks (24,000 tokens) at 2.5 requests/s (Table 5). Admission control then eliminates recomputation restarts, from 6.3 to 0 per 1,000 requests, but lowers attainment from 96.5% to 95.1%, because waiting for memory delays TTFT more than the rare restarts cost. In this regime, recomputation is cheaper than conservative reservation. Memory-aware admission is likely to pay off only when restarts are frequent or recomputation is expensive, as with long contexts.

5.4 Limitations

The results come from simulation, not a deployed engine. Vidur models execution from GPU operator profiles, and its authors report less than 9% error against real deployments (Agrawal et al., 2024b), but it omits effects such as network jitter and kernel-launch variability. Only one model and one GPU type were tested, arrivals are Poisson rather than bursty, the cluster controller and prefill–decode disaggregation were not evaluated, and scheduler parameters were not tuned per workload. Validation on a live vLLM deployment with bursty production arrivals is the natural next step.

6 Discussion

No single scheduler dominates. Disaggregation removes interference between phases but adds transfer costs that hurt small clusters and slow networks. Shortest-job-first lowers mean latency but can starve long requests and relies on uncertain predictions. Strict reservations protect latency-critical traffic but waste capacity when admission is pessimistic, as the memory-pressure experiment shows. Prefix-aware routing improves cache reuse but can create hotspots. Adapting across mechanisms is therefore likely to outperform committing to any one of them.

The main open problem is coordination across timescales. Token scheduling acts in milliseconds, migration in tens to hundreds of milliseconds, and autoscaling in seconds to minutes. If a local scheduler evicts state that the cluster controller has just migrated, the system thrashes; if autoscaling reacts to queues caused by a burst of long prompts, it

can amplify them. Future designs need shared state, explicit prediction uncertainty, and safety limits. Heterogeneous accelerators, speculative decoding, quantization, and agent workloads that pause for external tools will further change arrival and service-time distributions.

7 Conclusion

Scheduling high-concurrency LLM inference is a dynamic resource-allocation problem. Continuous batching and paged KV memory are necessary but do not by themselves protect tail latency, streaming quality, fairness, or cost under mixed load. In simulation with production-derived workloads, an adaptive replica scheduler that orders admission by deadline and sizes prefill chunks from predicted iteration time raised SLO attainment near saturation by 17–26 points over Sarathi-Serve and by 47–53 points over vLLM, without changing throughput. The maximum sustainable load improved by only about 5%, and conservative memory admission cost slightly more than it saved. Schedulers should therefore be judged by goodput and SLO attainment under overload, not by peak tokens per second.

References

- [1]. Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., & Ramjee, R. (2024a). Taming the throughput-latency tradeoff in LLM inference with Sarathi-Serve. Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.
- [2]. Agrawal, A., Kedia, N., Mohan, J., Panwar, A., Kwatra, N., Gulavani, B. S., Ramjee, R., & Tumanov, A. (2024b). Vidur: A large-scale simulation framework for LLM inference. Proceedings of Machine Learning and Systems, 6.
- [3]. Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, 613–627.
- [4]. Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., & Mace, J. (2020). Serving DNNs like Clockwork: Performance predictability from the bottom up. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 443–462.
- [5]. Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., & He, Y. (2024). DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference. arXiv:2401.08671.
- [6]. Jaillet, P., Jiang, J., Podimata, C., & Zhou, Z. (2025). Online scheduling for LLM inference with KV cache constraints. arXiv:2502.07115.
- [7]. Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. Proceedings of the 29th ACM Symposium on Operating Systems Principles, 611–626.
- [8]. Li, Y., Dai, J., & Peng, T. (2025). Throughput-optimal scheduling algorithms for LLM inference and AI agents. arXiv:2504.07347.
- [9]. Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation, 663–679.
- [10]. Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., & Bianchini, R. (2024). Splitwise: Efficient generative LLM inference using phase splitting. Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture, 118–132.
- [11]. Sheng, Y., Cao, S., Li, D., Hooper, C., Lee, N., Yang, S., Chou, C., Zhu, B., Zheng, L., Keutzer, K., Gonzalez, J. E., & Stoica, I. (2024). Fairness in serving large language models. Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.
- [12]. Sun, B., Huang, Z., Zhao, H., Xiao, W., Zhang, X., Li, Y., & Lin, W. (2024). Llumnix: Dynamic scheduling for large language model serving. Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

- [13]. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., et al. (2023). Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288.
- [14]. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.
- [15]. Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., & Chun, B.-G. (2022). Orca: A distributed serving system for transformer-based generative models. *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, 521–538.
- [16]. Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C. W., & Sheng, Y. (2024). SGLang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37.
- [17]. Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., & Zhang, H. (2024). DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*.