SLOs-Serve: Optimized Serving of Multi-SLO LLMs

Siyuan Chen*
siyuanc3@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Zhipeng Jia
zhipengjia@google.com
Google
Seattle, WA, USA

Samira Khan
samirakhan@google.com
Google
New York, NY, USA

Arvind Krishnamurthy
arvindkrish@google.com
Google
Seattle, WA, USA

Phillip B. Gibbons
pgibbons@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

arXiv:2504.08784v1 [cs.DC] 5 Apr 2025

Abstract

This paper introduces SLOs-Serve, a system designed for serving multi-stage large language model (LLM) requests with application- and stage-specific service level objectives (SLOs). The key idea behind SLOs-Serve is to customize the allocation of tokens to meet these SLO requirements. SLOs-Serve uses a multi-SLO dynamic programming-based algorithm to continuously optimize token allocations under SLO constraints by exploring the full design space of chunked prefill and (optional) speculative decoding. Leveraging this resource planning algorithm, SLOs-Serve effectively supports multi-SLOs and multi-replica serving with dynamic request routing while being resilient to bursty arrivals. Our evaluation across 6 LLM application scenarios (including summarization, coding, chatbot, tool calling, and reasoning) demonstrates that SLOs-Serve improves per-GPU serving capacity by 2.2x on average compared to prior state-of-the-art systems.

1 Introduction

Large Language Models (LLMs) are becoming increasingly popular. Current LLM applications, such as interactive chatbot [22], coding assistant [18], document summarizer [24], often feature *multi-stage* processing: A *prefill* stage processes the input text, and a *decode* stage generates tokens one by one. More recent *Reasoning LLMs* [9] introduce a new thinking stage between prefill and decode for solving highly complex problems. And, when moving to agentic scenarios where LLMs are given access to a list of tools to solve real-world problems [29], a request is returned after rounds of internal processing stages between the LLM and the tools.

An LLM serving system, hosted on shared cloud servers, delivers online responses to LLM requests under service level objectives (SLOs). Meeting SLOs plays a key role in ensuring a positive user experience. As summarized in Table 1, different stages in LLM serving often demand stage-specific SLOs for the best user experience. For example, in prefill-decode-based applications, document summarization demands high prefill throughput to digest lengthy documents, chatbot demands smooth decoding flow to match a human's reading speed, and coding assistant demands lower per-token decode latency for code generation. What's more, to respond to the user sooner, reasoning models may want to squeeze the time for the thinking stage, and an LLM-with-tools application may want to minimize the latency back and forth between the LLM and the tools.

Supporting application-specific SLOs across multi-stage processings presents unique challenges in LLM serving due to complex resource sharing across requests and stages. State-of-the-art (SOTA) serving systems [2, 13, 14, 25, 34, 41] adopt

Table 1. Multi-stage LLM applications with diverse SLOs.

Application	Stages	SLOs
Summarization	• Long, fast prompt processing • Short reading-speed response	Tight on prefill Loose on decode
Coder	• Normal length/speed prompt • Long, fast response	Loose on prefill Tight on decode
ChatBot	• Normal length/speed prompt • Reading-speed response	Loose on prefill Loose on decode
LLM with Tools	• Normal length/speed prompt • Fast toolCall-toolResponse loop • Reading-speed response	Tight on prefill Tight on pref./dec. Loose on decode
Reasoning LLM	• Normal length/speed prompt • Long fast thinking • Reading-speed response	Tight on prefill Tight on decode Loose on decode

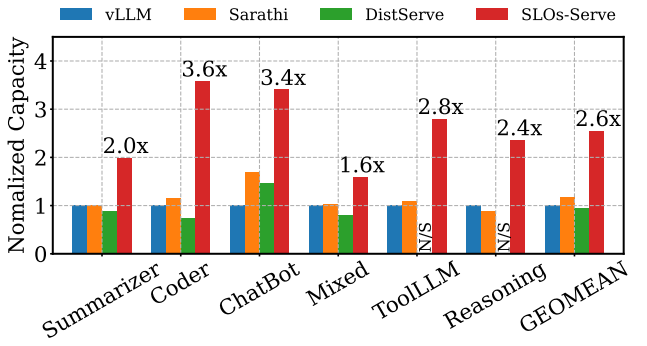


Figure 1. Serving Capacity comparison for LLM applications with heterogeneous SLOs, on a server with 4 A100s (experimental details in Tables 2 and 4 of §6). N/S: not supported.

*Part of this work was done during an internship at Google.

continuous batching [37] to maximize the serving throughput by forming batches comprising of *tokens* from different stages and requests. Moreover, LLM serving systems use either co-located or disaggregated approaches to map processing stages to GPUs. Both approaches exhibit unique challenges for the scheduler to meet per-stage SLOs (§2.3). Co-location runs all stages on the same GPUs, with schedulers like vLLM [14] and Sarathi-Serve [2] prioritizing different stages in isolation. This greedy approach causes significant SLO violations, especially during traffic spikes, as illustrated later in Fig. 3. The disaggregation approach [12, 13, 25, 28, 41], on the other hand, separates stages onto different devices with customized configurations. But the approach struggles when load distribution changes dynamically due to varying input/output lengths or shifts in application domains (as illustrated later in Fig. 4).

In this paper, we propose *SLOs-Serve*, an LLM serving system designed to handle LLM applications with per-stage or per-request SLOs, overcoming all the above challenges. SLOs-Serve introduces an SLO-optimized scheduling algorithm tailored for LLM requests (§3). The key idea is to customize token allocation to meet stage- and request-specific SLO requirements. Unlike prior schedulers that greedily attain the SLO of a single stage, SLOs-Serve's scheduler identifies a set of requests with attainable SLOs and, for that set, guarantees to satisfy every stage's SLO until completion. Specifically, SLOs-Serve designs a multi-SLO dynamic programming-based algorithm (§3.2.1) to continuously optimize token allocations in batches under stage- and request-dependent SLO constraints by exploring the full design space of chunked prefill and dynamic batch-size tuning (§3.2.2). Additionally, we propose *SLO-adaptive speculative decoding* (§3.2.3), which customizes the speculation length in speculative decoding to meet stage- and request-dependent SLOs. The accuracy of our optimization depends on a profiling-based characterization of the target GPU's capabilities on LLM workloads, which varies across GPU families (e.g., A100s vs. H100s).

Building on top of the scheduling algorithm, we prototype SLOs-Serve, a distributed LLM serving system that supports multi-SLOs and multi-replica serving, while being resilient to bursty arrivals. During temporary high request loads, SLOs-Serve delays requests with unattainable SLOs to secure the SLO attainment of the rest (§4.1), whereas existing schedulers' greedy approaches results in a cascading effect where every request fails to meet its SLO. In multi-replica serving, our system runs a centralized scheduler that virtualizes every replica, and proactively routes requests with unattainable SLOs at a replica to another with minimal scheduling overhead (§4.2). In this way, SLOs-Serve implements a form of *soft admission control*, in which "admitted" requests are guaranteed to meet their multi-stage SLOs and other requests are serviced either best effort (in single GPU settings) or routed to a different GPU for servicing (in multi-GPU settings).

In evaluation, we benchmark SLOs-Serve against state-of-the-art (SOTA) serving systems (vLLM, Sarathi-Serve, DistServe) on 6 LLM application scenarios with customized SLOs. As displayed in Fig. 1, SLOs-Serve improves the serving capacity (defined as the maximum request load per GPU while maintaining 90% SLO attainment) by a geo-mean of 2.2x compared to the best of Sarathi-Serve and vLLM and 2.4x compared to DistServe. In multi-replica serving, we observed that under bursty arrivals SLOs-Serve is able to serve up to 6.2x capacity with 4-replica serving compared with 1-replica serving. The linear scaling underscores SLOs-Serve scheduling algorithm's effectiveness in load balancing.

In summary, this paper makes the following contributions:

- We identify the challenges in multi-SLO LLM serving under fine-grained resource sharing and propose a multi-SLO dynamic programming-based scheduling algorithm to resolve the challenges. Our scheduler explores the full design space of chunked prefill, speculative decoding, and dynamic batch-size tuning, and leverages soft admission control.
- We build our design into SLOs-Serve, an LLM serving system that supports multi-SLOs, multi-replica serving, and is more robust to bursty arrivals than SOTA systems.
- Comprehensive evaluations across 6 application scenarios, including the first comparison study on modern tool-based and reasoning LLMs, show that SLOs-Serve is able to improve SOTA serving system's capacity by 2.2x on average.

2 Background and Motivation

In this section, we first summarize the current trend of LLM applications. We then articulate scheduling challenges introduced by emerging LLM applications while identifying pitfalls of current SOTA LLM schedulers.

2.1 LLM Applications and Serving

Multi-Stage LLM Applications. Emerging LLM applications are evolving beyond the traditional two-stage model (prefill and decode) toward multi-stage processing. While classic applications like chatbots and summarizers follow this basic pattern, reasoning models add a thinking stage for problem-solving. Agentic applications demonstrate even greater complexity through iterative tool interaction loops. This evolution necessitates more sophisticated resource management to handle diverse execution patterns. Table 1 summarizes characteristics of five multi-stage LLM applications.

Multi-SLOs Serving. Complex LLM processing can be modularized into prefill and decode phases with distinct service level objectives (SLOs). The prefill stage (initial prompt processing, reasoning, tool interaction) is measured by Time-To-First-Token (TTFT), which impacts user wait time. The decode stage (token generation) is measured by Time-Per-Output-Token (TPOT), affecting response delivery speed.

Stage-dependent SLO enforcement improves user experience. For reasoning models, prioritizing low TPOT in the

thinking stage minimizes perceived latency, while the decode stage can tolerate higher TPOT aligned with human reading speeds. Similarly, agentic applications benefit from low TTFT and low TPOT during tool interactions and reading-speed TPOT for final responses. This granular control enables tailored optimization across different stages.

Our goal is to maximize *serving capacity*, the maximum request load each GPU can process while maintaining a target SLO attainment rate (e.g., 90% attainment).

2.2 LLM Serving Optimizations

Continuous batching [37] improves throughput by combining processing stages from different requests. Since individual stages often underutilize hardware, batching increases simultaneous request handling with minimal latency overhead. Batches contain tokens from various stages and requests—prefill contributes prompt-length tokens, while decode adds one token at a time. *Token batch size* determines the throughput-latency tradeoff, with larger batches increasing throughput at the cost of latency, as shown in Fig. 2.

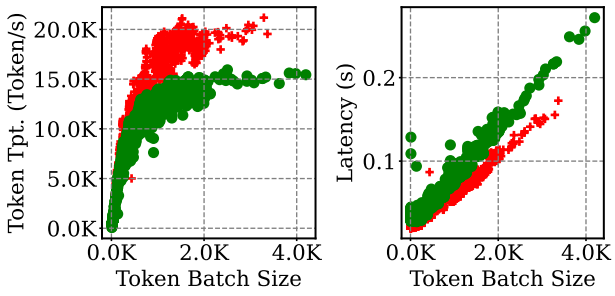


Figure 2. Throughput-latency trade-off for batching. Each data point–Green for the OPT-7B model [39] on A100, Red for the OPT-13B model [39] on H100–is a batch executed in SLOs-Serve's scheduling with both prefill and decode tokens.

Batch composition is diversified through:

- *Chunked prefill* [2] breaks long prefill stages into smaller chunks, preventing decode stages from stalling. Instead of using fixed maximum token sizes that limit throughput, batch sizes are dynamically adjusted based on system load (§3.2.2).
- *Speculative decoding* [15] uses a smaller "draft" model which speculates multiple token steps. Drafted tokens are verified in a single batch by the main model. The verification step processes multiple tokens simultaneously, creating new batching opportunities. While not always beneficial due to potential errors and overhead, adapting speculation lengths to SLOs can surprisingly improve throughput (§3.2.3).

2.3 Challenges

Supporting application-specific SLOs for multi-stage processing in LLM requests presents complex challenges due to the resource contention and diversified optimizations within continuous batching. We re-visit existing scheduling approaches and understand how they fail to meet the challenges.

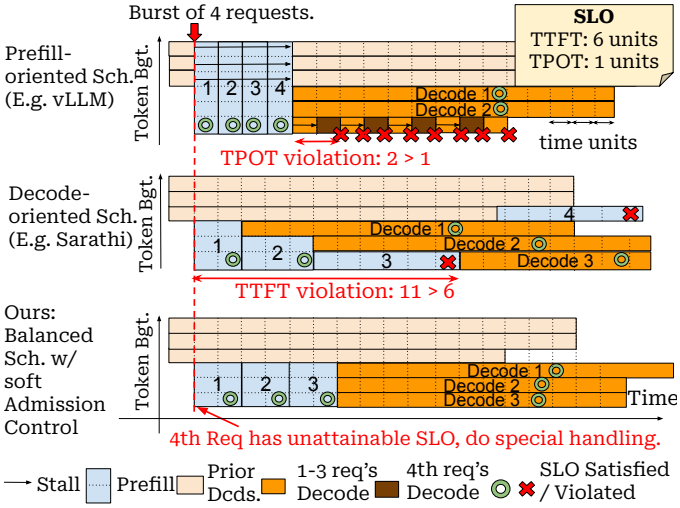


Figure 3. Comparison between different co-located scheduling approaches.

Co-located Scheduling. Current *co-located* schedulers (all GPUs run a mix of stage types) suffer from a fundamental flaw: they prioritize attaining SLOs at individual stages in isolation. For instance, vLLM [14] employs a *prefill-oriented scheduling* approach, which eagerly executes each request's prefill stage to minimize TTFT. But this strategy often leads to widespread TPOT violations due to the low priority assigned to decode processing. Conversely, Sarathi-serve attempts to mitigate this issue by breaking the prefill process into smaller chunks, but its *decode-oriented scheduling* prioritizes filling batches with decode tokens, resulting in TTFT violations.

To illustrate these shortcomings, consider the scenario in Fig. 3 where the system can process six tokens per time unit, and a burst of four requests arrives, each with six prefill tokens. At the time of arrival, the system has three ongoing decodes. The SLOs are set at six time units for TTFT and one time unit for TPOT. In a prefill-oriented scheduling scheme, the scheduler preempts ongoing decoding requests to execute incoming prefill requests, causing decoding stalls and subsequent TPOT violations. As the number of decoding requests overwhelms the system's processing capacity, some requests are guaranteed to miss their TPOT SLOs. In our example, requests three and four fail to meet their TPOT requirements. In contrast, a decode-oriented scheduler prioritizes filling available resources with decode tokens, leading to prolonged prefill stages and severe TTFT violations. E.g., when the first request's prefill completes, the scheduler allocates a token budget per time unit for decoding, delaying the second request's prefill. This delay propagates, causing

requests three and four to miss their TTFT SLOs, with request three's prefill completion delayed to the 11th time slot (TTFT violation) and request four's delayed even further.

The greedy nature of these scheduling approaches motivates our design of balanced scheduling with soft admission control. As shown later, our scheduler allocates tokens by taking into account the SLOs across stages (§3) and employs special handling for requests whose SLOs are inherently unattainable (§4). In the Fig.3 example, we successfully attain the SLOs for all existing requests, as well as three out of the four new requests. Thus, our multi-SLO attainment rate is higher than the prior approaches. Note that in general, some form of SLO-aware request prioritization (e.g., soft admission control) is required to prevent cascading-lateness effects and to maximize attainment rates (and serving capacity).

Disaggregated Scheduling. In *disaggregated* scheduling [25, 41], different stage types are separated onto different (groups of) GPUs. In this way, one can specialize for every stage's SLO by adjusting the hardware configurations [41], and balance workloads across stages by adjusting the GPUs allocated to every stage.

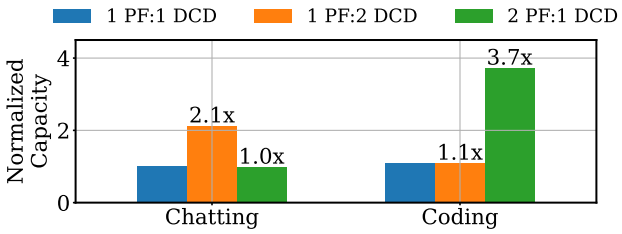


Figure 4. Capacity in DistServe with different prefill (PF), decode (DCD) device ratios when serving the OPT-13B model with H100 GPUs. The capacity is normalized to 1 PF:1 DCD.

However, the flexibility of disaggregated scheduling is limited by the fact that optimal hardware configurations depend on the stage's loads, which are highly dynamic in our setup. As illustrated in Figure 4, decode-heavy applications such as ChatBot require a higher allocation of decoding devices (1 decode: 2 prefills), whereas prefill-heavy workloads like Coder benefit from more prefill device allocation (2 prefills: 1 decode). Consequently, a single static allocation strategy is ineffective for serving multiple applications, as well as attaining request-dependent SLOs.

In fact, to achieve optimal serving capacity, the ideal prefill-to-decode device ratio in disaggregated scheduling can be expressed as: $\frac{n_{prefill}}{n_{decode}} = \frac{(1-\frac{C}{TPOT})E[prefillLength]}{E[decodeLength]}$, which is dependent on both the SLO requirement and the stage loads. See Appendix A for detailed discussions.

3 SLOs-Serve Scheduler

In this section, we introduce SLOs-Serve's scheduler that addresses the challenges discussed in §2.3.

3.1 Scheduling with Soft Admission Control

To ensure multi-stage SLO attainment, SLOs-Serve's scheduler introduces a *soft admission control* mechanism that guarantees SLO attainment for admitted requests and handles declined requests (say, 2%) with fallback mechanisms. Periodically, the scheduler selects an optimal subset of new requests that can attain their SLOs. Then, it determines execution plans, integrating these selected requests into future batched executions.

Algorithm 1 shows the pseudocode of SLOs-Serve's scheduling with soft admission control. The scheduler is invoked upon the occurrence of a timeout or when the number of new or completed requests surpasses a predefined threshold. Upon invocation, the scheduler takes in states for running and new requests, as well as a performance model, to generate admission decisions and future batch schedules (Line 2). Declined requests are (Line 5) specially handled, as elaborated later in §4. For accepted requests, the batch schedules ensure SLO attainment. Each batch is represented as:

$$Batch := [(ID_i, S_i \in \{Prefill, Decode\}, \#Token_i)_i], \quad (1)$$

where each entry specifies the processing of $\#Token_i$ tokens for request ID_i in state S_i. A batch supports chunked prefill by processing less than full tokens for prefill requests, as well as speculative decoding by verifying more than one token for decode requests. The computed batches are executed using a stateless `BatchForward` function (Line 9), which we elaborate in Algorithm 3 in the Appendix.

Algorithm 1: SLOs-Serve's LLM Serving

Input: $Reqs_{new}$
State: $Reqs_{running}, thresh_{new}, thresh_{finished}, \mathcal{M}$: $Perf.\ Model$

1 **Infinite Run:**
2 $Reqs_{admitted}, Reqs_{declined}, Batches \leftarrow$Schedule($Reqs_{running}$, $Reqs_{new}$, t, $\mathcal{M}$)
3 $Reqs_{running} \leftarrow Reqs_{running} + Reqs_{admitted}$
4 $Reqs_{new} \leftarrow \emptyset$
5 **foreach** *request in* $Reqs_{declined}$ **do**
6 Perform best effort serving (§4.1) or route to another replica (§4.2).
7 $\#finished \leftarrow 0$
8 **for** *batch in Batches* **do**
9 Call `BatchForward`($batch$);
10 **foreach** *request in* $Reqs_{running}$ **do**
11 **if** *request is finished* **then**
12 $Reqs_{running} \leftarrow Reqs_{running} - request$
$\#finished \leftarrow \#finished + 1$
13 **if** *TimeOut or* $\#finished > thresh_{finished}$ *or* $|Reqs_{new}| > thresh_{new}$ **then**
14 Goto line 1.

3.1.1 Performance Modeling for Batch Execution.

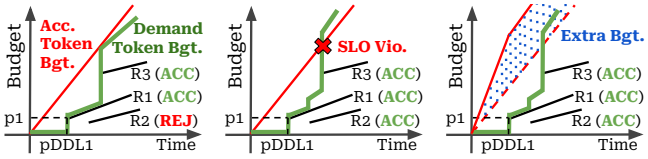
To make scheduling decisions, the scheduler relies on a performance model that accurately characterizes per-batch execution times. We design our model to work for different hardware architectures by adopting a generalized Roofline model [35]. Specifically, the execution time of a `BatchForward` call is modeled as:

$$\hat{T}(\texttt{BatchForward}(batch)) = max_l\ (k_1^l \#Tokens + k_2^l \#SpecStep + b_l)$$
$$\#Tokens := \Sigma_{i \in batch} \#Token_i$$
$$\#SpecStep := max_{i \in batch,\ S_i \text{ is prefill}} \#Token_i$$

where the k_1^l, k_2^l, b_ls are parameters obtained by regression on profiled data from runs on (multi-)GPU backends. Each term ($l = 2$ in practice) within the max function represents a source of execution time, such as $k_1^l \#Tokens$ for computation time, constant b_l for fixed memory access to the model weights, and $k_2^l \#SpecStep$ for the speculative model's overhead. The max operation identifies the bottleneck that limits performance, assuming sufficient parallelism. We validate the fidelity of the performance model against empirical measurements on A100s and H100s in Fig. 10b.

3.2 SLOs-Serve's scheduling algorithm

We now present SLOs-Serve's dynamic-programming based scheduling algorithm (Line 2 of Algorithm 1).



(a) Valid Scheduling with admission control.

(b) Invalid Scheduling w/o admission control.

(c) Gain from Dynamic Batch Size Tuning.

Figure 5. Illustration of the scheduling algorithms

An Example. We use a toy example in Fig. 5 to provide the intuition. This example requires the scheduler to make scheduling decisions on three new requests, SLOs of which are represented by lines. The line for request R_i starts at point $(pDDL_i, p_i)$, representing that p_i tokens must be allocated to R_i before the prefill deadline $pDDL_i$, and then grows at rate k_i tokens/s, representing token generation with speed k_i. This representation encodes multi-SLOs across requests and stages. For example, $R2$ may represent a chatting request with smooth decoding speed, $R1$ as a coding request with higher generation speed demand, and $R3$ as a summarization request with a longer input.

The *accumulated token budget* line (red) depicts the total available tokens, the slope of which represents the token throughput decided by the launched batches. In Fig. 5b, 5a, we assume a fixed token batch size over time, which results in a linear budget increase. Fig. 5c demonstrates a more efficient approach through *dynamic batch size tuning*, enabling nonlinear budget growth with higher token throughput.

The scheduling problem—admission control and batch schedule determination—is equivalent to finding a subset of lines whose cumulative demand always remains below the accumulated token budget. Violation of this condition, where the cumulative demand of admitted requests surpasses the budget, negates feasible schedules. Meeting the condition, on the other hand, ensures that the system can satisfy the decoding token requirements of all admitted requests, as their combined generation rate is less than the budget's slope, and the available budget at each request's prefill deadline exceeds the accumulated prefill demand.

For example, admitting all three requests in Fig. 5b (indicated by ACC) results in demand exceeding the budget before $R3$'s prefill deadline, leading to a violation. However, admitting only $R1$ and $R3$ (Fig. 5a) keeps the cumulative demand within the budget, indicating a valid batched schedule. Furthermore, implementing dynamic batch size tuning allows the attainment of SLOs for all three requests because of the enlarged token budget (Fig. 5c).

3.2.1 Scheduling via Dynamic Programming.

To make admission control and batching decisions, SLOs-Serve uses a multi-SLO dynamic programming (DP) algorithm.

We use the following notations. Suppose there are N requests in total, request R_i arrives at t_i with prompt of length p_i and a memory requirement m_i. R_i comprises a prefill stage followed by a decode stage. Every request has a deadline for its prefill stage $pDDL_i$. For the decode stage, a request can enforce the TPOT SLO for every token from a vector of choices $TPOT_1, TPOT_2, ..., TPOT_L$. We say that a request's SLO is attained if and only if the SLO for every stage is satisfied. Successfully attaining R_i's SLO yields value v_i.

The scheduling algorithm optimizes batch schedules to maximize the total value gained from attaining requests' SLO. We solve the problem using dynamic programming that incrementally calculates $DP[i, m, pb, n]$, defined as the best value a scheduler can ever get from serving the first i requests with earliest prefill deadlines, while generating *pb prefill budget* by the end of R_i's prefill deadline ($pDDL_i$) within m memory units. The prefill budget, defined as the token budget remaining after satisfying the decode SLOs for accepted requests, is always positive and is used to attain the prefill SLOs for unscheduled requests, i.e., requests with later prefill deadlines. The state transition function is

$$DP[i, m, pb, n] = \max_{\substack{j,\\ pDDL_j < pDDL_i,\\ \Delta pb \geq 0,\ m \geq m_i}} \left\{ DP[j, m - m_i, pb + p_i - \Delta_{i,j}^{n-1} pb, n - 1] + v_i \right\}.$$

This equation enumerates over the last accepted request j, whose prefill deadline precedes i's, to find the optimal prior state that leads to the highest value. To generate pb prefill tokens by $pDDL_i$, the prefill budget at $pDDL_j$ must be at least $pb+p_i-\Delta_{i,j}^{n-1}pb$. Here, p_i is prefill budgets consumed by R_i'th prefill stage, and $\Delta_{i,j}^{n-1}pb$ represents the newly generated prefill budgets during the interval between R_i and R_j's prefill deadlines, with $n-1$ accepted requests. Specifically, SLOs-Serve solves a constrained optimization problem to calculate the $\Delta_{i,j}^{n}pb$:

$$\Delta_{i,j}^{n}pb = PB^*(pDDL_i - pDDL_j, n), \text{and} \tag{2}$$

$$PB^*(t,n) = \max_{\text{partial batches}} \sum_{b\in\text{partial batches}} b.\text{prefillBudget},$$

$$\text{subject to attaining decode SLOs for } n \text{ requests.} \tag{3}$$

The equation above solves for partial batches that maximizes total prefill budgets while attaining decode SLOs in a time interval t. A partial batch specifies its total token budget, allocates them to decode requests and leaves the rest as prefill budget. The partial batch's total token budget and decode token allocation can be used by the performance model in §3.1.1 to estimate the per-batch execution time. Using the performance model, a solver identifies the optimal list of partial batches that maximize the remaining prefill budgets after satisfying the decode SLOs of accepted requests. We later present two solvers: one for auto-regressive decoding based serving (Algorithm 2), and another for speculative-decoding based serving (§3.2.3).

The Optimal Solution. The optimal value v^* is identified by $(i^*, n^*), v^* = \arg\max_{i,n} DP[i, M, 0, n]$, where M denotes total memory units. Here, R_{i^*} is the last accepted request, and n^* is the total number of accepted requests. We reconstruct the optimal schedule by tracking $j^*[i^*, M, 0, n^*]$ from Eqn. 5 for admission decisions, and $B^*[i^*, M, 0, n^*]$ from Eqn. 3 for the scheduled partial batches. Since tokens in partial batches are only allocated to decode requests, we allocate the prefill budget in every batch by prioritizing requests with earlier prefill deadlines. It is guaranteed to have sufficient prefill budgets for allocation, because the DP algorithm requires the prefill budget to be non-negative by every prefill deadline.

Continuous Optimization. The scheduler, previously discussed for a new-request-only scenario, can be extended to accommodate running requests through forced admission. Forcing admission for running requests maintains SLOs-Serve's invariant of guaranteeing SLOs for all admitted requests. Specifically, during enumeration in Eqn. 5, the last admitted request before R_i is constrained to either the latest running request with a prior prefill deadline or any request within their respective deadlines. Continuous optimization, as shown in §6.4, minimizes the overhead imposed by running requests.

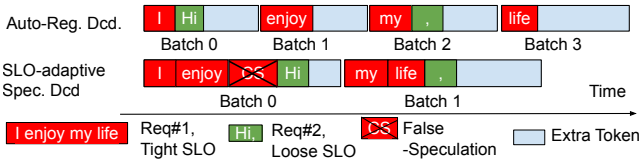


Figure 6. SLO-adaptive Speculative Decoding

Multi-Decode SLOs. For a request with multiple decode SLOs (e.g., a reasoning request with a tight SLO for thinking and a loose SLO for response), its tightest SLO is taken to upper-bound the resource demand. When requests have different decode SLOs, the DP scheduler tracks the accepted requests per TPOT SLO, updating its state to include SLO-specific request counts: $(i, m, pb, (n_1, n_2, ..., n_L))$. This information enables the solver to generate optimal batch schedules across varying decode SLOs.

Time Complexity. Given N total requests, N_{new} new requests, L decode tiers, and M memory units, since every transition enumerates at most N_{new} previous states, the total time complexity is bounded by $O(|States| \times N_{new}) = O(N \cdot N_{new}^{L+1} \cdot M \cdot \Sigma_i p_i)$. When the optimization goal is request throughput under SLOs, the time complexity is reduced to $O(N \cdot N_{new}^{L+1} \cdot M)$ (see Appendix C). In practice, with typical workloads of zero to ten new requests and dozens to hundreds of running requests, our scheduler introduces negligible overhead (see more in §6.4).

3.2.2 Batch Formation with Dynamic Size Tuning.

We now describe our approach to form batches in Eqn. 3 with auto-regressive decoding (top row in Figure 3.2.3) that maximizes the prefill token budgets subjecting to the decode SLO constraints. Since token throughput increases monotonically with batch sizes (Fig. 2), we form batches by *selecting the largest possible batch size that satisfies the decoding SLOs of all running requests.* Algorithm 2 details this process. First, the tightest TPOT SLO among running requests is determined (line 1), which sets the latency for all scheduled batches. Subsequently, tokens are allocated to decoding requests in an early deadline first priority queue (lines 4-14), ensuring their SLOs are met. Compared to Sarathi-Serve, which globally caps batch sizes based on the tightest TPOT SLO, this algorithm improves token throughput by dynamically adapting batch sizes to the current set of running requests.

3.2.3 SLO adaptive Speculative Decoding.

We extend the batch formation technique to incorporate speculative decoding for enhanced efficiency. With speculative decoding, token throughput is increased by relieving the per-batch latency constraint through processing more tokens per batch. This is because, as illustrated in Fig. 6, auto-regressive decoding constrains batch token capacity based on the tightest decode SLO. In contrast, speculative decoding permits the processing of multiple decode tokens per request within a

Algorithm 2: Batch Formation

Input: t, $Reqs_{Decoding}$, $\mathcal{M}$: Perf. Model
Output: *Batches*

$t_0 \leftarrow \min_{req \in Reqs_{Decoding}} req.TPOT$;
for $req \in Reqs_{Decoding}$ **do**
  $req.schDDL \leftarrow 0$;
$Q \leftarrow \text{PrioQueue}(Reqs_{Decoding}, \text{schDDL})$;
$batches \leftarrow []$;
for $i \leftarrow 0 \ldots \lfloor t/t_0 \rfloor - 1$ **do**
  $b \leftarrow \text{Batch}(prefillBgt = \mathcal{M}.time2bs(t_0))$;
  while $Q.front().schDDL \geq i \cdot t_0 \wedge b.prefillBgt > 0$ **do**
    $req \leftarrow Q.\text{pop}()$;
    $b.\text{scheduleDecode}(req)$;
    $b.prefillBgt \leftarrow b.prefillBgt - 1$;
    $req.schDDL \leftarrow req.schDDL + req.TPOT$;
    $Q.\text{push}(req)$;
  $batches.\text{append}(batch)$;
return *batches* ;

batch, thereby relaxing per-batch latency constraints (e.g., $2\times TPOT$ when generating two tokens per batch) and increasing token throughput. Furthermore, with multiple decode SLOs, token allocation can be dynamically adjusted per request, enabling SLOs-Serve to maximize the prefill token budget within a defined time frame. A contemporary work [16] proposes a similar idea to customize token allocation in speculative decoding for multi-decode SLOs. However, they are focusing on a decode-only setup, while SLOs-Serve targets multi-stage scenarios and utilizes speculative decoding to enable higher token throughput.

Specifically, suppose there are $n_1, ..., n_L$ requests in the decoding stage running with TPOTs $TPOT_1, ..., TPOT_L$, we decide the batch in Eqn. 3 with the maximum prefill token throughput by solving the following problem (see details in Appendix. D):

$$\max_{sl_{1:L}} prefillTpt := \frac{\text{PrefillBgtPerBatch}}{\text{Batch Time}}$$

$$\text{PrefillBgtPerBatch} = Time2BS(T(sl_{1:L}), sl_{1:L}) - \sum_i n_i sl_i$$

$$\text{Batch Time} = T(sl_{1:L}) = \min_{l \in 1,2,...,L} (TPOT_l \cdot Acc(sl_l))$$

Here, $sl_{1:L}$ are per-decode SLO speculation lengths, and $Acc(sl_l)$ is the expected number of tokens generated for requests with $TPOT_l$. Then, the optimal speculation lengths and prefill budget are used to construct batches used in the DP algorithm.

To account for the uncertainty in speculative decoding's prediction, we dynamically adjust the request's decode SLOs. For example, we strengthen its SLO when a request falls behind its SLO in the decoding stage.

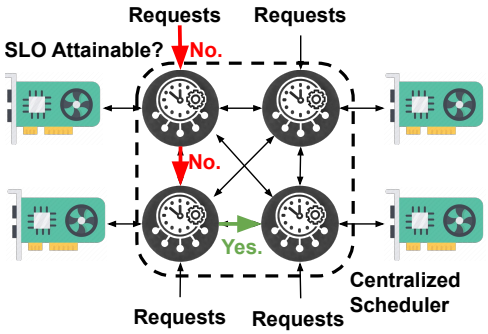


Figure 7. SLOs-Serve's Multi-Replica Serving.

4 SLOs-Serve

Building on top of the soft admission control mechanism, we design SLOs-Serve with two fallback mechanisms that enable *burst resilient scheduling* and *SLO-driven request routing*.

4.1 Burst Resilient Scheduling

When request bursts occur, resulting in instantaneous loads exceeding server capacity, SLOs-Serve introduces *burst resilient scheduling*. This mechanism delays requests with unattainable SLOs to ensure SLO attainment for the remaining requests.

In addition to standard services with defined SLOs, SLOs-Serve incorporates a cost-effective best-effort service tier. This tier utilizes any leftover resources after fulfilling SLO-guaranteed requests, similar to OpenAI's batch API [23], which offers lower costs with a longer response time. The scheduler seamlessly integrates this best-effort tier by batching requests to consume surplus token budgets and executing them when sufficient memory is available. To prioritize SLO-guaranteed requests, the scheduler can preempt best-effort requests upon new arrivals. Preemption overhead is minimized by discarding only the cached states (KV Cache), while retaining the generated tokens. This allows preempted requests to resume with a single prefill step, recomputing the cache for the original prompt and previously generated tokens, rather than repeating the entire decoding process.

SLOs-Serve then employs the best-effort service tier to handle bursts and requests with unattainable SLOs. After the admission decision by the scheduler, requests with unattainable SLOs are offloaded to the best-effort tier. While this tier lacks SLO guarantees, it is shown later that these requests are efficiently handled during low or zero-load periods following the burst (Fig. 11), maximizing system utilization.

4.2 Multi-replica Serving with SLO-driven Request Routing

SLOs-Serve's scheduler and performance modeling provide the foundation for a scalable multi-replica LLM serving system with high per-GPU utilization.

Fig. 7 shows the architecture of SLOs-Serve's multi-replica serving framework. Each replica is associated with a scheduler operating within a centralized controller, which virtualizes execution using the performance model. Upon arrival of a new request at a replica (dispatched by a load balancer), the scheduler determines SLO attainability. If the current replica cannot meet the SLO, the request is routed to the next replica sequentially. When the routing counts reach a pre-configured limit, a backup policy is invoked, deciding whether to decline the request or offload it to lower-tier resources. While previous works on LLM load balancing always rely on device profiling with indirect metrics (e.g., device loads and memory consumption) [13, 33, 34], SLOs-Serve directly balances the load based on SLO attainment, looking forward to full GPU utilization and rigorous SLO attainment. Note that sequential routing is one of many ways [7] to leverage SLOs-Serve's admission control mechanism in multi-replica routing, and we leave the exploration of this to future work.

5 Implementation

We build SLOs-Serve as a distributed LLM serving system that supports multi-SLOs, multi-replica serving, while being resilient to request bursts. Currently, we support vLLM as our backend but it is easy to extend to other backends as long as they comply with our batched execution model. We utilize PagedAttention [14] for memory management and Ray [19] for the orchestration between the scheduler and the backend. The implementation of the scheduling orchestrations with the executor interface, serving frontend, takes 10K lines of Python Code, whereas the resource planning algorithm is highly crafted in C++ with 1.5K LoC.

6 Evaluation

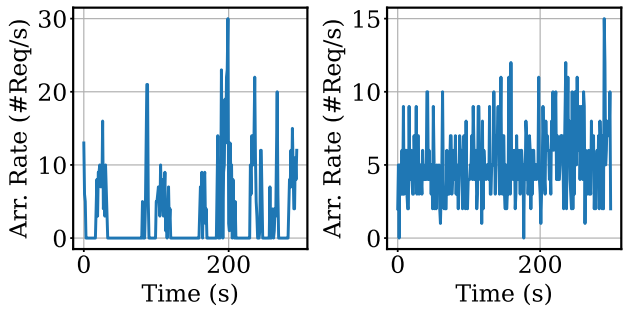
We evaluate SLOs-Serve on a total of six scenarios (Tab. 2) and five datasets (Tab. 4). We consider vLLM, Sarathi-Serve, DistServe as our baselines as they represent SOTA for LLM serving. Our evaluation seeks to answer these questions:

- What is the maximum capacity (i.e., request load) a single GPU can serve under specific SLO constraints with different serving systems, and how well does SLOs-Serve stay resilient with bursty arrivals (§6.1)?
- In multi-replica serving, how does SLOs-Serve scales with the number of replicas (§6.2)?
- How do SLOs-Serve's individual optimizations in isolation contribute to its performance (§6.3)?

Setup. We evaluate SLOs-Serve across different models: OPT series [39] for standard scenarios (ChatBot, Coder, Summarizer), ToolLlama [29] for ToolLLM, and a distilled Deepseek-R1-Qwen model [9] for the reasoning scenario—these models are either used in evaluation by previous work [15, 41] or well-suited for the application. For the end-to-end capacity evaluation, we use the Google a2-highgpu-4g VM with 4 NVIDIA 40GB A100 GPUs, connected with pairwise NVLINK. For the OPT-13B (OPT-30B) model, we run with 2-way (4-way) tensor parallelism. For the OPT series, we use OPT-125m as the speculative decoding model and replicate it on every GPU. For the OPT-7B model, we run SLOs-Serve with up to 4 replicas. For other models, we run SLOs-Serve with a single replica. We run SLOs-Serve with a best-effort tier across all scenarios. For memory management, we assume prior knowledge on decode lengths available to all baselines, which can be well-approximated from previous works [13, 26, 28, 34]. For the scalability evaluation, we use an additional Google a3-highgpu-8g with 8 NVIDIA 80GB H100 GPU to perform multi-replica serving for the OPT-13B model.

Table 2. Experiment Scenarios.

Scenarios	Arrival Pattern	Prompt Dataset	Model
ChatBot	Azure-Chatting [25]	ShareGPT [1]	Main: OPT-7B/13B/30B [39]
Coder	Azure-Coding [25]	HumanEval [4]	
Summarizer	Azure-Chatting	Arxiv Summary [5]	Spec: OPT-125m
Mixed	ChatBot + Coder + Summarizer		
ToolLLM	Azure-Coding	ToolBench [10]	ToolLlama-7B [29]
Reasoning	Azure-Chatting	s1K [20]	Deepseek-R1-Qwen-1.5B [9]



(a) Coding Trace. **(b)** Chatting Trace.

Figure 8. Traces from Azure Dataset [25] used in evaluation.

Workloads. We used Azure LLM inference traces [25] for realistic request arrival patterns: Coding shows bursty arrivals, while Chatting is more stable (Fig. 8). Request lengths are based on Azure traces, supplemented by Arxiv Summary [5], ToolBench [10], and s1K [20] for specific scenarios (Tab. 4). ChatBot and Reasoning are decode-heavy with long outputs, Summarizer is prefill-heavy, and Coder and ToolLLM exhibit the largest input/output length variations.

SLOs. We enforce the SLO based on the specification in Tab. 1. Particularly, we focus on the max TTFT slowdown compared to zero-load setup for the prefill stages and max TPOT for the decode stages. Because speculative decoding outputs more than one token at a time, we measure the TPOT every 10 tokens.

Baseline. We consider vLLM [14] (commit `c38eba30`), Sarathi-Serve [2] (enabled with vLLM), and DistServe [41] (commit `0a2d0c99`) as our baseline as they represent the

SOTA serving frameworks. For vLLM, we additionally evaluate its speculative decoding version (vLLM (Spec)). For Sarathi-Serve, we configure the batch size to the maximum size without violating the tightest decode SLO. For DistServe, we experimented with different prefill-decode device allocations (1:1, 2:1, 1:2) and report the best result.

Metric. We evaluate serving capacity (maximum GPU request load with less than 10% SLO violations [2, 41]). For multi-GPU systems, we normalize the request rate by dividing the total request rate by the number of GPUs.

Table 3. SLOs for different model configurations.

	Max TTFT Slowdown	Max TPOT
Tight SLO	3x	50ms
Loose SLO	5x	100ms

Table 4. Datasets used for evaluation. For the reasoning task, we list the number of tokens for thinking/response. For the ToolLLM, there is 2.7 +- 1.1 prefill-decode pairs in a request.

	Prompt Tokens			**Output Tokens**		
	Mean	P99	Std.	Mean	P99	Std.
ChatBot	763	1591	424	266	619	160
Coder	847	2010	617	26	232	47
Reasoning	127	421	83	4693/803	7297/1650	1442/280
Summarizer	1333	1946	444	202	1508	234
ToolLLM	690	2131	356	116	363	66

6.1 End-to-end Capacity Evaluation

The end-to-end evaluation (Fig. 9) demonstrates that SLOs-Serve consistently outperforms baseline systems. Specifically, it achieves a 1.76x capacity improvement over vLLM, 1.94x over Sarathi, and 2.6x over DistServe. The performance gain is attributed to SLOs-Serve's optimizations, including SLO-optimized, burst-resilient scheduling, as well as request routing. Even when enforcing a stringent 2% SLO violation constraint, which mitigates the impact of delaying requests with unattainable SLOs, SLOs-Serve maintains a capacity advantage in the ChatBot scenario: 1.91x over vLLM, 2.19x over Sarathi, and 2.4x over DistServe. These results underscore SLOs-Serve's robust support for multi-stage SLOs.

With stable request arrival rates in both ChatBot and Summarizer scenarios (Figure 9), maximizing token throughput under multi-stage SLOs is the key to high serving capacity. Among baselines, vLLM prioritizes prefill, resulting in widespread decode SLO violations. SLOs-Serve and Sarathi multiplex stages, but Sarathi's fixed batch size limits throughput, particularly in the Summarizer scenario with longer input text and tight prefill SLOs. SLOs-Serve addresses this with SLO-adaptive speculative decoding and dynamic batch size tuning, enabling larger batches. Specifically, shown in Figure 10a, SLOs-Serve utilizes batches exceeding 512 tokens for up to 25% of execution time, while Sarathi is capped at 512. Consequently, SLOs-Serve achieves 1.41-1.70x and 1.14-1.17x serving capacity than baselines across OPT-7B, 13B, and 30B models for ChatBot and Summarizer, respectively.

Coder and ToolLLM scenarios feature bursty request arrivals (Figure 8a) and short outputs, highlighting SLOs-Serve's burst-resilient scheduling. SLOs-Serve effectively manages load spikes (Figure 11) by deferring requests with unattainable SLOs (red line) during surges and processing them in low-load periods, maintaining SLO attainment for others (green line). Baselines, lacking this capability, suffer from cumulative delays during bursts, leading to violations of all request SLOs. Consequently, SLOs-Serve achieves up to 2.1x and 1.90x serving capacity than baselines in Coder and ToolLLM scenarios (Figure 9, third row and leftmost figure on the last row), respectively. This demonstrates robust burst handling, especially notable in ToolLLM where SLOs-Serve operates without the benefit of a speculative decoding model.

The Mixed scenario simulates diverse application workloads, testing system adaptability to varied patterns and SLOs. Shown in Figure 12, at 1.5 requests/second, vLLM and Sarathi exhibit significant p99 TTFT degradation, exceeding SLOs. Conversely, SLOs-Serve admission control mechanism maintains p99 TTFT and TPOT near specified SLOs for the standard services.

Lastly, the reasoning scenario has long generation lengths with customized SLOs for the thinking stage and the decode stage. The longer per-request lifespans indicate more concurrent requests in the system, magnifying the stalls caused by greedy schedulers. In contrast, SLOs-Serve's SLO-optimized scheduling algorithm ensures multi-stage SLO attainment, as well as improving throughput by tuning the batch sizes to fit the per-request SLOs. As a result, even without a speculative model, SLOs-Serve supports 2.4x serving capacity compared to baselines (Figure 9, second on the last row).

6.2 Scaling Evaluation

Now, we evaluate SLOs-Serve's multi-replica on the cluster with 4 A100 GPUs, serving the OPT-7B model with up to 4 replicas. Figure 13 shows the serving capacity across five scenarios. Following the mechanism in Figure 7, a request is first routed to a replica in by a one-shot round-robin dispatcher. Afterwards, depending on the admission decision of SLOs-Serve's scheduler for that replica, the request either gets served or routed to the next replica.

Across five cases, SLOs-Serve exhibits linear or higher scaling. For example, in the ChatBot scenario, SLOs-Serve sustains 2.18x, 3.36x, and 4.61x higher capacity with 2,3,4 replicas when serving an OPT-7B model.

SLOs-Serve's super-linear scalability, while seems counterintuitive, is the result of its scheduling mechanism (Algorithm 1). Although SLOs-Serve's scheduler makes locally

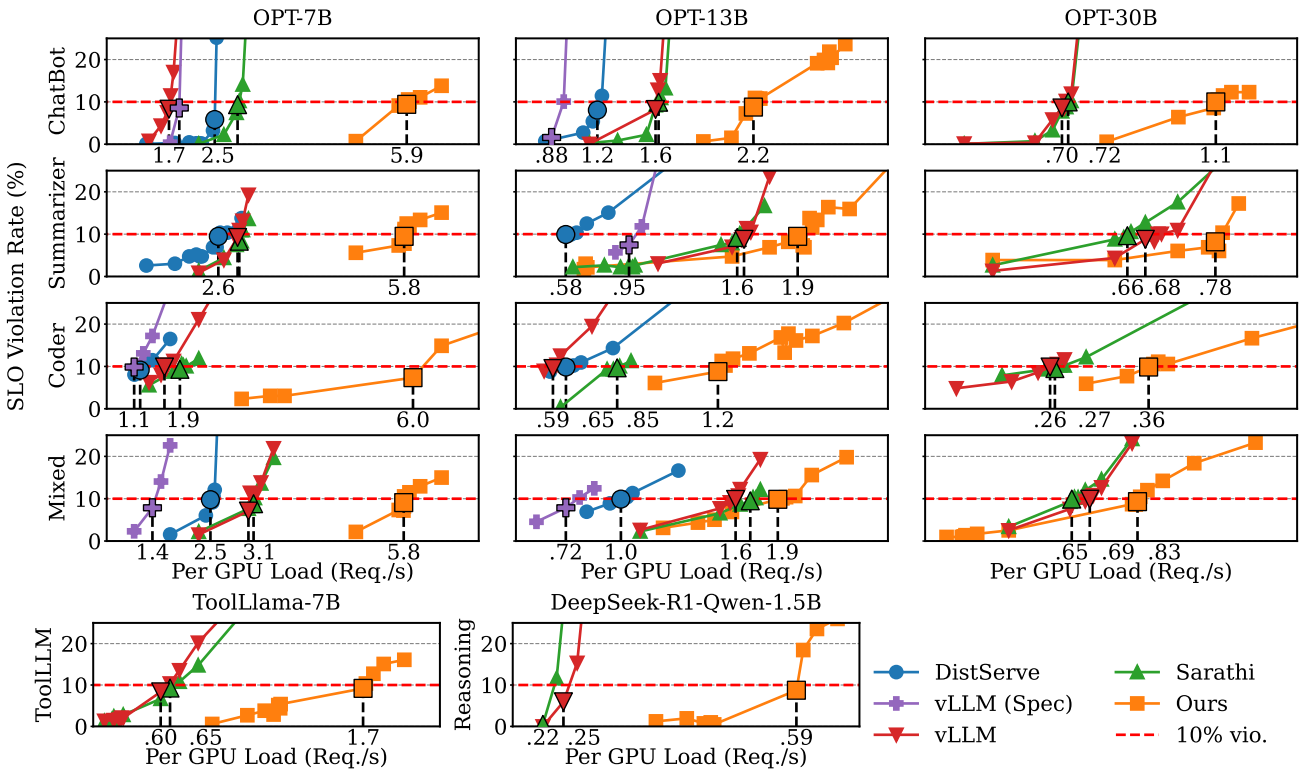
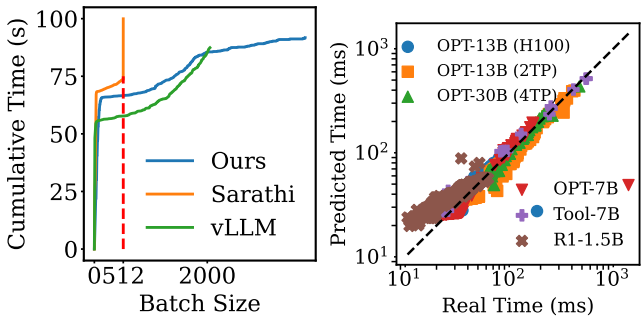


Figure 9. End-to-End evaluation of SLOs-Serve.



(a) Cumulative Execution Time along Batch Sizes for the OPT-7B with Summarization Scenario at 3.0 request/s. **(b)** Predicted Time vs. Real Time for the performance model across LLMs, #Tensor Parallel (TP), and hardwares (A100 if unmarked).

Figure 10. Analysis.

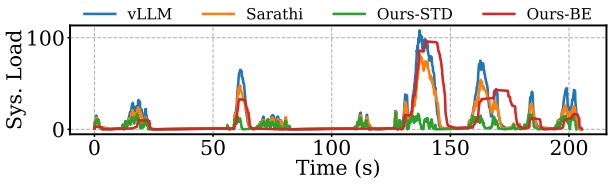


Figure 11. System Load (# Req. in System) across time for Coder under high-load (4.5 Req/s) scenario. Ours separates into standard services (STD) and the best effort service (BE).

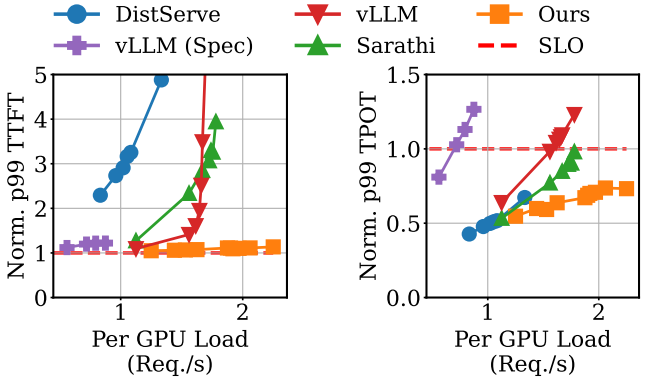


Figure 12. TTFT/TPOT Comparison for Mixed Scenario serving 13B model. Ours shows the requests belonging to the standard service tier.

optimal admission decisions upon each invocation, it is inherently greedy to currently present requests. When scaling to a multi-replica configuration, while each individual replica's scheduler maintains a constant request rate, the aggregate of all schedulers across replicas observes a linearly increasing number of requests. This expanded global view, facilitated by request routing, enables the combined scheduler network to make more informed, globally optimized admission decisions. Consequently, in scenarios like ToolLLM and Coder, where requests exhibit significant variance in input/output

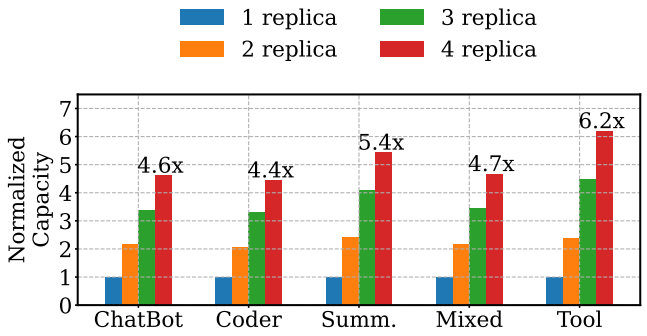


Figure 13. Capacity Scaling in SLOs-Serve's multi-replica Serving on 4 A100 GPUs with OPT-7B model.

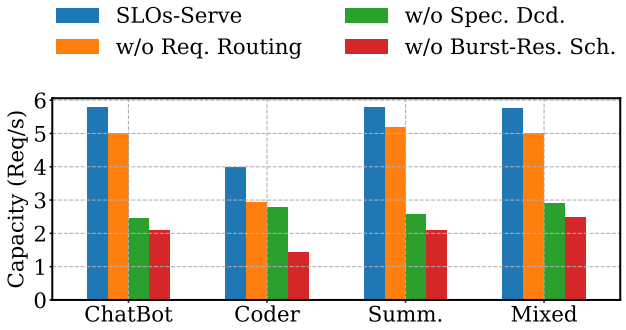


Figure 14. Ablation Study. We measure the serving capacity by gradually removing SLOs-Serve's optimizations.

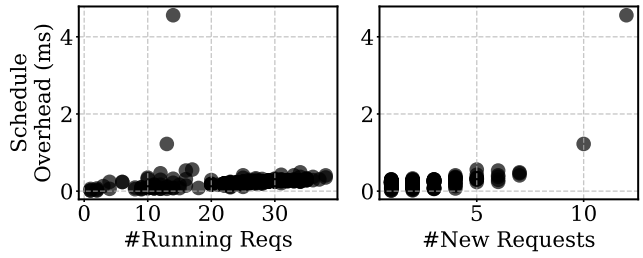


Figure 15. Schedule Overhead.

lengths, this multi-replica serving strategy mitigates the limitations of local greedy decisions, resulting in a 5.44x and 6.2x increase in served load on four replicas, respectively.

6.3 Ablation Study

We conducted an ablation study (Figure 14) to quantify the individual contributions of SLOs-Serve's optimizations. Starting from full SLOs-Serve, we remove multi-replica serving with request routing, SLO-adaptive speculative decoding, and the burst-resilient scheduling across four scenarios. For the baseline case, we implemented a prefill-oriented scheduling approach in SLOs-Serve for apple-to-apple comparison.

We found that request routing, SLO-adaptive speculative decoding, and burst-resilient serving independently improve serving capacity by 1.19x, 1.66x, and 1.34x. SLO-adaptive decoding demonstrated its largest gains in decode-heavy workloads (2.01x for ChatBot and 2.02x for Summarizer) by enabling larger token batch sizes. Conversely, request routing (1.92x) and burst-resilient serving (1.66x) were most effective in the bursty, variable-length Coder scenario, where they balanced instance load and deferred request with unattainable SLOs to secure the SLOs of the rest, respectively.

6.4 Scheduling Overhead

Last but not least, we assess the scheduling overhead of SLOs-Serve's scheduling algorithm. As depicted in Figure 15, the scheduling overhead for each call to SLOs-Serve's resource planning algorithm, profiled from real scheduling scenarios, consistently remains under 10 milliseconds, with the majority of overheads concentrated below 2 milliseconds. Considering that the scheduler typically plans 1-5 future batches, and each batch requires at least 25 milliseconds to complete, the overhead introduced by the scheduling algorithm is minimal. This ensures that the low-overhead request routing is in multi-replica serving.

6.5 Fidelity of the performance model

To validate our performance model, we conducted evaluations across diverse configurations (models, tensor parallel setups, speculative decoding, hardware architecture). Figure 10b shows that the model consistently achieved high fidelity, with R-squared scores ranging from 0.82 to 0.93.

7 Related Work

LLM Serving System. There are a variety of recent works on serving Large Language Models (LLM). These works have diverse research focus, including optimizing attentions [14, 27, 36], prefill/decode disaggregation [25, 41], KV cache [12, 28, 40], heterogeneous serving [17], on-device serving [11, 32], serverless serving [38], and serving emerging LLM model variations [3, 6]. A few previous works focus on similar goals (optimizing for SLOs) as SLOs-Serve, including Llumnix [34], DynamoLLM [33], ExeGPT [21] and VTC [31]. Comparing to these works, SLOs-Serve uses admission control paired with a dynamic programming alaogirithm to guarantee attainment of multi-stage SLOs.

Admission Control. *Admission control* is often needed when the server is under high request load. By rejecting some incoming requests, admission control ensures SLOs of running requests are satisfied. In the literature, admission control is used in cloud services [7, 30], elastic training [8]. Along serving frameworks like vLLM [14] may decline requests when the memory is not sufficient, SLOs-Serve admission control directly focuses on SLO-attainment.

8 Conclusion

In this work, we built SLOs-Serve, a distributed LLM serving system that supports multi-SLOs and multi-replica serving, while being resilient to burst arrivals. Particularly, we solve the key challenge of serving under application- and stage-dependent SLOs in a resource-sharing environment by customizing token allocations in batches. SLOs-Serve then designs a dynamic programming-based algorithm that effectively detects requests with unattainable SLOs and continuously optimizes the token compositions by exploring the full design space of chunked prefill and (optional) speculative decoding. Built on top of the scheduler, SLOs-Serve enabled burst handling and low overhead routing in multi-replica serving. In evaluation, comprehensive studies across six applications showed on average 2.2x improvement in serving capacity. In multi-replica serving, SLOs-Serve embodied super-linear scaling with bursty arrivals, underscoring the effectiveness of dynamic request routing.

9 Acknowledgement

This work is partly supported by grants from the National Science Foundation (NSF CNS-2211882), and by the member companies of the Wasm Research Center and PDL consortium. We thank Juncheng Gu for precious comments on the paper.

References

[1] [n. d.]. ShareGPT. https://sharegpt.com/. Accessed: 2024-12-07.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310* (2024).

[3] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. MoE-Lightning: High-Throughput MoE Inference on Memory-constrained GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 715–730. doi:10.1145/3669940.3707267

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[5] Arman Cohan, Franck Dernoncourt, Doo Soon Kim, Trung Bui, Seokhwan Kim, Walter Chang, and Nazli Goharian. 2018. A discourse-aware attention model for abstractive summarization of long documents. *arXiv preprint arXiv:1804.05685* (2018).

[6] Yinwei Dai, Rui Pan, Anand Iyer, Kai Li, and Ravi Netravali. 2024. Apparate: Rethinking Early Exits to Tame Latency-Throughput Tensions in ML Serving. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 607–623. doi:10.1145/3694715.3695963

[7] Rainer Gawlick. 1995. Admission control and routing: Theory and practice. (1995).

[8] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[9] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[10] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. arXiv:2403.07714 [cs.CL]

[11] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. 2024. NeuPIMs: NPU-PIM Heterogeneous Acceleration for Batched LLM Inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 722–737. doi:10.1145/3620666.3651380

[12] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. 2024. Memserve: Context caching for disaggregated llm serving with elastic memory pool. *arXiv preprint arXiv:2406.17565* (2024).

[13] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. 2024. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181* (2024).

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[15] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[16] Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, et al. 2025. AdaServe: SLO-Customized LLM Serving with Fine-Grained Speculative Decoding. *arXiv preprint arXiv:2501.12162* (2025).

[17] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 586–602. doi:10.1145/3669940.3707215

[18] Microsoft. 2023. GitHub Copilot: AI-powered Code Assistant. https://github.com/features/copilot Accessed: 2025-03-03.

[19] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul,

Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[20] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393* (2025).

[21] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 369–384. doi:10.1145/3620665.3640383

[22] OpenAI. 2023. ChatGPT: Large Language Model Chatbot. https://openai.com Accessed: 2025-03-03.

[23] OpenAI. 2024. *Batch Processing Overview*. https://platform.openai.com/docs/guides/batch/overview?lang=curl Accessed: 2024-12-22.

[24] OpenAI. 2024. Summarizing Long document. https://cookbook.openai.com/examples/summarizing_long_documents Accessed: 2025-03-03.

[25] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[26] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2024. vattention: Dynamic memory management for serving llms without pagedattention. *arXiv preprint arXiv:2405.04437* (2024).

[27] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1133–1150. doi:10.1145/3669940.3707256

[28] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079* (2024).

[29] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).

[30] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, et al. 2023. Kerveros: Efficient and Scalable Cloud Admission Control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 227–245.

[31] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. 2024. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 965–988.

[32] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 590–606. doi:10.1145/3694715.3695964

[33] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741* (2024).

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

[35] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[36] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948

[37] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[38] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating Serverless LLM Inference with Materialization *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 653–668. doi:10.1145/3669940.3707285

[39] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL]

[40] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems* 37 (2024), 62557–62583.

[41] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670* (2024).

A Detailed Analysis into the disaggregated Scheduling.

In disaggreagated scheduling [25, 41], different stages are separated onto different (group of) devices. In this way, one can specialize for every stage's SLO by adjusting the hardware configurations [41], and balance workloads across stages by customizing adjusting the devices allocated to every stages.

The flexibility of disaggregated scheduling is hindered by the fact that the optimal hardware configuration depends on the load across stages, making it a hard fit for our setup where every stage has dynamic load characteristics. Shown in Fig. 4, decode-heavy workloads like coding requires more decoding device allocation (1:2), whereas prefill-heavy workloads like chatting requires more prefill device allocation (2:1). As a result, not a single device allocation strategy works across different requests, resulting in poor performance when the requests have mixed load-patterns.

For the simplicity of analysis, we assumes an ideal setting where the prefill devices is operating under the maximum throughput and the system scales perfectly with the number of devices. Suppose the maximum goodput is g, then for the prefill, the token budgets generated by all prefill devices per time unit must match the new tokens need to be processed

$$g \times E_r[InputLength_r] \leq MaxTokenTpt \times n_{prefill}$$

, where $n_{prefill}$ denotes the number of prefill devices.

For decode, every request in the system contributes to one token budget in the next batch. Therefore, the time to execute this batch should below the TPOT requirement. Particularly,

$$BatchTime(\#ReqInSystem) \leq TPOT$$

$$\#ReqInSystem \times n_{decode} = g \times E_r[TPOT \times OutputLength_r]$$

By approximating the $BatchTime$ function using

$$BatchTime(n) = \frac{n}{MaxTokenTpt} + Overhead$$

We have

$$g \leq (1 - \frac{Overhead}{TPOT}) \frac{MaxTokenTpt \times n_{decode}}{E_r[OutputLength_r]}$$

Hence, g is bounded by

$$g \leq MaxTokenTpt \times \frac{min(\frac{n_{prefill}}{E_r[I_r]}, (\frac{TPOT - Overhead}{TPOT}) \frac{n_{decode}}{E_r[O_r]})}{n_{decode} + n_{prefill}}$$

By optimizing the device allocation $n_{prefill}$ and n_{decode}, we have

$$g^* = \frac{MaxTokenTpt}{\frac{TPOT}{TPOT - Overhead} E_r[O_r] + E_r[I_r]} \tag{4}$$

Under the condition

Note that the balanced ratio (Eqn. 2.3) between the prefill devices and the decoding devices are influenced by two factors (i) the SLO requirement, and (ii) input/output lengths of requests. Consequently, service providers must reconfigure their systems to adapt to evolving service requirements.

Table 5. Common Statistics in a LLM serving framework. The request rate data is obtained from [2, 41].

Request Rate	Lifespan	Prefill Lifespan	Decode Lifespan
0.5-10 Req/s	0.7-10s	0.1-1s	0.5-8s

The dependency on SLO and token length patterns highlights the foundamental difficulty of disaggregated scheduling to support multi-tier services. Specifically, when requests exhibit drastically different input length/output length patterns as well as SLOs, no fixed configuration can achieve balanced device allocation, leading to inefficient device utilization.

B More on LLM Serving

Unlike traditional applications such as web search and online data mining, where strict run-time budgets constrain request scheduling, LLM serving involves longer request lifespans and coarser request arrivals. This allows for more advanced and comprehensive scheduling algorithms. For instance, statistics from common serving frameworks (Tab. 5) indicate that a scheduler invoked every second typically handles around 0.35 to 100 requests per second, providing ample opportunity for well-optimized scheduling decisions.

C Time Complexity Analysis

When optimizing for request throughput, i.e. the number of request accepted, the DP state in Eqn. 5 directly encodes the objective function. Under this setup, we can refactor the DP to a $pd[i, m, n]$ that calculates the maximal prefill token budgets available by request i's prefill deadline under m memory units while accepting n requests.

$$pb[i, m, n] = \max_{\substack{j, \\ pDDL_j < pDDL_i, \\ \Delta pb \geq 0,\ m \geq m_i}} \{pb[j, m - m_i, n - 1] - p_i + \Delta pb\}. \tag{5}$$

Therefore, the time complexity is reduced to $O(N \times M \times N_{new}^L)$

D SLO-adaptive Speculative Decoding

We now explain how to solve the following problem:

$$\max_{sl_{1:L}} prefillTpt := \frac{\text{PrefillBgtPerBatch}}{\text{Batch Time}}$$

$$\text{PrefillBgtPerBatch} = Time2BS(T(sl_{1:L}), sl_{1:L}) - \sum_i n_i sl_i$$

$$\text{Batch Time} = T(sl_{1:L}) = \min_{l \in 1,2,\ldots,L} (TPOT_l \cdot Acc(sl_l))$$

First, suppose the speculation accuracy is α, $Acc(sl_l) = \frac{1-\alpha^l}{1-\alpha}$. Suppose $l^* = arg\min_{l \in 1,2,\ldots,L}(TPOT_l \cdot Acc(sl_l))$, it is easy to other l's satisfy $sl_{l'} = arg\min TPOT_{l'} \cdot Acc(sl_{l'}) > TPOT_l \cdot Acc(sl_l)$, which has a close-formed solution. Therefore, we just need to enumerate l that minimizes $\min_{l \in 1,2,\ldots,L}(TPOT_l \cdot$

Algorithm 3: BatchForward At Time Step t

Input: $r_{1...m}$: prefill requests, $R_{1...n}$: decode request,
$p^t_{1..m}$: prefill token per request,
$s^t_{1..n}$: decode token per request

State: $C^{request,model}_{b:e}$:
KV Caches for *request* on *model* for tokens from b to e.,
$x^{request,model}_{b:e}$:
input tokens for *request* on *model* for tokens from b to e.
$y^{request,model}_{b:e}$:
predicted tokens for *request* on *model* for tokens from b to e.

Output: Results of the batch forward process for drafter and base models

Perform prefill on the drafter model;

$[y^{R,drafter}_{m_R:m_R+p^t_R}]_R, [C^{R,drafter}_{m_R:m_R+p^t_R}]_R \leftarrow BatchFWD_{drafter}([x^{R,drafter}_{m_R:m_R+p^t_R}]_R, [C^{R,drafter}_{1:m_R}]_R)$;

Perform iterative decoding on the drafter model to calculate speculations;

for $k = 1$ **to** $\max_r(s^t_r)$ **do**

  $[y^{r,drafter}_{n_r+k}]_r, [C^{r,drafter}_{n_r+k}]_r \leftarrow BatchFWD_{drafter}([x^{r,drafter}_{1:n_r+k}]_r, [C^{r,drafter}_{1:n_r+k}]_r)$

Perform batched prefill and verification on the base model;

$[y^{r,base}_{n_r+1:n_r+s^t_r}]_r + [y^{R,base}_{m_R:m_R+p^t_R}]_R, [C^{r,base}_{n_r+1:n_r+s^t_r}]_r + [C^{R,base}_{m_R:m_R+p^t_R}]_R \leftarrow BatchFWD_{base}([x^{r,drafter}_{n_r+1:n_r+s^t_r}]_r + [x^{R,base}_{m_R:m_R+p^t_R}]_R, [C^{r,base}_{1:n_r}]_r + [C^{R,base}_{1:m_R}]_R)$;

Verify the correctness of the speculation;

$[y^{r,base}_{n_r+1:n_r+a^r_t}]_r \leftarrow \text{BatchVerify}([y^{r,drafter}_{n_r+1:n_r+s^r_t}]_r, [y^{r,base}_{n_r+1:n_r+s^r_t}]_r)$;

$Acc(sl_l)$) and find one that generates largest prefill throughput. This takes constant times in practice, given the maximum speculation decode lengths is belowe 10 and there is 2-3 different decode SLOs.