SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling

Ke Hong [1 2] **Xiuhong Li** [2] **Lufang Chen** [2] **Qiuli Mao** [2] **Xuefei Ning** [1] **Guohao Dai** [3 2] **Shengen Yan** [1]
Yun Liang [4] **Yu Wang** [1]

Abstract

Serving large language models (LLMs) efficiently requires elaborate request scheduling to satisfy service-level objectives (SLOs). In the context of LLM serving, SLOs include the constraints on Time-to-First-Token (TTFT) and Time-per-Output-Token (TPOT). Existing serving systems apply a coarse-grained request scheduling that follows a fixed principle at different iterations during the serving procedure, leading to (1) a significant request latency distribution bias between TTFT and TPOT, and (2) a significant distribution variance among different requests as shown in Fig. 1(a), and hence causes disappointing SLO attainment.

We identify that fine-grained scheduling based on a formal description of the design space addresses the issues mentioned above. To this end, we first formulate a scheduling design space with flexible control of the request execution order and the workload at each iteration. Based on that, we introduce a state-aware scheduling strategy, which is aware of two kinds of states: the states from the single request perspective and the states from the systemic perspective, and further leverages the trade-off between TTFT and TPOT and the trade-off among different requests to improve the SLO attainment, as shown in Fig. 2. We implement SOLA with the above insights. Given SLO constraints, the evaluation shows that SOLA enhances the SLO attainment from 45.5% to 99.4%, and serves 1.04-1.27× more requests than the state-of-the-art systems on average.

1 Introduction

Large language models (LLMs) have achieved impressive intelligence ability, empowering a broad range of application domains including chatbots (Brown et al., 2020; Adiwardana et al., 2020; Roller et al., 2021), code assistants (Feng et al., 2020; Svyatkovskiy et al., 2020), retrieval-augmented generation (Lewis et al., 2020; Izacard & Grave, 2020; Guu et al., 2020), and agent-based applications (Bubeck et al., 2023; Ouyang et al., 2022; OpenAI, 2023). However, LLM service introduces a considerable latency due to the billion-scale model parameters (Zhou et al., 2024; Aminabadi et al., 2022; Vaidya et al., 2023; Hong et al., 2024). Hence, specifying the latency constraints in the service-level objectives (SLOs) for the practical demand is essential. For LLMs, the processing of a request contains two phases: *prefill* phase and *decode* phase. The *prefill* phase processes the input prompt and generates the first output token, and the *decode* phase generates the output tokens iteration by iteration in an auto-regressive manner, with one token at each iteration. The latency between request arrival and the first output token being generated via the *prefill* phase is denoted as Time-to-First-Token (TTFT), and the latency per token in the *decode* phase is denoted as Time-per-Output-Token (TPOT). TTFT and TPOT are both essential for user experience. For real-time applications including conversation, TTFT is important for responsive promptness, and TPOT is crucial for reading smoothness and efficiency.

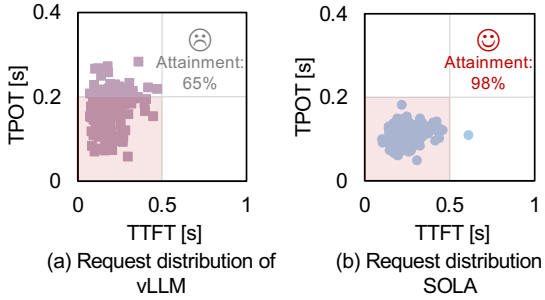


Figure 1. Using vLLM's default strategy vs. enhanced attainment (*i.e.*, the percentage of requests within the SLOs) with SOLA, when serving Llama3-70B (Meta, 2024) with ShareGPT (Sharegpt, 2023) dataset on 4 A100 GPUs. The SLOs are 500ms for TTFT and 200ms for TPOT, represented by the left corner box. The input request rate is set to 4.6 request/s.

LLM service providers receive thousands of requests within

[1]Tsinghua University [2]Infinigence AI [3]Shanghai Jiaotong University [4]Peking University. Correspondence to: Xiuhong Li <lixiuhong@infini-ai.com>, Guohao Dai <daiguohao@sjtu.edu.cn>, Yu Wang <yu-wang@tsinghua.edu.cn>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025.

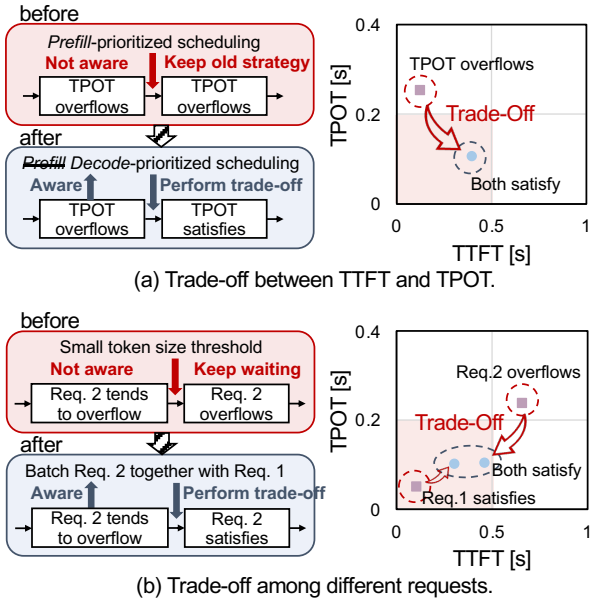


Figure 2. The twofold trade-offs in SOLA benefit the SLO attainment. (a) The trade-off between TTFT and TPOT from the perspective of a request. (b) The trade-off among different requests.

a short period. Adhering to the latency constraints in SLOs, how to improve the system throughput to support more requests is an important topic. In general, TTFT and TPOT are measured on individual requests. Prioritizing a single request means assigning more resources to the request within a time slice, which benefits its TTFT and TPOT. Nevertheless, system throughput focuses on the total of the requests. Over-assigning resources to a single request diminishes the resources available for others, negatively impacting the overall system throughput. Therefore, scheduling is essential for managing resource assignments when requests compete for limited resources. The goal of scheduling in LLM serving is to maximize the request processing rate of the system under TTFT and TPOT constraints.

Request batching is a preliminary scheduling method to improve throughput. However, requests in LLM serving may have different input and output lengths, making request-level batching inefficient. To this end, Orca (Yu et al., 2022) proposes the continuous batching technique, scheduling execution and batching requests at the granularity of iterations (an execution of *prefill* phase or *decode* phase) instead of requests. Based on that, Sarathi-Serve (Agrawal et al., 2024) and DeepSpeed-FastGen (Holmes et al., 2024) propose the *SplitFuse* method to chunk the *prefill* requests along the sequence dimension and schedule the *decode* requests together with the chunked *prefill* requests within one iteration. Thus, an iteration refers to the execution of a batch of *prefill* requests or *decode* requests, or a *mixed* batch containing both types of requests.

The essence of scheduling is forming a sequence of iterations where each iteration can be a *prefill*, *decode*, or a *mixed* iteration, and specifying the workload size for each iteration. Specifically, vLLM (Kwon et al., 2023) prioritizes *prefill* iteration, resulting in better TTFT at the cost of TPOT degradation. The higher priority of *prefill* iteration aggregates a large number of requests in *decode* phases, which improves the system throughput but prolongs the TPOT for each request. In contrast, FasterTransformer (NVIDIA, 2017a) prioritizes *decode* iteration, leading to better TPOT at the cost of worse TTFT and system throughput. Sarathi-Serve (Agrawal et al., 2024) and DeepSpeed-FastGen (Holmes et al., 2024) try to balance TTFT and TPOT by scheduling the *mixed* iterations.

We identify that existing serving systems fail to recognize and handle the bias and the variance of the latency distribution depicted in Fig. 1(a). The bias represents the gap between the SLO constraint margin on TTFT and that on TPOT, **featuring one of them having much lower attainment against its SLO**. The variance represents the inter-request variance against the SLO constraint margin, **with many requests failing the SLOs while others still having redundant budgets to spare**. The reason is that the existing designs schedule requests according to some fixed principles (*e.g.*, *prefill*-prioritized or *decode*-prioritized) but lack more fine-grained management of the trade-off between TTFT and TPOT, and the trade-off among different requests. Some previous works support the request-level priority, including FastServe (Wu et al., 2023) that uses preemptive scheduling to minimize the total latency by giving shorter requests a higher priority. Recent work (Fu et al., 2024) tries to predict the relative order among requests without knowing the accurate lengths, also following the shortest-job-first (SJF) scheduling principle. Another work (Sheng et al., 2024) defines the fairness between users and treats the requests from different users with different priorities. But those works decide the execution order of requests merely based on lengths or token numbers but are unaware of the SLOs, therefore lacking the ability to handle the trade-off among different requests to maximize the SLO attainment.

In this work, we propose a fine-grained scheduling framework to maximize the SLO attainment for LLM serving. Compared to existing works, **the fined-grained scheduling enables the changing of scheduling strategy at iteration level**, and hence tune the workload for each execution during the serving procedure. First, we model the scheduling workflow and build the design space to unleash the potential changes in the scheduling strategy. The core idea is to control the execution order for different requests and the workload size for each iteration.

Secondly, complex contentions exist between *prefill* and *decode* phases, and exist among different requests. At the phase level, *prefill* phase and *decode* phases have heavy con-

tention on the limited computation and memory resources, leading to TTFT and TPOT interference. At the request level, managing the scheduling priority among different requests to balance the latencies is non-trivial. To this end, we further propose a state-aware scheduling strategy, which monitors two kinds of states: the states from the single request perspective and the states from the systemic perspective, to support system-level strategic decisions to optimize request-level latency. Based on the states, we leverage the trade-off between TTFT and TPOT (see Fig. 2(a)), and the trade-off among different requests (see Fig. 2(b)). For each iteration, the request execution order and the workload size are derived by solving a constrained optimization problem. Thus, state-aware scheduling enables effective trade-offs as mentioned, further enhancing the SLO attainment from Fig. 1(a) to Fig. 1(b). As a result, the improved SLO attainment enables the system to serve more requests under certain SLO constraints.

This paper makes the following contributions:

- We formulate a fine-grained scheduling framework for LLM serving, which supports the adjustable scheduling of requests at the granularity of iterations. At each iteration, the framework enables the variation of the scheduling by controlling the request execution order and the workload size.
- We introduce a state-aware scheduling strategy, which is generated based on the monitoring of two kinds of states: the states from the perspectives of a single request and the system, respectively. Based on the monitored states, we optimize the scheduling strategy to balance the trade-offs between TTFT and TPOT and among different requests, to maximize the SLO attainment for the whole system.
- We implement SOLA and evaluate on various benchmarks. The results show that SOLA improves the SLO attainment by over 50%, and hence can serve up to 1.27× more requests compared to the existing designs.

2 Background and Related Works

In this section, we introduce the background knowledge involved in LLM serving, and the related works.

2.1 LLM Processing Phases

The mainstream LLMs (Meta, 2024; Zhang et al., 2022; QwenLM, 2024; Young et al., 2024) utilize the Transformer (Vaswani et al., 2017) architecture, which predicts the probability of a sequence of tokens $(x_1, \ldots, x_n)$ based on autoregressive decomposition (Bengio et al., 2000). The core of a Transformer-based model is the self-attention mechanism, which produces query, key, and value vectors for each token via linear projection, calculates the attention score by multiplying the query vector with all the preceding key vectors, and then computes the output as a weighted average of the value vectors. For the one-by-one token generation, the key and value vectors of earlier tokens (denoted as KV cache) are cached for future token generation. Thus, sequence processing with LLM is typically divided into the following two phases.

***Prefill* Phase.** *Prefill* phase acts as the initial execution for a request. The model receives a sequence of input tokens sent by the request and calculates the probability of the first new token. The model also produces the key vectors and value vectors, which are stored as KV cache.

***Decode* Phase.** At each iteration, the model processes the latest output token, calculating the probability of the next token while utilizing both KV cache and the newly computed key and value vectors. This phase continues until the generation of output tokens reaches a pre-defined maximum length or an end-of-sequence token is produced.

2.2 LLM Serving

In LLM serving, users send requests to the server. The serving system processes the request on the LLM and returns the output tokens to the users as responses. Commonly used performance metrics in LLM serving include TTFT, TPOT, throughput, and goodput. TTFT and TPOT measure the request-level latency perceived by users, while throughput and goodput measure the system-level performance. As mentioned in Sec. 1, TTFT and TPOT are both essential for user experience, and the agreement between service providers and users determines SLOs (*e.g.*, TTFT $\leq$ 0.5s, TPOT $\leq$ 0.1s) as the latency constraints. Goodput (Zhong et al., 2024) is a system-level metric that considers the latency constraints, and we employ goodput as the evaluation metric in this work.

2.3 Related Works

Fundamentals for LLM Serving. Continuous batching proposed by Orca (Yu et al., 2022) is the basic technique in LLM serving, which allows the finished requests to exit and the arrived requests to be batched for computation at each iteration, thereby beneficial for both system-level utilization and request-level latency. Current LLM serving systems widely apply the PagedAttention by vLLM (Kwon et al., 2023), managing the KV cache as pages. Such paged storage significantly removes memory fragmentation and improves memory utilization. Moreover, the efficient GPU kernels (Dao, 2023; Dao et al., 2023; NVIDIA, 2017b) benefit the hardware utilization and improve the serving performance. We adopt those fundamental techniques in SOLA.

LLM Scheduling. Considering the distinct *prefill* phase

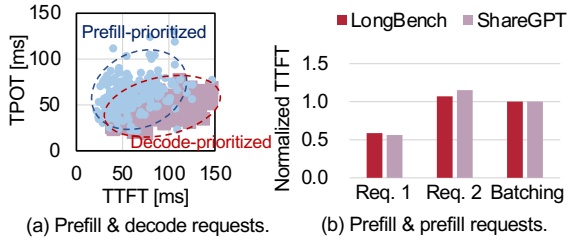


Figure 3. Different types of contention. (a) The prioritization of *prefill* requests stalls *decode* requests and vice versa. (b) Normalized TTFT comparison when batching two *prefill* requests or not.

and *decode* phase, the scheduling in LLM serving requires specific designs. vLLM (Kwon et al., 2023) by default prioritizes the *prefill* phase of new arrivals over the *decode* phases of the running requests. FasterTransformer (NVIDIA, 2017a) executes the *decode* phases of all the running requests without interruption of the incoming requests. Intuitively, such monotony in prioritization always benefits either TTFT or TPOT and harms the other, leading to the distribution bias as mentioned in Sec. 1. The *SplitFuse* method in Sarathi-Serve (Agrawal et al., 2024) and DeepSpeed-FastGen (Holmes et al., 2024) takes a step to balance TTFT and TPOT, via adjusting the stalls between two phases. But *SplitFuse* fails to ensure the bias between TTFT and TPOT is corrected. Besides, previous works (Wu et al., 2023; Sheng et al., 2024; Fu et al., 2024) discuss the prioritization among requests but do not consider the SLOs. But those works reveal only part of the scheduling design space, and are not aware of the request-level latency during scheduling, thus being sub-optimal in SLO attainment.

Disaggregated System Design. Recent works including Splitwise (Patel et al., 2023), TetriInfer (Hu et al., 2024), and DistServe (Zhong et al., 2024) demonstrate the efficiency of disaggregating the *prefill* and the *decode* phases of a request to different instances. ExeGPT (Oh et al., 2024) proposes different scheduling strategies involving assigning the execution of *prefill* and *decode* phases to different GPUs, and discusses the performance under the latency constraint. Those disaggregated systems demand more GPUs to deploy the service, necessitating high request rates to saturate the GPU capacity. This work mainly focuses on the system that locates *prefill* and *decode* phases on the same GPU.

3 Motivation

3.1 Coarse-Grained versus Fine-Grained Scheduling

3.1.1 Performance Comparison

Based on Fig. 1(a), we go one step further to analyze the disadvantages of coarse-grained scheduling in existing designs. In Fig. 1(a) about 35% of requests fail the TPOT SLO, and among them, there exist requests whose TTFTs are far lower than the constraint, **indicating a potential satisfaction for the SLO with the trade-offs between TTFT and TPOT**. Moreover, the distribution in Fig. 1(a) is highly decentralized with a significant variance among requests, **potentially indicating a better SLO attainment with the trade-off among different requests**. As outlined in Fig. 2, fine-grained scheduling enhances the SLO attainment by taking advantage of twofold trade-offs. From the perspective of a single request, fine-grained scheduling reduces the TPOT of the request while maintaining its TTFT within the SLO, refining the non-compliant request to meet both SLOs (see Fig. 2(a)). For requests with significant latency differences, fine-grained scheduling manages more resources for the request with large latency while assuring the SLO satisfaction of the request with short latency (see Fig. 2(b)). Therefore, compared to coarse-grained scheduling, fine-grained scheduling enables the twofold trade-offs and exploits the potential optimization for SLO attainment.

3.1.2 Trade-Off Analysis

We further discuss the ability of fine-grained scheduling to manage the twofold trade-offs described in Fig. 2. The processing of each request includes the *prefill* phase and the *decode* phase, with each comprising one or more iterations. For each request, TTFT measures the latency of waiting and the *prefill* phase, while TPOT measures the average latency during the *decode* phase. **The difficulty of scheduling under the SLOs arises from the latency being measured at the request level, whereas the scheduling occurs at the iteration level.** Existing works focus on the iteration-level designs but overlook the global observation of the request latency, thus failing to manage the twofold trade-offs in Fig. 2. To address the issue, the following three types of contention need careful handling.

- **Contention 1. *Prefill* and *Decode* Requests.** As shown in Fig. 3(a), compared to the *prefill*-prioritized situation, the prioritization for *decode* requests keeps the TPOT distribution lower but prolongs TTFT distribution. Therefore, the contention exists when the requests in one phase are prioritized while the requests in the other phase are stalled. Note that such contention has varying impacts on different requests, *e.g.*, inserting *prefill* requests causes a larger TPOT increase for the *decode* requests with shorter output lengths.
- **Contention 2. *Prefill* and *Prefill* Requests.** If the two requests (Req. 1 and Req. 2) in Fig. 3(b) are executed one by one, the prioritized request will gain a better TTFT (Req. 1) but the other one will incur additional waiting time (Req. 2). Batching two requests together leads to a same TTFT for both, and the TTFT is slightly lower than the original TTFT for the second request, if the GPU capacity is not saturated, which is

Table 1. Symbols in the scheduling framework at the i-th iteration.

Class	Symbol	Explanation
Setting	M	max KV cache memory
	T^{TTFT}	SLO of TTFT
	T^{TPOT}	SLO of TPOT
State (system)	m_i^{ratio}	KV cache memory used ratio
	$\boldsymbol{Q}_i^{\text{wait}}$	waiting request queue
	$\boldsymbol{Q}_i^{\text{run}}$	running request queue
	p_i^{TTFT}	ratio of real-time TTFT to SLO
	p_i^{TPOT}	ratio of real-time TPOT to SLO
	$\boldsymbol{D}_i$	output length distribution
State (request r)	$t_{i,r}^{\text{TTFT}}$	real-time TTFT
	$t_{i,r}^{\text{TPOT}}$	real-time TPOT
	$l_{i,r}^{\text{out}}$	generated output length
	$l_{i,r}^{\text{left}}$	predicted left length to run
	$k_{i,r}^{\text{new}}$	number of tokens to be added
	l_r^{in}	input length
Strategy	n_i	number of requests to run
	k_i	number of tokens to run
	$\mathcal{F}_i$	sorting func. for waiting requests
Cost Model	$\mathcal{C}_i^{\text{p}}$	cost model for *prefill* requests
	$\mathcal{C}_i^{\text{d}}$	cost model for *decode* requests

the situation in Fig. 3(b). The TTFT change caused by batching depends on the request input lengths and also the computational capacity of the system. Without batching, the execution order between Req. 1 and Req. 2 decides which request bears the waiting latency.

- **Contention 3. *Decode* and *Decode* Requests.** The contention among *decode* requests happens when the system memory is full, and some requests are preempted to spare space for continuing the *decode* iterations of the others. The preempted requests are either swapped to CPU or recomputed, thereby suffering a larger TPOT than the retained ones. One alternative solution is to early reduce the batch size for *decode* iterations based on the peak memory prediction, hence avoiding the memory overflow, which we will specify in Sec. 5.1. However, such an early reduction still forces part of the *decode* requests to wait. Retaining how many (workload size) and which (execution order) requests decides the TPOTs of all requests.

Considering that the system has hundreds of requests simultaneously, the scheduling strategy unchanged for different iterations fails to handle those contentions properly. Thus, fine-grained scheduling is necessitated to perform better trade-offs and optimize the system performance.

3.2 Challenges for Fine-Grained Scheduling

Performing fine-grained scheduling faces several challenges, and we list corresponding insights to uncover the approach to building an efficient fine-grained scheduling framework.

Lack of design space. Previous works reveal only part of the design space in the scheduling procedure (Agrawal et al., 2024; Wu et al., 2023; Fu et al., 2024), and the scheduling is always hard-coded (Kwon et al., 2023; ModelTC, 2024), thus failing to vary with iterations. To address the issue, we propose a fine-grained framework with comprehensive control of the request execution order and the workload size, enabling the strategy to change at every iteration.

How to control request-level latency via iteration-level behavior. As mentioned in Sec. 1, scheduling plays a role in managing the system resources for serving different requests. However, the existing scheduling design fails to be aware of and control the latency of a certain request but merely follows a tendency toward the objective. To tackle the challenge, we propose the method of state-aware scheduling, monitoring both the request states and the system states to guide the fine-grained adjustment.

How to achieve an effective trade-off between TTFT and TPOT. The current scheduling strategies including *SplitFuse* fail to reduce TTFT or TPOT while moderately maintaining the other within the constraints, and it is essential to design a scheduling strategy that considers both TTFT and TPOT during serving. Therefore, we propose to solve a constrained optimization problem to generate the strategy at each iteration, taking both SLOs into account.

4 SOLA: State-Aware Scheduling

In this section, we present the proposed fine-grained scheduling framework and the state-aware method in detail. First, we model the scheduling workflow in LLM serving and clarify the orthogonal variables that a scheduling strategy is designed on. The modeling enables the expression of existing scheduling strategies and further uncovers the extended space to support fine-grained scheduling. Then we formulate the constrained optimization problems based on the monitored states and generate the optimized strategy by solving the optimization problem for each iteration.

4.1 Design Space and Modeling

The variables in Table. 1 model the scheduling process in LLM serving. As outlined in Table 1, the **Setting** class comprises the deployment settings predetermined by users or constrained by the hardware platform. These settings are fixed once the LLM is deployed for a specific scenario. The variables within the **State** class change during the serving process, reflecting real-time updates in memory, sequence length, and request latency. The variables in the **Strategy** collectively form the design space for scheduling strategies. Besides, SOLA utilizes **Cost Model** for latency prediction, which is detailed in Sec. 5.2.

Algorithm 1 Scheduling at the i-th iteration.

Input: M, $\boldsymbol{Q}_i^{\text{wait}}$, $\boldsymbol{Q}_i^{\text{run}}$, m_i^{ratio}, n_i, k_i, $\mathcal{F}_i$

1: Initialize $\boldsymbol{Q}_i^{\text{run}} \leftarrow \emptyset$
2: **if** get_req_num($\boldsymbol{Q}_i^{\text{wait}}$) $= 0$ **then**
3: Return ($\boldsymbol{Q}_i^{\text{wait}}$, $\boldsymbol{Q}_i^{\text{run}}$)
4: **end if**
5: $\boldsymbol{Q}_i^{\text{wait}} \leftarrow \mathcal{F}_i(\boldsymbol{Q}_i^{\text{wait}})$
6: **for** $r \in \boldsymbol{Q}_i^{\text{wait}}$ **do**
7: $m^{\text{peak}} \leftarrow$ cal_peak_mem($\boldsymbol{Q}_i^{\text{run}} \cup r$)
8: **if** $m^{\text{peak}} \geq M \times (1 - m_i^{\text{ratio}})$ **then**
9: **continue**
10: **end if**
11: $\boldsymbol{Q}_i^{\text{run}} \leftarrow \boldsymbol{Q}_i^{\text{run}}$.push_req_with_token_num($r, k_{i,r}^{\text{new}}$)
12: $\boldsymbol{Q}_i^{\text{wait}} \leftarrow \boldsymbol{Q}_i^{\text{wait}}$.pop_req_with_token_num($r, k_{i,r}^{\text{new}}$)
13: **if** get_req_num($\boldsymbol{Q}_i^{\text{run}}$) $\geq n_i$ **then**
14: **break**
15: **end if**
16: **if** get_token_num($\boldsymbol{Q}_i^{\text{run}}$) $\geq k_i$ **then**
17: **break**
18: **end if**
19: **end for**
20: Return ($\boldsymbol{Q}_i^{\text{wait}}$, $\boldsymbol{Q}_i^{\text{run}}$)

4.1.1 *Design Space*

Following the insight of scheduling based on the request execution order and workload size control, for the i-th iteration, we employ the function $\mathcal{F}_i$ to control execution order and the variables k_i and n_i to control workload size. The strategy with those variables is characterized as follows: (1) $\mathcal{F}_i$ decides the order in which the waiting requests are added for execution. (2) For the i-th iteration, the maximum number of requests for execution is defined as n_i, and the maximum number of tokens for execution is denoted as k_i, having $n_i = |\boldsymbol{Q}_i^{\text{run}}|$, $k_i = \sum_{r \in \boldsymbol{Q}_i^{\text{run}}} (k_{i,r}^{\text{new}})$. Note that $k_{i,r}^{\text{new}} = 1$ for *decode* requests, and $k_{i,r}^{\text{new}} \leq l_r^{\text{in}}$ for *prefill* requests considering being chunked. Importantly, n_i, k_i and $\mathcal{F}_i$ are treated as variables that evolve with each iteration, enabling the strategy to adapt to the twofold trade-offs (see Sec. 3.1) during the serving process.

4.1.2 *Scheduling Algorithm*

At the i-th iteration, the scheduler selectively transfers requests from the waiting queue $\boldsymbol{Q}_i^{\text{wait}}$ to the running queue $\boldsymbol{Q}_i^{\text{run}}$ for execution. The waiting queue $\boldsymbol{Q}_i^{\text{wait}}$ and the running queue $\boldsymbol{Q}_i^{\text{run}}$ can include both the *prefill* requests and the *decode* requests. **Setting**, **State**, and **Strategy** are the inputs for scheduling, and the details are outlined in Algorithm 1. For simplification, preemption and swapping are excluded, as both introduce extra latency and are generally avoidable through strategic design. Initially, the running request queue $\boldsymbol{Q}_i^{\text{run}}$ is initialized as empty. If there is no request waiting, there will be no execution. Otherwise, each request in the waiting queue $\boldsymbol{Q}_i^{\text{wait}}$ is evaluated individually as detailed in the for-loop starting at line 6. Before this evaluation, $\boldsymbol{Q}_i^{\text{wait}}$ is sorted to prioritize certain requests. During validation, based on the peak memory prediction, the scheduler checks whether sufficient memory is available for accommodating the candidate request (see details in Sec. 5.1). Then it verifies if the number of requests in $\boldsymbol{Q}_i^{\text{run}}$ surpasses the limit n_i or the number of tokens exceeds k_i set by the strategy. The pushing and popping operations at lines 11 and 12 are adjusted for the chunked *prefill* requests. At the end of the algorithm, the requests in $\boldsymbol{Q}_i^{\text{run}}$ are scheduled for execution.

The modeling enables the expression of existing scheduling strategies. Take the *SplitFuse* method implemented in vLLM (Kwon et al., 2023) as an example, n_i is 256 and k_i is 512 under the default setting. For the sorting function $\mathcal{F}_i$, *decode* requests are prioritized, and the requests at the same phase are sorted in the first-come-first-serve (FCFS) order.

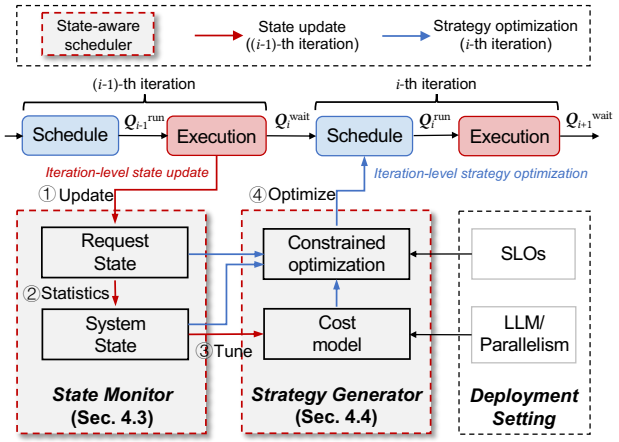


Figure 4. Overview of state-aware scheduling. ① The states of all requests are updated when an iteration (*prefill*, *decode*, or *mixed*) finishes. ② The statistics of request states are collected to form the system states. ③ The cost models are tuned based on the system states. ④ The strategy optimizer derives the scheduling strategy based on the states, the cost models, and the given SLOs.

4.2 System Overview

As illustrated in Fig. 4, the workflow of SOLA includes two parts: *state monitor* and *strategy generator*. The state monitor is responsible for updating the states of each request after a finished iteration and derives the statistical information, *i.e.*, the system states, from all the request states. The cost models in the strategy generator predict the processing latency and are tuned by the system states at every iteration to achieve high prediction accuracy. The predicted latency enables the strategy optimizer to generate the optimal strategy for scheduling at the next iteration.

Table 2. The conversion of constrained optimization problems.

Condition (or)	Constrained Optimization
(1) $1 > p_i^{\text{TPOT}} > p_i^{\text{TTFT}}$ (2) $p_i^{\text{TPOT}} > 1 > p_i^{\text{TTFT}}$	$\min \max_r(t_{i,r}^{\text{TPOT}})$ *s.t.* $\max_r(t_{i,r}^{\text{TTFT}}) \leq T^{\text{TTFT}}$
(1) $1 > p_i^{\text{TTFT}} > p_i^{\text{TPOT}}$ (2) $p_i^{\text{TTFT}} > 1 > p_i^{\text{TPOT}}$	$\min \max_r(t_{i,r}^{\text{TTFT}})$ *s.t.* $\max_r(t_{i,r}^{\text{TPOT}}) \leq T^{\text{TPOT}}$

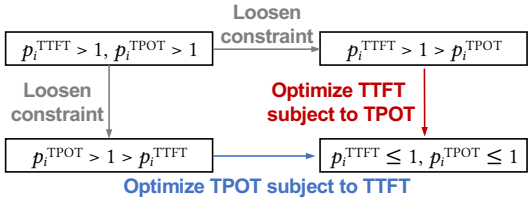


Figure 5. State transition of the system latency statistic. The target transition is towards the state where both TTFT and TPOT satisfy the constraints (lower right corner).

4.3 State Monitor

The state monitor in SOLA updates the following states when an iteration finishes. **(1) Request latency**. The execution time of the latest iteration is added to the total latency, and the real-time TTFT and TPOT (*i.e.*, $t_{i,r}^{\text{TTFT}}$ and $t_{i,r}^{\text{TPOT}}$) of each request are calculated. **(2) System latency statistic**. To obtain the systemic view of latency fulfillment, we define $p_i^{\text{TTFT/TPOT}} = \max_r(t_{i,r}^{\text{TTFT/TPOT}})/T^{\text{TTFT/TPOT}}$. The real-time attainment of TTFT/TPOT SLO is 100% if $p_i^{\text{TTFT/TPOT}} \leq 1$. SOLA handles the trade-off between TTFT and TPOT based on $p_i^{\text{TTFT/TPOT}}$, which is detailed in Sec. 4.4.1. **(3) Request length.** The state monitor records the generated length $l_{i,r}^{\text{out}}$ and the remaining output length $l_{i,r}^{\text{left}}$ for each request. Since the output length is unknown before a request is finished, $l_{i,r}^{\text{left}}$ is a predicted value. Specifically, SOLA estimates $l_{i,r}^{\text{left}}$ based on the collected output length distribution. **(4) System length distribution.** The output length distribution $\boldsymbol{D_i}$ is collected for length prediction. The finished requests are classified by the ranges of the input length and the maximum output length, and the output lengths are collected separately for each range combination to form a distribution. **(5) System memory usage.** For the convenience of memory management, the ratio of the used KV cache memory at the current iteration m_i^{ratio} is recorded.

4.4 Strategy Generator

Using the monitored states, the strategy generator generates the scheduling strategy based on the formulation of two constrained optimization problems, which consider both TTFT and TPOT SLOs via constraint and optimization. At each iteration, the strategy generator first identifies which constrained optimization problem to solve, *i.e.* either optimize TPOT subject to TTFT, or optimize TTFT subject to TPOT. Then, the strategy generator derives request execution order ($\mathcal{F}_i$) and workload size (n_i, k_i) for scheduling by solving one of the constrained optimization problems.

4.4.1 *Constrained Optimization*

As discussed in Sec. 3, the strategies for optimizing TTFT and TPOT interfere with each other. To address the challenge, we formulate constrained optimization problems with dynamic conversion between the constraint and the optimization objective, to be aware of both SLOs. As shown in Table. 2, the conversion is controlled based on the real-time TTFT/TPOT of the requests in the system. We optimize the less fulfilled latency (TTFT or TPOT) respective to its SLO. However, when both fail to satisfy the SLOs, the constraint is supposed to be loosened. As outlined in Fig. 5, we loosen the constraint until one of the latencies satisfies the constraint, so that the constrained optimization still works. To loosen the constraint, suppose that there are $a\%$ (a<100) of the requests satisfying the constraint, then the maximum operations in Sec. 4.3, the following Eq. 1 and Eq. 2 are correspondingly modified to $a\%$-level maximum. Then, we employ hierarchical prioritization to control request execution order and make the final decision on the scheduled requests according to the constrained workload size.

4.4.2 *Hierarchical Prioritization ($\mathcal{F}_i$)*

As outlined in Fig. 6, we first decide the execution order between the two phases. We prioritize *prefill* requests when ***optimizing* TTFT *subject to* TPOT**, otherwise prioritize *decode* requests when ***optimizing* TPOT *subject to* TTFT**. Based on that, we apply a sliding window over iterations to compare the fulfillment between TTFT and TPOT, to mitigate the influence of sudden changes.

The second level of prioritization focuses on the execution order among requests at the same phase. For each *prefill* request, the TTFT comprises the waiting and *prefill* latencies. Thus, instead of the first-come-first-serve (FCFS) order, all the *prefill* requests in waiting are sorted to a descending order of $(t_{i,r}^{\text{TTFT}} + \mathcal{C}_i^{\text{p}}(r))$, where $\mathcal{C}_i^{\text{p}}(r)$ is the predicted future *prefill* time based on the cost model, and the sum is the predicted TTFT of request r. Considering that TPOT measures the average latency per output token (except for the first token), its output length significantly impacts the TPOT of each request. Therefore, all the *decode* requests in waiting are sorted to a descending order of $(t_{i,r}^{\text{TPOT}} \times l_{i,r}^{\text{out}} + \mathcal{C}_i^{\text{d}}(\boldsymbol{Q}_{i-1}^{\text{run}}) \times l_{i,r}^{\text{left}})/(l_{i,r}^{\text{out}} + l_{i,r}^{\text{left}})$, which represents the predicted TPOT of request r. Since the output length is unknown before a request is finished, we predict $l_{i,r}^{\text{left}}$ based on the system length distribution (see Sec. 4.3). The method of length prediction is not the main focus of this work, and there are works exploring the design to improve

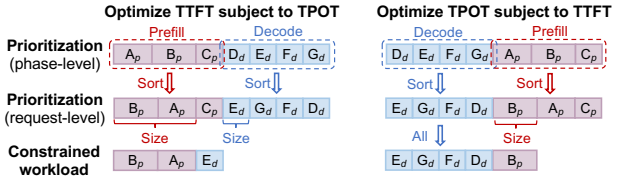


Figure 6. Prioritization and workload control of SOLA. A-G are different requests. Subscript *p* denotes a *prefill* iteration and subscript *d* denotes a *decode* iteration, respectively.

the prediction accuracy (Wu et al., 2023; Hu et al., 2024). We illustrate the details of the cost models $\mathcal{C}_i^{\text{p}}$ and $\mathcal{C}_i^{\text{d}}$ in Sec. 5.2.

4.4.3 *Constrained Workload* (n_i, k_i)

Note that optimizing TTFT or TPOT does not mean reducing it toward the SLO at any cost. Such extreme adjustment leads to the oscillation between TTFT and TPOT. **The core idea lies in ensuring the minimum resources to meet the constraint for TTFT or TPOT while doing the best to optimize the other one**. Thus, we elaborate on the constrained workload design from the perspectives of ensuring the constraint and trying the best for optimization. As shown in Fig. 6, the workload controls of the two constrained optimization problems are discussed separately.

***Optimize* TTFT *subject to* TPOT.** The *prefill* requests are prioritized but the execution number is constrained. Considering that the insertion of the *prefill* requests increases TPOT, the workload of the *prefill* requests is constrained to meet the TPOT SLO by

$$\max_{\{r|t_{i,r}^{\text{TPOT}}>0, r\in \boldsymbol{Q}_i^{\text{wait}}\}} \left(t_{i,r}^{\text{TPOT}} + \frac{\mathcal{C}_i^{\text{p}}(\boldsymbol{Q}_i^{\text{run}})}{l_{i,r}^{\text{out}}}\right) \leq T^{\text{TPOT}}. \quad (1)$$

The chunking of *prefill* requests allows the number of added tokens to be controllable (see lines 19 and 20 in Algorithm 1). We opt to set k_i instead of n_i to achieve more fine-grained control. Leveraging the tile-quantization effect on hardware (Agrawal et al., 2024), the token number limitation k_i is set to be the maximum multiples of the tile size (*e.g.*, 128 for A100 GPUs) that satisfies Eq. 1. If all the *prefill* requests in $\boldsymbol{Q}_i^{\text{wait}}$ are added to $\boldsymbol{Q}_i^{\text{run}}$ without breaking Eq. 1, the *decode* request is added one by one until the total token number is k_i. Meanwhile, n_i is set to an invalid value.

***Optimize* TPOT *subject to* TTFT.** All the *decode* requests are added for execution first, and the number of the *prefill* requests in waiting is constrained by

$$\max_{r\in \boldsymbol{Q}_i^{\text{wait}}} \left(t_{i,r}^{\text{TTFT}} + \mathcal{C}_i^{\text{p}}(\boldsymbol{Q}_i^{\text{run}}) + \mathcal{C}_i^{\text{d}}(\boldsymbol{Q}_i^{\text{run}}) + \mathcal{C}_i^{\text{p}}(r)\right) \leq T^{\text{TTFT}}, \quad (2)$$

where $t_{i,r}^{\text{TTFT}}$ is the existing TTFT of request r, the sum of $\mathcal{C}_i^{\text{p}}(\boldsymbol{Q}_i^{\text{run}})$ and $\mathcal{C}_i^{\text{d}}(\boldsymbol{Q}_i^{\text{run}})$ is the predicted latency of this iteration, and $\mathcal{C}_i^{\text{p}}(r)$ is the predicted *prefill* latency of request r. Here we use n_i rather than k_i to control the workload size, as TTFT is measured only if all the input tokens of the request are processed. Thus, n_i is set to be the minimum number that satisfies Eq. 2, and k_i is given an invalid value.

5 Implementation

We implement SOLA as a standalone framework in Python code. SOLA contains interfaces to integrate into serving systems and replace the original schedulers. Currently, SOLA supports vLLM (Kwon et al., 2023) as the backend, which is an open-source serving system, with techniques including continuous batching, paged memory management, optimized operators, and tensor-parallel distributed inference.

5.1 Peak Memory Prediction

Preemption happens when the memory allocation for the next iteration fails, and the low-priority requests are swapped out to the CPU or handled later with recomputation, to spare memory for the next token generation of other requests. Such preemption overhead grows with the increasing workload, leading to an intolerable overhead. To address the issue, LightLLM (ModelTC, 2024) proposes to predict the peak memory consumption of a certain request upon its arrival, preventing the potential occurrence of preemption. Following LightLLM, we adopt the design of the peak memory prediction mechanism to evaluate the impact of a new adding request on memory consumption to determine whether to process the new request (see line 9 in Algorithm 1), ensuring the future allocation is available.

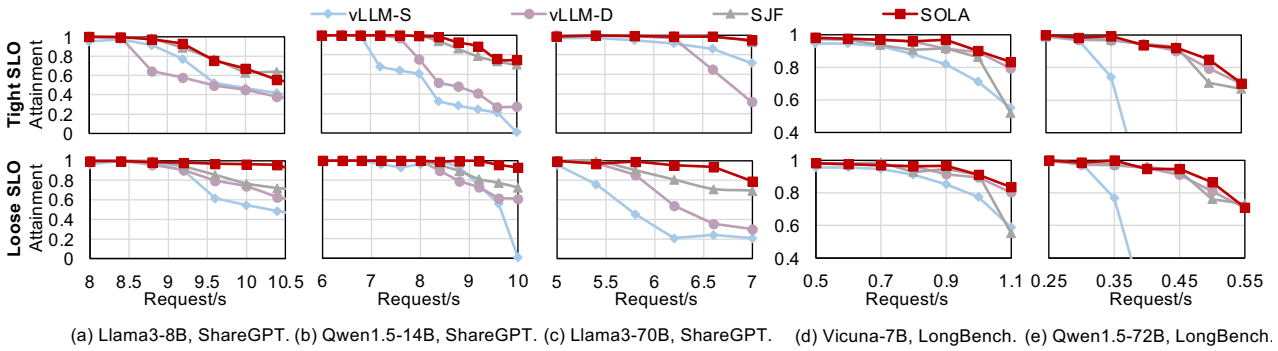
5.2 Cost Model $(\mathcal{C}^{\text{p}}, \mathcal{C}^{\text{d}})$

Given certain requests for execution, the cost models are introduced to predict the processing latency. Due to the difference in dataflow, we respectively build cost model $\mathcal{C}^{\text{p}}$ and $\mathcal{C}^{\text{d}}$ for *prefill* and *decode* requests. As shown in Fig. 4, the cost models are initialized using profiling data from off-line inference. We initialize the cost model as the polynomial of input request length, KV cache length, batch size, etc.

$$\mathcal{C}^{\text{p}} = a_0 \sum_r l_r^{\text{has}} l_r^{\text{in}} + b_0 \sum_r (l_r^{\text{in}})^2 + c_0 \sum_r l_r^{\text{in}} + d_0, \quad (3)$$

$$\mathcal{C}^{\text{d}} = a_1 \sum_r 1 + b_1 \sum_r (l_r^{\text{has}}) + c_1, \quad (4)$$

where $a_0, .., d_0, a_1, .., c_1$ are fitting parameters. In Eq. 3, we introduce l_r^{has} to denote KV cache length in case the *prefill* request r is chunked. $\mathcal{C}^{\text{p}}$ mainly considers the floating point operations (FLOPs) of the linear and attention operators under request batching, while $\mathcal{C}^{\text{d}}$ measures the memory access in the dominated operators.



(a) Llama3-8B, ShareGPT. (b) Qwen1.5-14B, ShareGPT. (c) Llama3-70B, ShareGPT. (d) Vicuna-7B, LongBench. (e) Qwen1.5-72B, LongBench.

Figure 7. End-to-end SLO attainment comparison under tight and loose SLOs.

Table 3. Benchmarks and SLO requirements in evaluation.

Model	Parallelism	SLOs (TTFT/TPOT) Tight	Loose	Dataset
Llama3-8B	TP=1	0.4/0.15s	0.6/0.225s	ShareGPT
Llama3-70B	TP=4	0.5/0.2s	1.0/0.4s	ShareGPT
Qwen1.5-14B	TP=2	0.8/0.2s	1.2/0.3s	ShareGPT
Vicuna1.5-7B-16k	TP=1	2.55/0.16s	3.825/0.24s	LongBench
Qwen1.5-72B	TP=4	2.425/0.2s	4.85/0.4s	LongBench

Leveraging the continuity in serving procedure (*e.g.*, the trend of the KV cache length variation), for the i-th iteration, we employ a scaling ratio γ_i to adjust the cost model linearly. The scaling ratio is tuned by the real-time latency from the state monitor as follows.

$$\gamma_i = \alpha \frac{t(r)}{\mathcal{C}_0^{\text{p/d}}(r)} + (1-\alpha)\gamma_{i-1},\ \mathcal{C}_i^{\text{p/d}} = \gamma_i \mathcal{C}_0^{\text{p/d}}, \quad (5)$$

where $t(r)$ is the recorded latency of request r, and α is the confidence of the tuning result. We sum up the predicted latencies from two cost models for the *mixed* iteration, and the scaling ratio γ_i implicitly reflects the batching effect.

6 Evaluation

We evaluate SOLA on various benchmarks (see Table 3) and the evaluation demonstrates that SOLA improves input request rate by 1.08-1.27 $\times$ and 1.04-1.11$\times$ on average while staying within the latency SLOs for 90% and 99% of the requests, respectively. Meanwhile, SOLA introduces merely 0.40%-0.45% scheduling overhead.

6.1 Setup

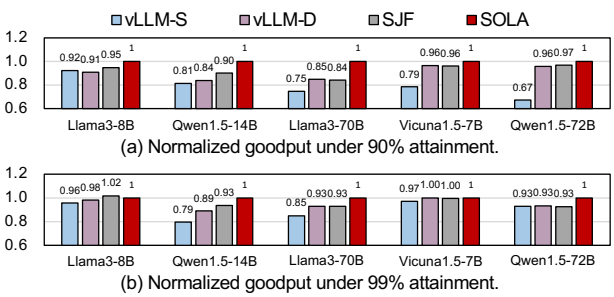
Testbed. We conduct experiments on a server with eight pairwise connected NVIDIA A100 80GB SXM4 GPUs, and the corresponding software environment includes CUDA 12.1 (Nvidia, 2024), NCCL 2.18 (Awan et al., 2016), PyTorch 2.3.0 (Paszke et al., 2019).

Benchmark. As listed in Table. 3, we evaluate SOLA under various scenarios including with open-source LLMs of different sizes. For each scenario, we sample from suitable datasets and generate request arrival times using Poisson distribution to emulate the real-world workloads.

- **Chatbot**. We sample 2000 requests from the ShareGPT (Sharegpt, 2023) dataset and test on the latest Llama3 (Meta, 2024) and Qwen1.5 (QwenLM, 2024) series models.
- **Long-text Understanding**. We use all 832 requests from the LongBench (Bai et al., 2023) dataset for evaluation. LongBench has an input length of 3.3k on average, and we test with the Vicuna1.5-7B-16k (Chiang et al., 2023) fined-tuned from Llama2 (Touvron et al., 2023) and Qwen1.5-72B (QwenLM, 2024) models.

Metric. We set 10$\times$/15$\times$ of the single request latency to be the tight/loose SLOs for 7B-14B models. Considering the batch size shrinks with larger models, we set 5$\times$/10$\times$ of the single request latency to be the tight/loose SLOs for 70B-72B models. The detailed numbers are listed in Table 3. To measure the sensitivity against SLO variation, we also conduct experiments of other SLO settings in Sec. 6.5. We measure the SLO attainment as the major metric under increasing input request rates and compare the goodput (*i.e.*, the maximum input request rate) under 90% and 99% SLO attainment.

Baseline. SOLA is integrated into the SOTA serving system vLLM (Kwon et al., 2023) (v0.4.2) to demonstrate the effectiveness. We compare SOLA with vLLM-D and vLLM-S strategies, where vLLM-D denotes the default strategy that prioritizes the *prefill* requests and vLLM-S denotes the integration of the *SplitFuse* method. We use vLLM-S instead of the original implementation from Sarathi-Serve (Agrawal et al., 2024), as Sarathi-Serve is an early fork from vLLM. Such comparison avoids the impact of irrelevant factors such as operator-level differences. Since the chunk size (*i.e.*, the maximum token number) impacts the performance of the *SplitFuse* method, we vary some choices to select the best one for evaluation. Searching for an optimal chunk size in advance is non-trivial, as discussed in the work (Cheng et al.,



(a) Normalized goodput under 90% attainment.

(b) Normalized goodput under 99% attainment.

Figure 8. Normalized goodput comparison.

Table 4. The scheduling overhead ratio comparison.

Method	Llama3-70B+ShareGPT	Vicuna1.5-7B+LongBench
vLLM-S	1.17%	0.85%
vLLM-D	1.38% (+0.19%)	1.05% (+0.20%)
SJF	1.39% (+0.20%)	1.03% (+0.18%)
SOLA	**1.62% (+0.45%)**	**1.28% (+0.40%)**

2024). For this work, to compare with vLLM-S, we set the chunk size to 1024 for chatbot and 4096 for long-context understanding. We also implement the shortest-job-first (SJF) principle adopted in previous works (Wu et al., 2023; Fu et al., 2024) on top of vLLM (v0.4.2) as a baseline, which is denoted as SJF in the results.

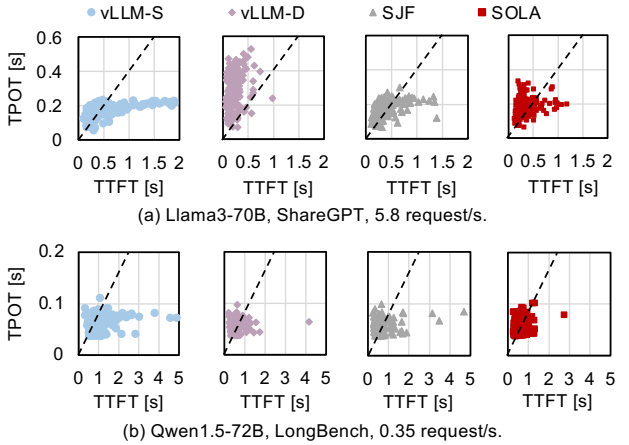
6.2 End-to-End Performance

6.2.1 *SLO Attainment Comparison*

Fig. 7 illustrates the end-to-end serving performance of SOLA compared to baselines. On most benchmarks, SOLA achieves better SLO attainments and supports higher input request rates under the given SLOs. On average, SOLA serves 1.27×, 1.11×, and 1.08× more requests than vLLM-S, vLLM-D, and SJF, while staying within the SLO constraints for 90% of the requests (Fig. 8(a)). Moreover, considering 99% of the requests satisfying the SLOs, SOLA serves 1.11×, 1.06×, and 1.04× more requests than vLLM-S, vLLM-D, and SJF (Fig. 8(b)), respectively. It is worth noting that vLLM-S performs poorly with long context, as the limit of chunk size prevents the long inputs from immediate handling. The *prefill*-prioritized principle of vLLM-D is well suited for the LongBench dataset, and SOLA emulates such behavior, together with the request execution order control, to gain a slightly higher attainment. The performance of SJF is close to SOLA under the tight SLO, as it also performs trade-offs among different requests, and when SLO is tight, the space for performing trade-offs between TTFT and TPOT is little.

6.2.2 *TTFT-TPOT Plane Visualization*

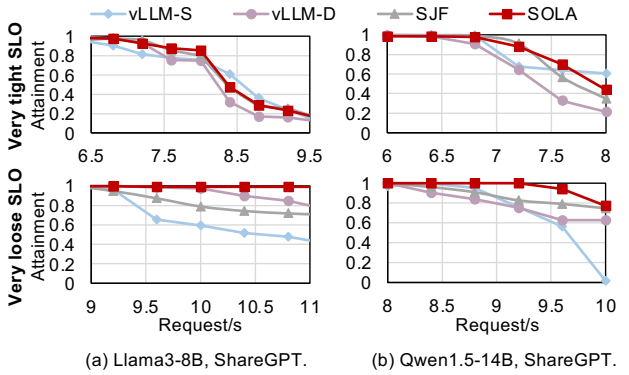
The latency distribution helps us explore the underlying mechanism in SOLA. First, the distribution center of SOLA lying close to the dotted line represents an effective trade-off between TTFT and TPOT, which removes the bias in



(a) Llama3-70B, ShareGPT, 5.8 request/s.

(b) Qwen1.5-72B, LongBench, 0.35 request/s.

Figure 9. Latency distribution comparison. The dotted line is the linear scaling of the single request latency.



(a) Llama3-8B, ShareGPT. (b) Qwen1.5-14B, ShareGPT.

Figure 10. Performance against SLO variation.

Fig. 2(a). Besides, the distribution of SOLA maintains collected with low variance, which arises from the trade-off between low-latency and high-latency requests.

6.3 Ablation Studies

6.3.1 *Benefit Breakdown*

On the LongBench dataset, the memory is fully utilized and the prediction mechanism becomes effective. Both the state-aware scheduling and the peak memory prediction mechanism (see Sec. 5.1) contribute to the SLO attainment. *E.g.*, when serving Qwen1.5-72B on LongBench, the attainment of SOLA is 96.5% under 0.45 request/s. The attainment degrades to 93.5% without state-aware scheduling and 93.0% without peak memory prediction.

6.3.2 *Cost Model Accuracy*

To observe the accuracy of the cost models (see Sec. 5.2), we record each iteration's predicted latency and real latency. The average absolute errors are 4.98%-5.92% for all the benchmarks in Table. 3. We also test the accuracy on the arXiv (Cohan et al., 2018) dataset with extremely long inputs, and the average absolute error is 8.91%.

6.4 Overhead

We measure the scheduling overhead by calculating the ratio of the scheduling duration to the whole serving duration. To reduce the scheduling overhead, we use a memory threshold to apply the memory prediction computation only when necessary. Moreover, we optimize the statistical procedure in the state monitor and use the momentum update to avoid repeated matrix multiplications in memory prediction. As shown in Table 4, the scheduling overhead of SOLA is negligible, leading to merely 0.45% and 0.40% overhead increases with the heavier workload (Llama3-70B & ShareGPT) and the lighter workload (Vicuna1.5-7B & LongBench), respectively. In fact, the overhead mostly comes from preparing the structured inputs instead of the scheduling logic.

6.5 Sensitivity against SLO Variation

For Llama3-8B and Qwen1.5-14B, we set 5× and 20× of the single request latency to be the very tight and very loose SLOs, respectively. The results under those SLOs are depicted in Fig. 10. SOLA achieves similar performance with the best under the very tight SLO, and significantly outperforms others under the very loose SLO, which is consistent with the intuition that SOLA gains advantages with a looser SLO, as there is more space for trade-offs.

7 Conclusion

In this paper, we introduce SOLA, a fine-grained scheduling framework to optimize the SLO attainment in LLM serving. Based on the formulated design space that enables the flexible control of request execution order and workload size, SOLA applies a state-aware mechanism to optimize the scheduling strategy at each iteration. Such fine-grained scheduling allows SOLA to manage the twofold trade-offs at the phase level and the request level, further enhancing the SLO attainment and serving more requests.

References

Adiwardana, D., Luong, M.-T., So, D. R., Hall, J., Fiedel, N., Thoppilan, R., Yang, Z., Kulshreshtha, A., Nemade, G., Lu, Y., and Le, Q. V. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020. URL https://arxiv.org/abs/2001.09977.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., and Zheng, E. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.

Awan, A. A., Hamidouche, K., Venkatesh, A., and Panda, D. K. Efficient large message broadcast using nccl and cuda-aware mpi for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pp. 15–22, 2016.

Bai, Y., Lv, X., Zhang, J., Lyu, H., Tang, J., Huang, Z., Du, Z., Liu, X., Zeng, A., Hou, L., Dong, Y., Tang, J., and Li, J. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Bengio, Y., Ducharme, R., and Vincent, P. A neural probabilistic language model. In *Advances in neural information processing systems*, 2000.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020. URL https://arxiv.org/abs/2005.14165.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023. URL https://arxiv.org/abs/2303.12712.

Cheng, K., Wang, Z., Hu, W., Yang, T., Li, J., and Zhang, S. Towards slo-optimized llm serving via automatic inference engine tuning. *arXiv preprint arXiv:2408.04323*, 2024.

Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*, 2023.

Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., and Goharian, N. A discourse-aware attention model for abstractive summarization of long documents. *arXiv preprint arXiv:1804.05685*, 2018.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flash-decoding for long-context inference. [Online], 2023. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. URL https://arxiv.org/abs/2002.08155.

Fu, Y., Zhu, S., Su, R., Qiao, A., Stoica, I., and Zhang, H. Efficient llm scheduling by learning to rank. *arXiv preprint arXiv:2408.15792*, 2024.

Guu, K., Lee, K., Tung, Z., Pasupat, P., and Chang, M.-W. Retrieval augmented language model pre-training. *arXiv preprint arXiv:2002.08909*, 2020. URL https://arxiv.org/abs/2002.08909.

Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., and He, Y. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

Hong, K., Dai, G., Xu, J., Mao, Q., Li, X., Liu, J., Chen, K., Dong, Y., and Wang, Y. Flashdecoding++: Faster large language model inference on gpus, 2024.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., Sun, N., and Shan, Y. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

Izacard, G. and Grave, E. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2020. URL https://arxiv.org/abs/2007.01282.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020. URL https://arxiv.org/abs/2005.11401.

Meta. Introducing meta llama 3: The most capable openly available llm to date, April 2024. URL https://ai.meta.com/blog/meta-llama-3/.

ModelTC. Lightllm. [Online], February 2024. https://github.com/ModelTC/lightllm.

NVIDIA. Fastertransformer: About transformer related optimization, including bert, gpt. [Online], 2017a. https://github.com/NVIDIA/FasterTransformer.

NVIDIA. cublas: Basic linear algebra on nvidia gpus. [Online], 2017b. https://developer.nvidia.com/cublas.

Nvidia. Cuda toolkit. [Online], June 2024. https://developer.nvidia.com/cuda-toolkit.

Oh, H., Kim, K., Kim, J., Kim, S., Lee, J., Chang, D.-s., and Seo, J. 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*, pp. 369–384, 2024.

OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. URL https://arxiv.org/abs/2303.08774.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Lang, J., Christopher, K., et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022. URL https://arxiv.org/abs/2203.02155.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Patel, P., Choukse, E., Zhang, C., Shah, A., Íñigo Goiri, and Maleki, S. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

QwenLM. Introducing qwen1.5. [Online], February 2024. https://qwenlm.github.io/blog/qwen1.5.

Roller, S., Dinan, E., Goyal, N., Ju, D., Williamson, M., Liu, Y., Xu, J., Ott, M., Shuster, K., Smith, E. M., Boureau, Y.-L., and Weston, J. Recipes for building an open-domain chatbot. *arXiv preprint arXiv:2004.13637*, 2021. URL https://arxiv.org/abs/2004.13637.

Sharegpt. Sharegpt. [Online], 2023. https://sharegpt.com.

Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. Fairness in serving large language models, 2024.

Svyatkovskiy, A., Deng, S., Fu, S., and Sundaresan, N. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*, 2020. URL https://arxiv.org/abs/2005.08025.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaidya, N., Oh, F., and Comly, N. Optimizing inference on large language models with nvidia tensorrt-llm, now publicly available. [Online], 2023. https://github.com/NVIDIA/TensorRT-LLM.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, 2017.

Wu, B., Zhong, Y., Zhang, Z., Liu, S., Liu, F., Sun, Y., Huang, G., Liu, X., and Jin, X. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

Young, A., Chen, B., Li, C., Huang, C., Zhang, G., Zhang, G., Li, H., Zhu, J., Chen, J., Chang, J., et al. Yi: Open foundation models by 01. ai. *arXiv preprint arXiv:2403.04652*, 2024.

Yu, G.-I. et al. Orca: A distributed serving system for transformer-based generative models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 521–538, 2022.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models, 2022.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Yuan, Z., Li, X., Yan, S., Dai, G., Zhang, X.-P., Dong, Y., and Wang, Y. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024. URL https://arxiv.org/abs/2404.14294.