USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

JITServe: SLO-aware LLM Serving with Imprecise Request Information

Wei Zhang, Zhiyu Wu, and Yi Mu, *University of Illinois, Urbana-Champaign;*
Rui Ning, *unaffiliated;* Banruo Liu, *University of Illinois Urbana-Champaign;*
Nikhil Sarda, *Google;* Myungjin Lee, *Cisco Research;*
Fan Lai, *University of Illinois Urbana-Champaign*

https://www.usenix.org/conference/nsdi26/presentation/zhang-wei

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

JITServe: SLO-aware LLM Serving with Imprecise Request Information

Wei Zhang[1] *, Zhiyu Wu[1]*, Yi Mu[1], Rui Ning[2], Banruo Liu[1], Nikhil Sarda[3], Myungjin Lee[4], Fan Lai[1]
[1]*University of Illinois Urbana-Champaign* [2]*Unaffiliated* [3]*Google* [4]*Cisco Research*

Abstract

The integration of Large Language Models (LLMs) into applications ranging from interactive chatbots to multi-agent systems has introduced a wide spectrum of service-level objectives (SLOs) for responsiveness. These include latency-sensitive requests emphasizing per-token latency in streaming chat, deadline-sensitive requests requiring rapid full responses to trigger external tools, and compound requests with evolving dependencies across multiple LLM calls. Despite—or perhaps, because of—this workload diversity and unpredictable request information (e.g., response lengths and dependencies), existing request schedulers have focused on aggregate performance, unable to ensure application-level SLO needs.

This paper presents JITServe, the first SLO-aware LLM serving system designed to maximize *service goodput* (e.g., the number of tokens meeting request SLOs) *across diverse workloads*. JITServe novelly schedules requests using imprecise request information and gradually relaxes this conservatism by refining request information estimates as generation progresses. It applies a *grouped margin goodput maximization* algorithm to allocate just enough serving bandwidth to satisfy each request's SLO *just-in-time* (JIT), maximizing residual capacity for others, while deciding the composition of requests in a batch to maximize efficiency and goodput with provable guarantees. Our evaluation across diverse realistic workloads, including chat, deep research, and agentic pipelines, shows that JITServe improves service goodput by 1.4×-6.3×, alternatively achieving 28.5%–83.2% resource savings, compared to state-of-the-art designs.

1 Introduction

As large language models (LLMs) enable language-driven interaction between humans and intelligent agents, modern applications increasingly go beyond conventional chatbot scenarios like ChatGPT. They often integrate LLMs with external tools (e.g., AI-assisted coding platforms [13] and autonomous web agents [89]), making it crucial to ensure LLM responsiveness (e.g., avoiding external system timeouts [86] or degraded user experience [79]). Increasingly, applications issue dependent LLM requests to enhance problem-solving, such as response aggregation in test-time scaling [50] or coordination in multi-agent systems (MAS) [47], introducing larger generation token volumes[1] and evolving request dependencies.

The ever-expanding landscape of LLM applications and user bases has introduced increasingly diverse service-level objectives (SLOs) for request responsiveness. Our analysis of millions of LLM requests across real-world applications—corroborated by user studies with hundreds of LLM users and extensive discussions with service providers—reveals that requests fall into three dominant patterns (§2.1): (i) *Latency-sensitive requests*: prioritize per-token latency metrics such as time-to-first-token (TTFT) and time-between-tokens (TBT), as in interactive chatbots where incremental streaming directly impacts user experience [42,79]; (ii) *Deadline-sensitive requests*: require low end-to-end latency (i.e., E2EL like time-to-last-token) to return a complete response quickly, as seen in cloud AIOps [67], batch-processing APIs [26], or agent interactions that trigger external tools [88]; (iii) *Compound requests*: consist of multiple dependent LLM calls to complete a task, for example in MAS [45], hierarchical reasoning [85], or planning workflows like deep research. The latter two categories often interact with external tools or agents, making it critical that their E2EL meets application-level deadlines to avoid downstream timeouts and ensure system reliability [78].

As SLOs directly capture application-level performance needs, maximizing service goodput (i.e., effective service throughput like the total number of tokens that meet request SLOs) is essential for both users and service providers. However, deploying dedicated clusters for each SLO workload is impractical due to prohibitive costs and the wide variability of SLO requirements, even among requests of the same type. For example, latency-sensitive chat requests exhibit different TBT requirements depending on user reading speeds [79,83], while deadline-sensitive and compound requests in cloud AIOps workflows impose heterogeneous E2EL constraints depending on task urgency and downstream operators (e.g., triggering remediation versus updating monitoring dashboards) [78,88].

In the face of growing diversity of workloads and request unpredictability (e.g., response lengths and runtime dependencies), existing LLM serving systems remain misaligned with service goodput. Instead, they typically optimize for aggregate serving throughput [10], average request completion time [45], or latency-sensitive requests only [83], which we prove can yield arbitrarily poor service goodput (Appendix E.1). For example, producing a response in 0.5 seconds

*indicates equal contribution.
Project code: https://github.com/UIUC-MLSys/JITServe.git

[1]A token is a basic unit of text processed or generated by an LLM.

rather than the expected 5 seconds provides little application benefit if a downstream tool requires 1 minute to execute, yet the excess "bandwidth" consumed could have been allocated to requests with more stringent SLOs.

This paper introduces JITServe, an SLO-aware serving system designed to maximize service goodput across diverse LLM workloads. At its core, JITServe employs a Just-in-Time (JIT) scheduling principle: it leverages imprecise request information (e.g., response length and dependency) to make initial scheduling decisions, and progressively refines these estimates as generation unfolds. JITServe then allocates just enough serving bandwidth (i.e., the number of tokens required to generate within a time frame) to satisfy each request's SLO, maximizing the residual capacity for others. JITServe enables aligning application-level SLO needs with the underlying LLM execution (e.g., vLLM [8]) with only a few lines of code modification to existing serving stacks (§3).

JITServe addresses two key request uncertainties to guide scheduling. First, while predicting exact response lengths is notoriously error-prone (§2.2), we find that estimating an upper bound is both more reliable and more efficient. This upper bound naturally translates into a conservative estimate of the maximum serving bandwidth required. To this end, JITServe leverages a lightweight quantile regression forest (QRF) method [39] to estimate this bound, and incrementally refines predictions as more response tokens are generated. Second, JITServe captures evolving execution dependencies in compound requests using *pattern graphs*, where nodes and edges represent LLM/tool inputs, outputs, and their dependencies. By performing incremental graph matching, it identifies prior requests with similar execution patterns, enabling informed group scheduling that mitigates per-stage stragglers (§4.1).

Beyond uncertainties, request scheduling in LLM serving introduces a unique two-dimensional scheduling challenge. First, *single-request scheduling*: even with complete future information (e.g., request length, dependencies, and arrival times), maximizing goodput by scheduling individual requests is already NP-hard. Second, *batch composition*: batching requests of heterogeneous lengths slows down per-layer execution due to uneven sequence loads, a bottleneck that persists even under state-of-the-art kernels such as Flash Decoding [3, 19]. To tackle both dimensions, JITServe proposes a novel *Grouped Margin Goodput Maximization (GMAX)* algorithm. The key idea is to ensure that each request receives its minimum serving bandwidth within every time frame by prioritizing those with high margin goodput, allowing surplus bandwidth to be reclaimed in later frames. At the same time, *GMAX* schedules requests with similar input lengths into batches. This joint strategy maximizes goodput and provides provable scaling and performance guarantees (§4.2–§4.3).

We implement JITServe atop vLLM [8], preserving API compatibility to support a wide range of applications (§5). We evaluate JITServe on diverse LLMs and real-world workloads, including chatbots [55], deep research [20], and agentic workflows [82]. Our results show that JITServe improves service goodput by 1.4×–6.3×, achieving 28.5%–83.2% resource savings for equivalent goodput, while achieving near-oracle performance (§6).

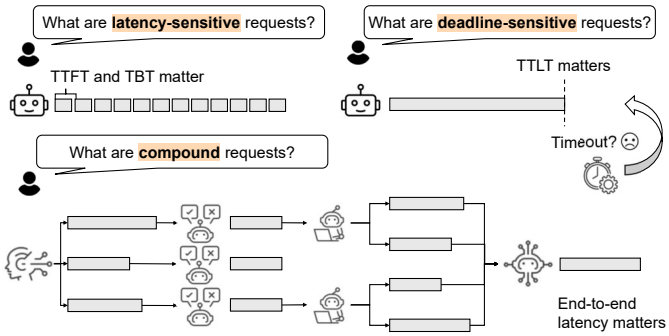


Figure 1: ***Illustration of three request patterns.***

In summary, our contributions are:

- We perform real-world studies of LLM services, including user surveys and discussions with providers, introducing a new characterization of request patterns (§2);
- We design a novel scheduler and the *GMAX* algorithm that estimate, refine, and exploit imprecise request information to maximize goodput with provable performance (§3-§4);
- We demonstrate JITServe's significant improvements in application-level performance across real workloads (§6).

2 Motivation

We begin with our real-world studies of LLM requests (§2.1), which reveal new challenges that motivate our work (§2.2).

2.1 Characterizing LLM Serving Requests

As LLMs enable interactive collaboration between human users and AI agents, and proliferate across diverse applications and user bases, optimizing *interaction latency* has become fundamental. To better understand realistic LLM request patterns, we conducted an extensive workload analysis covering millions of requests from diverse applications, including chatbots (LMSys Chatbot Arena [91], WildChat [90]), agentic AI systems (MetaGPT [21], GAIA [47]), and reasoning tasks (deep research [4, 20], math reasoning [85]). To validate and enrich these findings, we engaged in in-depth discussions with two major LLM service providers handling millions of daily requests, and conducted anonymized surveys with over 550 LLM users and developers across six academic and industry organizations. Together, our studies reveal that as LLM applications evolve, they introduce new request characteristics and diverse SLOs (Table 1). Detailed methodology and analysis are provided in Appendix A.

Latency-sensitive vs. Deadline-sensitive vs. Compound Requests. Our studies show that LLM requests can be increasingly categorized into three key patterns (Figure 1):

- *Type 1: Latency-sensitive Requests.* They generate responses consumed in a streaming fashion. Representa-

LLM Applications	Real-Time	Direct Use	Content-Based
Code generation	**38.1%**	30.5%	31.4%
Report generation	**39.1%**	36.2%	24.7%
Deep research	38.6%	**47.1%**	14.3%
Real-time translation	36.2%	**39.9%**	23.9%
Batch data processing	15.6%	**49.6%**	34.8%
Reasoning task	28.9%	**47.4%**	23.7%

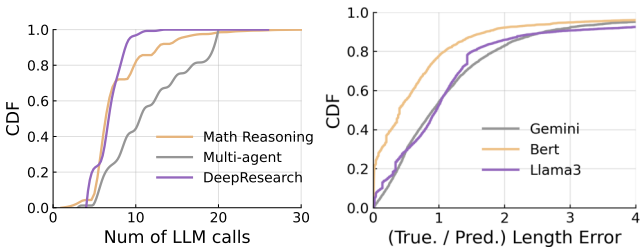
Table 1: ***Our real-world user study reports that users exhibit diverse SLO needs both across and within applications. "Real-Time" denotes users who prefer low per-token latency; "Direct Use" refers to those demanding for fast full responses; "Content-Based" reflects users whose needs vary depending on the specific context.***

tive applications include ChatGPT-style web services, AI-powered customer support [11], and real-time speech-to-text services [6]. For such requests, it is critical to maintain a content delivery rate (i.e., TTFT and TBT) that matches or exceeds the user's consumption pace (e.g., reading speed) to ensure a smooth interactive experience [79, 83].

- *Type 2: Deadline-sensitive Requests.* They require the full response (i.e., E2EL) to be generated within a specified deadline (e.g., to prevent downstream tool timeouts). This pattern arises in applications such as agent interactions that invoke external tools (e.g., for cloud AIOps [67, 88]), data cleaning [14], large codebase generation [33], and batch processing APIs at OpenAI and Gemini [26].
- *Type 3: Compound Requests.* These involve multiple dependent LLM calls, often forming graph-structured execution dependencies [43, 45, 70], with the requirement that the entire end-to-end generation (i.e., E2EL of finishing all requests) completes within the deadline. Examples include reasoning tasks using test-time scaling [85], multi-agent workflows [45], hierarchical planning scenarios such as deep research [2, 77], and reinforcement learning from human feedback pipelines where multiple responses must be generated per prompt [58].

Need for Accommodating Diverse SLO Requirements. Practical LLM workloads often involve mixed request types, and even a single request may transition across types during execution. For example, our user studies find that in multi-step reasoning tasks, the initial "thinking" phase is often deadline-sensitive—users expect internal reasoning to complete within seconds to avoid perceived stalls. Once the generation transitions to producing the final response, the workload becomes latency-sensitive, where smooth TBT is critical for interactive reading. This observation has been corroborated by other user-experience studies [76, 83] and our discussions with service providers. Further, certain requests may not impose explicit SLOs (e.g., synthetic data generation [41]), yet do not want to suffer starvation.

Beyond workload and application heterogeneity, SLO requirements also vary across users. As shown in Table 1, 30.5% of users prefer minimizing E2EL in code generation for rapid testing and direct execution, whereas 38.1% favor streaming code delivery to facilitate interactive reading and comprehension. Even within streaming use cases, users exhibit different reading speeds, translating into heterogeneous TBT requirements [44, 79, 83]. Batch APIs provide differentiated response guarantees across pricing tiers, leading to distinct deadline constraints [9, 56]. Agentic workflows also diverge in their downstream integrations (e.g., triggering automated remediation or updating monitoring dashboards), resulting in varied E2EL requirements [60].



(a) Varying number of LLM calls. (b) Large prediction deviations.

Figure 2: ***(a) LLM workloads increasingly involve varying numbers of subrequests (LLM calls) in a request. (b) Predicting response length remains highly inaccurate.***

A naive approach to request diversity is to dedicate clusters to each workload type. However, this is both cost-prohibitive and insufficient, as SLO needs vary even among requests of the same type. Worse, request types may evolve during execution (e.g., from "reasoning" to "streaming" phases), making migration (e.g., KV cache) across clusters costly.

2.2 Challenges and Limitations of Existing Solutions

As widely recognized in application-aware networking [15, 17], computing [64], and storage [31, 32], SLOs directly capture application-level performance needs. Completing requests much faster than their specified demands yields little additional *service goodput*. An effective scheduler should therefore allocate just enough serving "bandwidth" to satisfy each request's SLO, meeting application requirements while maximizing residual capacity for other requests or enabling substantial resource savings. Unfortunately, modern LLM serving systems face unique challenges and fundamental inefficiencies under this growing workload diversity.

Pervasive Request Uncertainties. LLM request scheduling must contend with multiple sources of uncertainty. First, request arrival patterns in online serving can fluctuate sharply, with load variations of up to 5× within minutes [75]. Second, modern LLM workloads often exhibit complex and unpredictable execution dependencies. As shown in Figure 2(a), applications such as multi-agent workflows [82], test-time scaling for reasoning [85], and deep research tasks introduce highly variable numbers of LLM invocations, often unknown a priori due to reflective reasoning or adaptive exploration (e.g., until reaching sufficient confidence [5]).

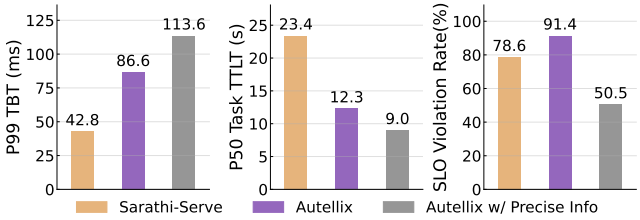


Figure 3: ***Existing advances face significant performance drops due to growing LLM request diversity.***

Finally, even when the dependency structure is known, predicting response lengths—especially for downstream requests whose inputs are not yet available but are required to ensure E2EL—is highly impractical, further challenged by probabilistic token sampling and self-reflection dynamics in response generation [46]. As shown in Figure 2(b), even with the full prompt input provided, both self-prediction (e.g., Gemini estimating its own output length) and fine-tuned predictors (e.g., BERT- or Llama3-based models [61]) exhibit substantial length prediction errors.

Misaligned Service Goodput and Inefficiency in Existing Solutions. State-of-the-art schedulers fail to generalize under increasing workload and SLO diversity. First, they primarily optimize aggregate metrics, such as minimizing mean E2EL via Shortest-Job-First (SJF) variants with predicted length ranking [93] or Least-Attained Service (LAS) First as in Autellix [45]. However, we theoretically prove that they can lead to arbitrarily poor goodput (Appendix §E.1): even if the mean improves, many requests still miss their SLOs, leading to service unpredictability, cascading violations across dependent tasks, and overclaiming resources to sustain service. As concrete evidence, Figure 3 shows that while Autellix improves average E2EL compared to Sarathi-Serve [10], it suffers from higher and over 90% SLO violation rates.

Second, perhaps due to the challenges of request uncertainty, existing serving optimizations that consider user experience are restricted to latency-sensitive workloads (e.g., Sarathi-Serve), since TTFT and TBT primarily depend on known input lengths. Yet, as shown in Figure 3, Sarathi-Serve achieves low TBTs but performs poorly for deadline-sensitive requests (e.g., large TTLTs), resulting in high SLO violations. Finally, even within their intended design regimes, these schedulers fall substantially short of the oracle baseline with perfect knowledge of request lengths and dependencies: Figure 3 demonstrates that Autellix achieves 41% higher SLO attainment rates when provided with precise information.

Addressing these limitations is critical for both LLM service users and providers, calling for a new scheduler that explicitly aligns LLM execution with application-level needs and satisfies three essential properties:

- *Generalizability*: Support diverse request types (e.g., latency-, deadline-, and compound requests) and SLO requirements, ensuring predictable SLO satisfaction without collecting intrusive request and application information.
- *Goodput Efficiency*: Maximize service goodput and resource utilization for providers, while remaining robust to runtime dynamics with provable performance guarantees.
- *Deployability and Scalability*: Integrate seamlessly with existing serving stacks and provide strong scaling capabilities (e.g., extending to multiple concurrent models).

3 JITServe Overview

This paper introduces JITServe, an SLO-aware LLM request scheduler that generalizes across diverse workloads and SLOs, maximizing service goodput (e.g., the number of useful tokens delivered) and resource utilization for both LLM users and providers under imprecise request information. JITServe provides provable guarantees and achieves empirically near-oracle performance, complementing existing serving infrastructure with only a few lines of code change in APIs.

Design Space. JITServe targets practical serving deployments where requests arrive online and completed ones exit, covering diverse LLM workloads and SLOs. We adopt the widely used notion of goodput [1]:

- *Latency-sensitive requests*: measured as the number of tokens delivered within the expected *timeline* [79, 83]. Specifically, token i counts toward goodput if it finishes by $TTFT_{SLO} + i \times TBT_{SLO}$.
- *Deadline-sensitive requests*: measured as the total number of tokens, including input and output, if the request completes before its deadline; zero otherwise [79].
- *Compound requests*: measured as the total number of tokens across all subrequests if the final generation completes by the E2EL deadline; zero otherwise [45].

JITServe is agnostic to the specific definition of goodput and operates directly over the metric provided by the service provider. For example, the provider can define the final subrequest in compound requests as latency-sensitive requests in the goodput objective function. It also accommodates non-SLO requests (e.g., best-effort requests) by assigning a default completion deadline to avoid starvation (§4.2). We further validate that JITServe consistently improves various goodput objectives (§6.2), such as request-level goodput like maximizing the number of requests that meet their SLOs [94].

At its core, JITServe employs a Just-in-Time (JIT) scheduling strategy to be conservative yet adaptive under runtime uncertainties. Instead of assuming precise knowledge or ignoring uncertain information, JITServe initially estimates quantile-based upper bounds for response lengths and request dependency graphs, allocating conservative bandwidth to prevent SLO violations. As response generation progresses, JITServe refines these estimates, gradually relaxing bandwidth allocations to maximize residual capacity for other requests.

System Workflow. As illustrated in Figure 4, JITServe operates as a middleware layer that aligns application-level performance needs with underlying execution backends (e.g.,

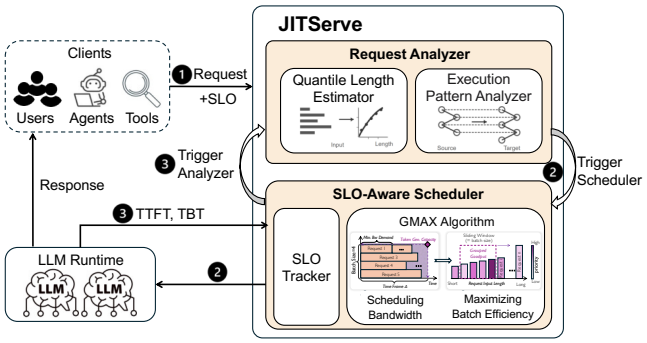
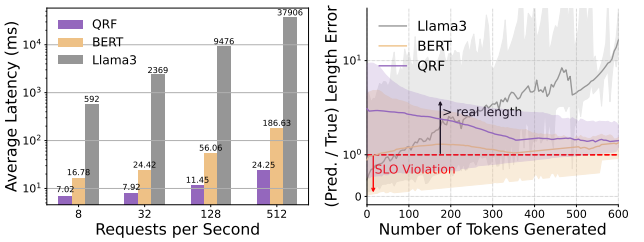


Figure 4: ***JITServe system overview and workflow.***



(a) Estimation overhead. (b) Estimation accuracy.

Figure 5: ***(a) Average prediction latency. (b) QRF achieves better upper-bound prediction than tuned BERT and LLama3-based predictor over time. The shaded area shows P5-P95 distribution, and the red line represents the ground truth (ratio = 1).***

vLLM [8]). ❶ Upon request arrival, along with its SLO (specified by users [44] or developers [42]), the Request Analyzer estimates key request information, including an upper bound on output length and execution dependencies (e.g., via graph matching to historical request patterns). ❷ The SLO-Aware Scheduler determines the minimum number of tokens each request should generate within a time frame, prioritizing those with higher margin goodput while grouping requests with similar input lengths into batches to maximize per-iteration batch execution speed. ❸ Leveraging the SLO Tracker, which monitors actual generation speeds and continuously updates estimates from the Request Analyzer, the scheduler efficiently makes admission and preemption decisions.

4 JITServe Design

SLO-aware LLM serving must address three fundamental challenges: (1) *Generalizability*: proactively estimating and refining request information during execution to meet diverse SLOs (§4.1), which informs (2) *Goodput Efficiency*: scheduling individual requests and their batch compositions to maximize service goodput (§4.2), and (3) *Deployability*: adapting to diverse deployment considerations, such as fairness and strong scaling capabilities (§4.3). We next describe how JITServe addresses them in real time.

4.1 Request Analyzer: Refining Imprecise Information

Predicting request information is notoriously difficult, especially for downstream requests without knowing their prompt inputs, yet we prove that scheduling without it—e.g., using traditional Least-Attained Service [45] or Earliest-Deadline-First policies—can lead to arbitrarily poor service goodput due to the misalignment between optimizing for aggregate performance and meeting individual SLOs (Appendix E.1). Our key insight is that LLM responses are generated autoregressively over hundreds of decoding iterations, enabling a middle-ground approach between assuming full information and assuming none. Specifically, we leverage imprecise but actionable, continuously refinable estimates to inform scheduling. To this end, JITServe employs two key techniques using the information already available in serving: (1) *quantile-based upper-bound prediction of response length*, and (2) *dynamic pattern-graph matching* based on history.

Estimating Response Length Upper-bound. Determining just enough serving bandwidth to meet each request's SLO requirements requires knowing its response length. Underestimation risks SLO violations as deferring long-response requests by mistake will miss deadlines, while overestimation wastes capacity due to overclaiming bandwidth.

To balance these risks, JITServe conservatively overestimates response length yet progressively refines the estimate over generations. It uses a Quantile Regression Forest (QRF) model [52] for response length prediction. Compared to range classification approaches [61] that require predetermined buckets, QRF customizes quantile intervals in predictions for each specific request, making it more adaptable across diverse prompts and models. We leverage QRF to output a high-quantile bound on response length given a request's prompt, which corresponds to the maximum bandwidth requirement. The QRF prediction is lightweight: as shown in Figure 5(a), each prediction takes only 7 ms, 7× faster than a fine-tuned BERT predictor [61].

Our lightweight design makes online refinement practical: the Request Analyzer augments the prompt with newly generated tokens and periodically re-invokes QRF (e.g., every 50 tokens) to refine the upper bound, thereby progressively reducing conservatism. This provides two additional benefits: (1) it adapts to generation-time variability, where even identical prompts may yield different outputs from the same model (e.g., due to probabilistic token sampling); and (2) it generalizes across models by incrementally incorporating model-specific responses into predictions, thereby improving accuracy. Even disregarding BERT's substantial overhead, Figure 5(b) shows that both fine-tuned BERT and Llama3 prediction models often underestimate response lengths, risking frequent SLO violations. In contrast, QRF produces reliable upper-bound estimates that relax as generation progresses.

Estimating Dependency with Pattern-Graph Matching. Beyond uncertainty at the individual request level, many workloads exhibit graph-structured execution (e.g., deep research [2]), where nodes represent LLM or tool invocations and edges encode their dependencies. Moreover, these dependency structures may evolve dynamically (e.g., to achieve suf-

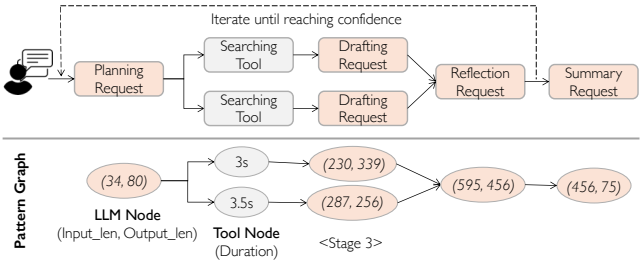
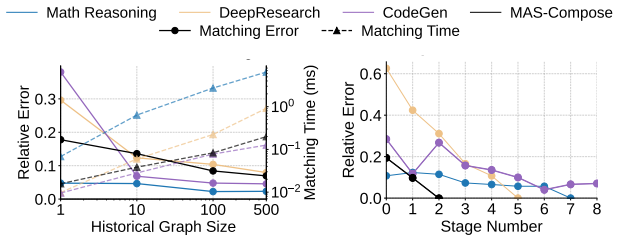


Figure 6: ***Example of Pattern Graph consisting of five stages. LLM node weighted by (input_len, output_len), tool node weighted by execution time.***



(a) Impact of historical repo size. (b) Impact of online matching.

Figure 7: ***(a) larger historical graph sets reduce matching error while exhibiting sublinear time growth. (b) next-stage estimation error decreases as more stage information becomes available. Note that the next-stage estimation error becomes zero when the maximum number of stages is already reached (i.e., $t_s = 0$).***

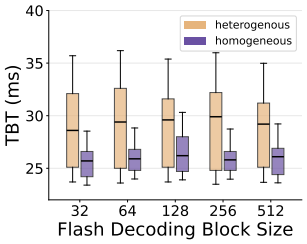


Figure 8: ***Batching requests with heterogeneous lengths slows down execution.***

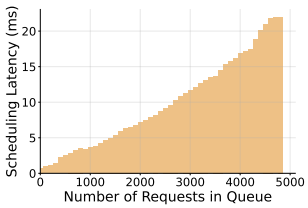


Figure 9: ***GMAX scales efficiently to schedule thousands of concurrent requests.***

ficient response confidence [2]), making it difficult to satisfy SLOs (e.g., E2EL), since the response generation of deeper-stage requests depends on unknown parents' outputs (§2.2).

Our key insight is twofold: (1) exploit historical requests with structurally similar execution graphs to infer likely dependency patterns, and (2) amortize SLO requirements (e.g., deadlines) across intermediate stages to ensure steady progress, thereby avoiding overly deep planning that introduces significant noise. We represent each served request as a primitive *pattern graph*, without needing raw input/output plaintext. As illustrated in Figure 6, each node correspond to one LLM or tool invocation, annotated with input/output length (for LLMs), execution time (for tools), and the model/tool identity, while edges capture node dependencies.

As a new request unfolds in response lengths and new invocations, the Request Analyzer incrementally extends its partial graph with newly revealed dependencies, prunes past patterns (graphs) whose *prefix structures* diverge (e.g., invoking a different model/tool at the current stage), and performs similarity matching against the remaining candidates. Node and edge similarities are computed using Gaussian-kernel functions [36] over their attributes (output lengths for nodes, input lengths for edges), enabling progressively refined pattern matching as more information becomes available.

Once the most similar historical pattern graph is identified, we use it to estimate the cumulative contribution of prior stages relative to overall execution. This enables proportional sub-deadline allocation across stages rather than treating them uniformly. Specifically, we compute the accumulated share as $\phi(s) = \frac{t_{\leq s}}{t_{\text{total}}}$, where $t_{\leq s}$ is the accumulated execution time up to stage s, and t_{total} is the total execution time *in the pattern graph*. Intuitively, $\phi(s)$ captures the historical progress made up to stage s as a fraction of the full execution timeline. The amortized deadline for stage s in a new request with total deadline D is then set as $D_s = \phi(s) \cdot D$, ensuring that each stage receives a sub-deadline proportional to its cumulative contribution. We also evaluated alternative formulations (e.g., t_s/t_{total}) and found that our accumulated-share design consistently outperforms them in both analytical modeling and empirical evaluation (Appendix B), offering greater robustness and accuracy by grouping previous stages' information.

To ensure efficiency, we cluster historical pattern graphs offline using a K-medoids mechanism [51], and evict patterns with low reuse frequency (decayed by 0.9 every hour). Each stored pattern graph is compact, typically under 0.2KB. As shown in Figure 7(a), our matching procedure achieves both high efficiency and accuracy: the matching latency remains below 5 ms even with 500 historical graphs, while accuracy already saturates with such a modest history size. This lightweight design enables online matching in real-time serving. As shown in Figure 7(b), the relative error in next-stage ratio estimation decreases as additional stage information becomes available, demonstrating progressive refinement.

4.2 SLO-aware Scheduler with *GMAX* Algorithm

With imprecise yet continuously refined request estimates, the SLO-aware scheduler aims to maximize service goodput by allocating just enough serving bandwidth (e.g., the minimum generation tokens required within a time frame) to satisfy each request's SLO, thereby preserving residual capacity for others. This creates a unique two-dimensional scheduling challenge that extends beyond traditional scheduling problems. First, *single-request scheduling*: even under complete future information (e.g., exact response length, dependency, and arrival time), we prove that maximizing goodput by scheduling individual requests is already NP-hard (Appendix D.1). Second, *batch composition*: LLMs execute requests in batches, but batching requests with heterogeneous input lengths reduces per-token generation speed due to uneven input loads across samples in each model layer's batch execution, distinct from prior continuous batching problems [87]. As shown in Fig-

Algorithm 1: *GMAX* Algorithm

```
1  Class RequestAnalyzer:
2      Function AnalyzeRequest(req):
           // Capture minimum serving bandwidth each request
              needs to meet SLOs
3          req.len_rem ← PredictLength(req)
4          req.bw ← req.len_rem / EstimateRemainingTime(req)
5          req.goodput ← EstimateGoodput(req)
6          req.priority ← req.goodput / req.bw

7  Class Scheduler:
8      Function Schedule(batch_size, cutoff):
9          Q ← GetRequestQueue()
10         foreach req in Q do
11             AnalyzeRequest(req)
12         bp ← BatchPriority(Q, batch_size)
           // Step 1: candidate filtering by priority cutoff p
13         Candidates← Q.Filter(req : req.priority ≥
             bp × cutoff)
           // Step 2: group by input length (sliding window)
14         Candidates.sort(req : req.input_length)
15         BestGroup ← ∅, max_score ←-∞
16         foreach window G of size B in Candidates do
17             score ← Σ_{r∈G} r.priority
18             if score > max_score then
19                 BestGroup ← G, max_score ← score
20         return BestGroup
```

ure 8, this inefficiency persists even with advanced kernels such as Flash Decoding [3].

Algorithm 1 summarizes how JITServe addresses the online two-dimensional scheduling challenge with the *Grouped Margin Goodput Maximization* (*GMAX*) algorithm. The scheduler first queries the Request Analyzer to determine each request's *minimum serving bandwidth* requirement (Lines 2-6). It then prioritizes requests with the highest goodput payoff relative to their bandwidth consumption, while grouping (scheduling) those with similar input lengths into a batch to maximize grouped goodput and batching efficiency (Lines 13-19). As generation progresses, both the Request Analyzer and the scheduler continuously refine request estimates and update scheduling decisions accordingly (Lines 10-11).

Capturing Minimum Serving Bandwidth per Request. For each request r, the *minimum serving bandwidth* depends on its remaining work (i.e., the remaining response length to generate) and the remaining time budget. Formally, we define: $bw(r) = \frac{t_{\text{gen}}(r)}{t_{\text{rem}}(r)}$, where $t_{\text{gen}}(r) = len_{\text{rem}}(r) \cdot v_{\text{token}}(r)$ is the upper-bound estimate of the remaining generation time, computed as the remaining response length $len_{\text{rem}}(r)$ upper

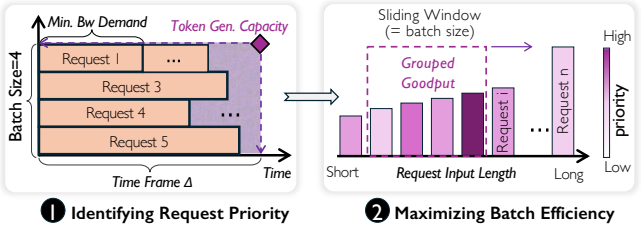


Figure 10: ***GMAX* (1) identifies the scheduling priority of each request from its bandwidth demand, and then (2) selects requests to maximize the grouped margin goodput and batch efficiency.**

bound times the average per-token generation speed $v_{\text{token}}(r)$. This estimate is conservatively initialized and incrementally refined during generation (§4.1). $t_{\text{rem}}(r)$ is the remaining time to the request deadline for deadline-sensitive or compound requests. For latency-sensitive requests, explicit SLOs (e.g., TBT) already define the per-token service bandwidth.

Because maintaining a fixed bandwidth throughout a request's lifetime is impractical due to runtime dynamics (e.g., new request arrivals or early completions), *GMAX* amortizes bandwidth allocation over discrete scheduling frames of length Δ. As illustrated in Figure 10, execution is decomposed into consecutive frames. Each frame provides a token-generation capacity (the purple shaded area) across the time (Δ) and batch-size dimensions. A request r is represented as a rectangle occupying one batch slot with a frame-level bandwidth of $bw_\Delta(r) = \frac{t_{\text{gen}}(r)}{t_{\text{rem}}(r)} \cdot \Delta$. For compound requests, both $len_{\text{rem}(r)}$ and $bw_\Delta(r)$ are aggregated across all subrequests within the current stage, since completing a single subrequest does not advance the stage.

Analogously, we amortize each request's potential goodput: $goodput_\Delta(r) = \frac{goodput(r)}{t_{\text{rem}}(r)} \cdot \Delta$, where $goodput(r)$ denotes the achievable goodput contribution of completing r, depending on the developer's SLO specification (§3). Scheduling thus reduces to efficiently placing request rectangles into the per-frame capacity while maximizing aggregate $goodput_\Delta$. A natural solution is dynamic programming (e.g., $DP(\mathcal{R}, t, B)$ for request set $\mathcal{R}$, time t, and batch size B), but such methods scale poorly with large request sets and cannot flexibly incorporate practical factors like preemptions.

To address this, *GMAX* uses a lightweight design that prioritizes requests by their *margin goodput per unit bandwidth*:

$$Priority(r) = \frac{goodput_\Delta(r)}{bw_\Delta(r)} = \frac{goodput(r)}{t_{\text{gen}}(r)}.$$

This formulation naturally prefers requests with high payoff relative to their bandwidth demand, while eliminating sensitivity to Δ. To avoid starvation, including for best-effort requests without explicit SLOs, *GMAX* inflates each deemed $goodput(r)$ by a small additive constant δ per frame, ensuring long-waiting requests eventually rise in priority. We later show that this heuristic achieves competitive performance guarantees in theory and near-optimal goodput empirically.

Grouped Margin Goodput Maximization for Batch Scheduling. Simply prioritizing requests by margin goodput can produce batches with highly heterogeneous input lengths, which degrades batching efficiency and ultimately reduces service goodput (Figure 8). Next, *GMAX* extends individual prioritization into a *grouped scheduling strategy* that jointly balances goodput payoff and length homogeneity.

As illustrated in Figure 10, let B denote the batch size. *GMAX* first filters requests by retaining only those whose priority is at least $p \cdot Priority(r_{(B)})$, where $Priority(r_{(B)})$ is the B-th highest priority and $0 < p \leq 1$ is a tunable cutoff (e.g., 0.95). This ensures that subsequent group scheduling focuses on a promising candidate pool. The retained requests are then sorted by input length, and a sliding window of size B traverses the list. For each candidate group $\mathcal{G} \subseteq \mathcal{R}$ of size B, *GMAX* computes the aggregate priority: $Priority(\mathcal{G}) = \sum_{r \in \mathcal{G}} Priority(r)$. The group with the maximum $Priority(\mathcal{G})$ is selected as the execution batch, ensuring both high expected goodput and input-length alignment.

The cutoff parameter p controls the tradeoff: smaller p admits more candidates, improving homogeneity at the expense of diluting group-level goodput with low-priority requests, while larger p enforces stronger goodput guarantees but increases heterogeneity due to the long-tailed input length distribution (Figure 8). Fortunately, because LLM serving is long-running, *GMAX* automates and continuously adapts p online by exploring different thresholds and converging to those that maximize end-to-end goodput.

Preemption to Correct Scheduling Errors. Optimal scheduling decisions rely on both future arrivals and progressively refined request information, introducing uncertainty that may lead to suboptimal online decisions. Preempting running requests can correct these errors but incurs overheads (e.g., batch stalls and KV cache eviction/reload).

JITServe explicitly models these costs to ensure generalizability across hardware settings. KV cache reload latency is primarily bounded by memory I/O bandwidth when restoring states from DRAM to on-device memory, while recomputation latency is instead dominated by GPU compute throughput (FLOPs), creating a hardware-dependent trade-off. For either strategy, JITServe estimates the resulting stall duration directly from the affected sequence length and the corresponding bandwidth or compute rate. We quantify the potential impact using $goodput\_loss = stall\_duration \times token\_generation\_speed$, and perform preemption only when the projected gain from admitting a higher-priority request exceeds this cost. To avoid excessive churn, scheduling updates are restricted to discrete time frames (e.g., $\Delta = 50$ decoding steps; about 300 ms). This time-slicing aligns with the frame-based scheduling formulation, smooths execution, and allows any surplus bandwidth to be reclaimed in subsequent frames. Together, these mechanisms enable JITServe to correct scheduling decisions with negligible overhead ($< 1\%$) in practice (§6.2).

Achievable Guarantees. We next analyze the scheduling efficiency and quality of *GMAX*: (1) *Scalability*: Given N requests, computing the minimum serving bandwidth for each request requires $O(N)$ time, ordering their priorities adds $O(N \log N)$, and composing batches through sliding-length grouping incurs another $O(N)$. Overall, the scheduling process is bounded by $O(N \log N)$ complexity. Figure 9 shows that *GMAX* scales efficiently, scheduling thousands of concurrent requests within 20 milliseconds, making it practical for online serving (§6.2); (2) *Quality*: Through the *amortized analysis* method [35], we prove that *GMAX* achieves a competitive performance guarantee relative to the even *offline* optimal scheduler with future request arrival information. A detailed proof is provided in Appendix E.2, specifically:

Theorem 4.1. *Let $G_{GMAX}(\mathcal{R})$ denote the goodput achieved by online GMAX on the request set $\mathcal{R}$, and $G^{\star}(\mathcal{R})$ corresponds to the goodput achieved by the optimal offline scheduler. Then we have a guarantee that $G_{GMAX}(\mathcal{R}) \geq \frac{1}{8.56} \cdot G^{\star}(\mathcal{R})$.*

4.3 JITServe across Design Space

An ideal scheduler must adapt to diverse LLM deployment scenarios without requiring reinvention, handling multiple models, ensuring fairness, and maintaining robustness under unfavorable SLO settings.

Supporting Multiple Models. Practical deployments often replicate models to scale throughput, with replicas potentially operating at different speeds (e.g., due to heterogeneous hardware or batch sizes). While JITServe scales efficiently—its request estimation and refinement can run in parallel across requests, and *GMAX* achieves low computational complexity ($O(N \log N)$)—supporting multiple models introduces a new challenge: a single request r may have different serving bandwidth requirements across model replicas due to varying generation speeds or data locality (e.g., KV cache).

To address this, we extend *GMAX* using a *power-of-K* approach. For each request r, we create K dummy copies $[r_1, \ldots, r_K]$, by randomly sampling K models from the M available models. Each dummy r carries a replica-specific priority $priority(r)$, and scheduling proceeds as usual over the enlarged set. Once a request is assigned to a replica, its other dummies are removed from the queue, incurring negligible overhead since no real LLM execution is involved. Thanks to *GMAX*'s strong scaling capability, K can be set equal to M, ensuring full replica coverage. This multi-model extension increases scheduling complexity at most by $O(K)$ while aligning requests with their most favorable replicas, preserving provable performance guarantees. Empirical results confirm that JITServe consistently achieves superior performance in multi-model deployments (§6.4).

Extending to Other Objectives. Prioritizing requests solely by goodput can lead to unfairness and be vulnerable to outliers in unfavorable settings. For example, corrupted users may continuously submit requests with extremely strict SLO

demands to monopolize serving bandwidth, which again boils down to ensuring fairness in the wild.

JITServe can seamlessly incorporate additional objectives, such as fairness, with minimal changes. Given a developer-specified fairness function $\text{Fair}(r)$, we redefine the request priority as $priority'(r) = (1-f) \cdot priority(r) + f \cdot \text{Fair}(r)$, where $priority(r)$ is the default goodput density (§4.2) and $f \in [0,1]$ balances efficiency and fairness. As $f \to 1$, requests with lower fairness attainment gain higher priority. This lightweight adaptation allows JITServe to enforce diverse fairness policies while offering flexible tradeoffs.

5 Implementation

We implemented JITServe atop vLLM [8] with about 2,800 lines of code, preserving its APIs for broad compatibility.

Execution Backend. JITServe augments the vLLM core engine with a policy module that generalizes the scheduler layer to support multiple scheduling policies, while preserving the efficiency guarantees of chunked-prefill execution. JITServe further inherits vLLM's prefix caching [92] and sharing mechanisms to maximize reuse across overlapping requests. The module maintains a compact priority cache to amortize priority computations, updating only upon request arrivals or preemption events to reduce redundant overhead.

Control Plane. The QRF-based length predictor and pattern graph matcher are offloaded to a separate asynchronous process through gRPC communication. The exchanged metadata is only a few bytes per event, making the communication overhead negligible. For robustness, it integrates a monitoring daemon that tracks component liveness and persists periodic metadata checkpoints, ensuring rapid state reconstruction and minimal recovery latency under failures. JITServe extends the OpenAI API [7] with SLO-aware parameters, specifically *client.responses.create(model, input, deadline=None, target_tbt=0.2, target_ttft=5, waiting_time=5)*. For admission control, JITServe enforces a maximum waiting_time (e.g., 5 seconds): requests unscheduled beyond it are dropped to prevent overload, ensuring predictable service behavior.

6 Evaluation

We evaluate JITServe with a variety of popular models and LLM applications. Our main observations include:

- JITServe improves service goodput by 1.4×-6.3× over existing advances, alternatively achieving 28.5%–83.2% resource savings to sustain the same goodput, while achieving near-oracle performance (§6.2);
- JITServe effectively balances performance across diverse request types, achieving strong P50 metrics and comparable P95 tail latency for all request patterns (§6.3);
- JITServe demonstrates robustness across different SLO requirements, workload compositions, and distributed settings, consistently outperforming baselines (§6.4).

Workload	Req Type	Metric	Mean	Std.	P50	P95
Chatbot	Single	Input	93	244	27	391
		Output	318	313	225	1024
	Compound	Input	1300	912	1097	2767
		Output	4458	1176	4417	6452
Deep Research	Single	Input	1911	2781	403	7573
		Output	534	644	410	1544
	Compound	Input	12223	8407	10807	29282
		Output	3541	2370	3148	7525

Table 2: ***Our evaluations include four popular applications: Chatbot, Deep Research, Agentic CodeGen, and Math Reasoning. This table shows example request length statistics for two of them.***

6.1 Experimental Setup

We evaluate JITServe on a set of widely used LLMs with diverse architectures, including Llama-3.1-8B [28], Qwen2.5-14B [84], Qwen3-30B-MoE-A3B [72], and Llama-3.1-70B [28]. These models span both dense and MoE designs as well as different parameter scales. Experiments are conducted on a cluster of 16 NVIDIA A100 GPUs.

Workloads. To construct the three request patterns, we use the Alpaca [71] and LMsys-chat [91] datasets to build the Chatbot application. We further incorporate a long-context math reasoning application [85], a deep research application based on the Search Arena benchmark [49], and an agentic code generation application [82]. From LMsys-chat usage analysis [91], we extract the distribution of real-world use cases. Requests are tagged according to statistics from our user study (Table 1). For example, 38.1% of code generation requests are classified as latency-sensitive, while the deep research application is modeled as compound requests. Table 2 reports the request characteristics for two of all four applications. Request arrivals follow Microsoft's real-world LLM serving trace, scaled to match our cluster resources. Following prior advances [37, 68], We also perform ablation studies with arrivals generated using a Poisson distribution. Each evaluation run involves more than 10K requests over an online deployment window of at least one hour.

We set request SLOs using the P95 latencies measured from 1K DeepSeek API calls. This results in latency-sensitive requests requiring ∼2s TTFT and ∼100ms TBT, while deadline-sensitive requests have an E2EL of 20s. For compound requests, the E2EL SLO is scaled with the number of stages, defined as 20× (number of stages) seconds. The QRF model is configured with 300 trees and a maximum depth of 150.

Unless otherwise noted, we adopt a 1:1:1 ratio across the three request patterns, which yields a workload mix dominated by latency-sensitive requests. We also show that JITServe achieves consistent improvement across different settings of SLO requirements and workload compositions (§6.4).

Baselines. We evaluate against four state-of-the-art designs:

- *Autellix* [45]: Uses Program-level Least Attained Service scheduling (PLAS) to imitate SJF, optimizing request

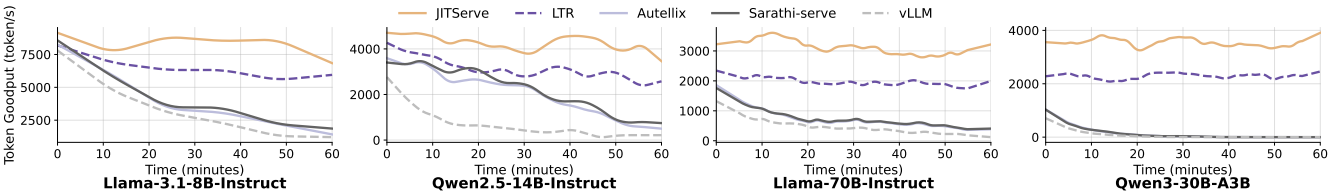


Figure 11: ***Service goodput over time in a one-hour online serving experiment. JITServe achieves consistently high service good while the baselines suffer cascading SLO violations and degraded service goodput over time.***

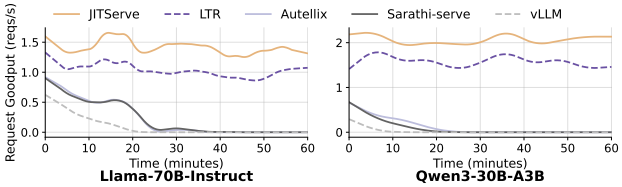


Figure 12: ***JITServe achieves consistently better request-level SLO service goodput in online deployments.***

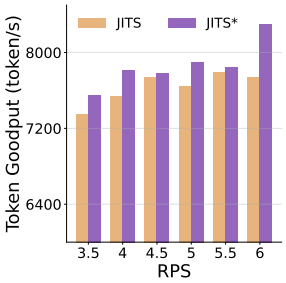


Figure 13: ***JITServe achieves close-to-oracle performance.***

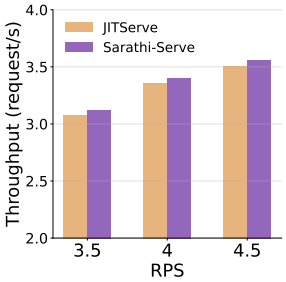


Figure 14: ***JITServe introduces little overhead in throughput.***

completion time (E2EL) for compound workloads.

- *Learn to Rank (LTR)* [24]: Leverages an LLM prediction model to predict the relative response length ranking of requests and prioritize the smallest one, imitating SJF.
- *vLLM* [37]: A recent advanced LLM serving backend with continuous batching [87] and PagedAttention [37]. Uses FCFS scheduling.
- *Sarathi-Serve* [10]: Extends vLLM with chunked prefills to optimize TTFTs and TBTs for requests.
- *SLOs-Serve [16]*: A multi-SLO targeting approach that employs dynamic programming scheduling framework to optimize resource allocation.

Metrics. We focus on higher *service goodput*. Since our design is agnostic to the definition of goodput (§3), we choose two popular goodput definitions focus on different levels: (1) *Token-level goodput*: the number of tokens meeting SLO requirements, following popular SLO preferences; and (2) *Request-level goodput*: the number of requests meeting the SLO requirements. We also report traditional metrics such as TTFT, TBT, and throughput for performance breakdown.

All results are averaged over five independent runs.

6.2 End-to-End Performance

We start with end-to-end evaluations in online deployments.

JITServe substantially improves service goodput. We first deploy JITServe in a one-hour online experiment to evaluate its long-term performance. As shown in Figure 11, JITServe consistently achieves high service goodput over time, outperforming LTR by 1.3×–1.7× and Autellix by 5.3×–6.1×. This improvement arises from JITServe's ability to dynamically prioritize requests based on per-request SLOs and their serving bandwidth requirements.

In contrast, existing systems such as Sarathi-Serve and vLLM suffer from increasing head-of-line (HOL) blocking, leading to cascading SLO violations and reduced service goodput over time. JITServe mitigates this degradation by leveraging conservative upper-bound length predictions and maximizing residual bandwidth for other requests, resulting in stable service goodput throughout the evaluation. Notably, JITServe maintains high service goodput consistently over the entire one-hour deployment.

In addition to token-level goodput improvement, we also measure request-level goodput. As shown in Figure 12, JITServe achieves 2.3×-4.5× higher goodput than LTR.

JITServe achieves near-oracle performance. We further benchmark JITServe against an oracle variant, JITServe *, which operates with perfect foresight of request information (i.e., response length and execution graph) across varying request-per-second (RPS) settings. As shown in Figure 13, JITServe achieves performance within 3–9% of the oracle despite relying on imperfect predictions. This robustness arises from two key factors: (1) the Request Analyzer generalizes effectively to unseen workloads and request mixes, and (2) the scheduling policy is designed to tolerate uncertainty, leveraging approximate information and progressively relaxing the conservatism to make near-optimal decisions. Together, these results demonstrate that JITServe approaches the best achievable performance without strong assumptions.

JITServe does not hurt system throughput. A common concern with sophisticated scheduling is the potential throughput loss. To evaluate this, we compare JITServe against Sarathi-Serve, which employs FIFO scheduling without preemption and thus represents a near upper-bound on serving throughput. As shown in Figure 14, JITServe achieves comparable throughput to Sarathi-Serve across different request-per-

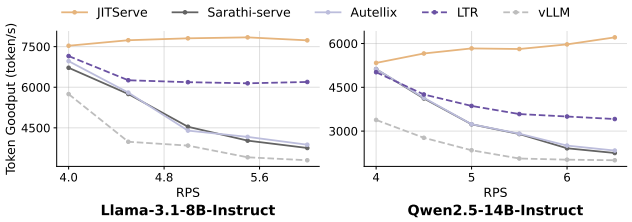
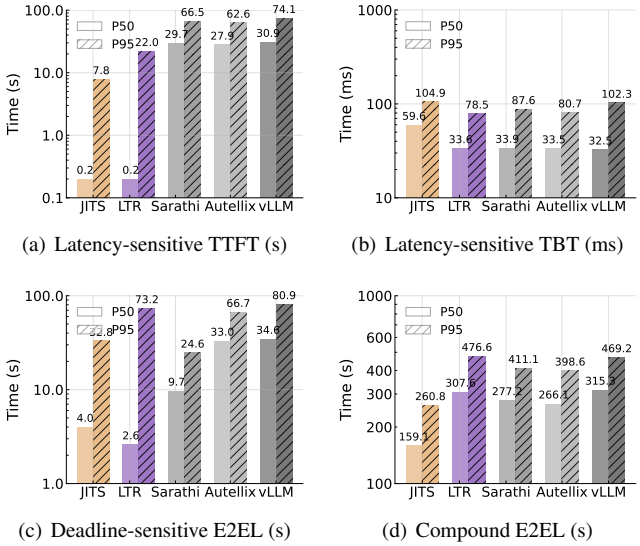


Figure 15: ***JITServe sustains high goodput across request loads.***



(a) Latency-sensitive TTFT (s) (b) Latency-sensitive TBT (ms)

(c) Deadline-sensitive E2EL (s) (d) Compound E2EL (s)

Figure 16: ***Performance metrics comparison across different baselines. (a) Time to First Token, (b) Time Between Tokens, (c) Deadline-Sensitive Request Latency, (d) Compound Request Latency. All metrics are shown in log scale with P50/P95 percentiles.***

second (RPS) settings, reaching 96%–98% of its performance. This demonstrates that JITServe's additional modules incur negligible overhead, aided by its cost-aware design that selectively corrects scheduling errors only when the potential goodput gains outweigh preemption costs (§4.2).

JITServe sustains high goodput under load surges. To evaluate scalability and robustness, we measure service goodput under varying request arrival rates. As shown in Figure 15, all baselines exhibit sharp performance drop as system load increases due to contention. In contrast, JITServe consistently achieves the highest goodput across all load levels by dynamically adjusting priorities and resource allocations.

As a result, JITServe significantly mitigates the impact of increasing request rates on goodput degradation, confirming its suitability under real-world load dynamics.

6.3 Performance Breakdown

We next analyze JITServe's performance by breaking it down into two key aspects: its ability to handle different request types, and its performance when key components are ablated. By understanding these aspects, we gain deeper insights into JITServe's effectiveness and how it performs in various scenarios.

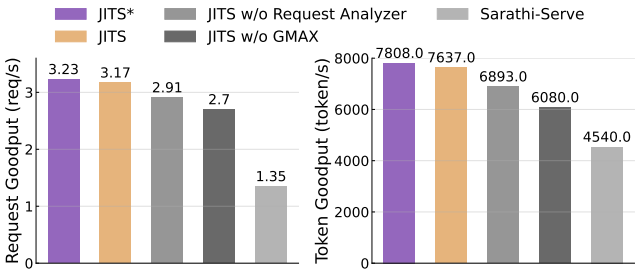


Figure 17: ***Request analyzer and GMAX algorithm jointly contribute to JITServe's high-quality, resilient serving.***

Breakdown by Request Types. While we have demonstrated JITServe's large goodput improvement, we next study how JITServe handles diverse SLOs across different request types, using conventional performance metrics such as TTFT, TBT, and E2EL. As shown in Figure 16(a), JITServe excels at minimizing TTFT for latency-sensitive requests, demonstrating its ability to deliver responsive service under strict SLOs. Notably, this is achieved without incurring excessive TBT (Figure 16(b)). For deadline-sensitive and compound requests, JITServe achieves favorable median E2EL (Figures 16(c) and 16(d)), ensuring that many requests meet their SLOs without oversubscribing bandwidth merely to minimize the average E2EL. Across all request types, JITServe maintains strong P95 performance, highlighting the effectiveness of its conservative scheduling while avoiding starvation.

Another common concern in LLM scheduling is whether optimization comes at the expense of a small subset of requests. Our results show that JITServe avoids this pitfall: it significantly reduces tail latency (P95) compared to baselines like SJF and Autellix, indicating that it does not sacrifice much fairness for efficiency. By integrating deadline-awareness and service gain estimation, JITServe mitigates starvation and ensures that SLO-critical requests are not blocked by long-running tasks. In contrast, LTR shows very competitive E2EL on deadline-sensitive requests (Figure 3) as by design it prioritizes the request with potentially shortest response, thus achieving strong average E2EL performance. However, it struggles under diverse workloads: it oversubscribes bandwidth to deadline-sensitive requests which degrades overall goodput.

Breakdown by Components. Figure 17 presents a component-level breakdown of JITServe and several ablated variants: (1) JITServe with precise knowledge (JITServe*), (2) JITServe without the Request Analyzer (falling back to average response length estimation), (3) JITServe without *GMAX* scheduling (replaced by SJF scheduling based on Request Analyzer estimates), and Sarathi-Serve. Beyond the near-oracle performance achieved by JITServe in terms of SLO goodput, we observe that removing either the Request Analyzer or *GMAX* results in a noticeable degradation in goodput, highlighting their essential roles in the design.

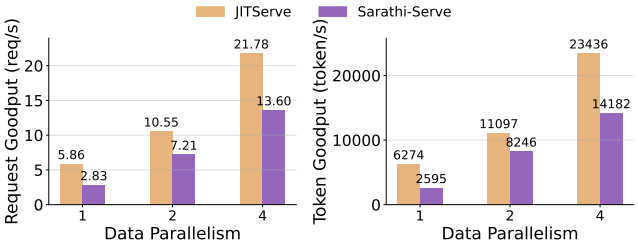


Figure 18: ***JITServe scales effectively to multi-model deployments.***

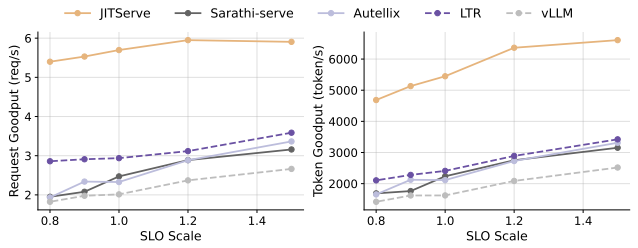


Figure 19: ***JITServe outperforms across various SLO tightness.***

6.4 Sensitivity and Ablation Studies

Extending to Multiple Models. We next investigate JITServe's performance when serving multiple model replicas with data parallelism. We scale the request arrival rates proportionally to the number of model replicas. As shown in Figure 18, while all systems achieve higher goodput with additional replicas, JITServe consistently outperforms the baseline by 1.34×–2.42× across all configurations.

Impact of SLO Constraints. We evaluate how JITServe responds when the SLO requirements are uniformly relaxed across all request types. The SLO constraints are scaled by a common factor (e.g., 0.8×, 1.5×), as users or applications may tolerate varying response times. As shown in Figure 21, relaxing SLO constraints naturally improves the SLO goodput. JITServe consistently improves both request and token goodput by 2.3×–2.8× over existing advances.

Impact of Workload Composition. We next study different workload compositions, under a wide range of workload mixes, including settings dominated by a single request type (e.g., 0% latency-sensitive workloads) and heterogeneous mixtures. Across all cases, Figure 20 shows that JITServe consistently achieves higher service goodput than existing approaches, such as 1.8× goodput improvement in a setting with 33% latency- and 66% deadline-sensitive workloads. Notably, JITServe outperforms Sarathi-Serve by 1.72× even for latency-sensitive-only requests, its intended design point.

Comparison to More SLO-aware Baselines. To further investigate the robustness of JITServe under multi-SLO constraints, we conducted a comparative analysis with SLOs-Serve [16], which utilizes dynamic programming for SLO-aware scheduling. Figure 21 illustrates the performance trend as the workload scales. While both systems maintain stable goodput under light loads, JITServe demonstrates superior

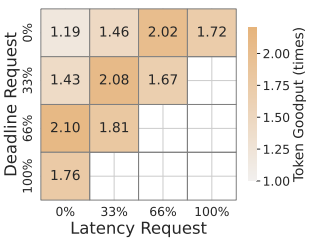


Figure 20: ***JITServe maintains higher performance across varying workload compositions.***

Figure 21: ***JITServe outperforms slo baseline.***

scalability as the RPS increases. This performance gap is primarily attributed to our GMAX policy's ability to leverage fine-grained request knowledge for proactive bandwidth allocation. Unlike the DP-based approach in SLOs-Serve, which may struggle with increased search complexity and rigid allocation under high contention, JITServe refines token generation predictions to ensure high-priority SLOs are met without sacrificing overall system efficiency.

7 Discussions

Limitations of the all-or-nothing goodput metric. JITServe adopts an all-or-nothing goodput definition that assigns zero value to requests missing their SLO deadlines, which aligns well with deadline-sensitive and compound workloads interacting with external tools. However, this abstraction does not capture scenarios where near-miss completions still provide partial utility. Supporting soft deadlines or graded goodput, where utility decays smoothly beyond the target, would enable finer-grained trade-offs. Importantly, JITServe and *GMAX* operate over an abstract goodput function and can directly accommodate such extensions (§3).

Robustness under workload distribution shifts and prediction errors. JITServe relies on conservative upper-bound estimation and progressive refinement of response lengths and execution patterns. Under significant workload distribution shifts or poor pattern matching, early scheduling decisions may be suboptimal, particularly in multi-tenant settings with conflicting SLO priorities. JITServe mitigates these effects through conservative initialization, online refinement, and frame-based rescheduling, which allow it to recover from mispredictions. Persistent shifts may require retraining predictors or incorporating explicit fallback policies.

GPU profiling overhead and system heterogeneity. JITServe does not perform runtime GPU profiling. Instead, we profile I/O bandwidth offline before deployment and only monitor lightweight iteration-level execution time during serving, which incurs negligible overhead. This design choice simplifies deployment and avoids introducing profiling-induced variance on the critical path. For more complex deployments, such as multi-node, highly parallel, multi-replica, or mixed-GPU environments, runtime profiling may become necessary to capture heterogeneous execution characteristics. In

such settings, JITServe could incorporate selective or coarse-grained runtime profiling to refine bandwidth and preemption cost estimates.

8 Related Work

LLM Serving System. Recent advancements in LLM have led to the development of numerous inference systems. Orca [87], dLoRA [81], VTC [66], and FastServe [80] present varied strategies for batching, concurrent serving, and fairness. vLLM [37] and InfiniGen [38] focus on KV-cache management to improve memory utilization and throughput. DistServe [94], Sarathi-Serve [10], Llumnix [68], and Splitwise [59] capitalize on characteristics of the prefilling and decoding stages to minimize TTFT and TBT latency. Some other systems explore GPU kernel optimizations [18, 19, 22, 53, 54, 65], model parallelism [12, 40], and preemptive scheduling [48] like HyGen [69] that elastically co-locates online and offline requests. However, most prior works primarily target single-request latency without accounting for the diverse requirements of different applications. Parrot [43] leverages the interconnections within LLM applications. JITServe builds on these approaches, categorizing LLM applications into three types and addressing their SLO requirements collectively. Parrot [43] offers APIs to extract LLM request execution dependency. JITServe builds on them, categorizing LLM applications and addressing their SLO requirements collectively.

LLM Output Length Prediction. Recent efforts such as TetriInfer [29], S^3 [34], and u-Serve [62] have proposed training multi-class classifiers (e.g., based on BERT) to predict LLM output length. However, these approaches are resource-intensive, both in training and during online inference. Moreover, our analysis reveals that existing classifiers exhibit significant deviations in prediction accuracy, ultimately leading to suboptimal scheduling decisions (§2). JITServe adopts a lightweight QRF model to estimate an upper bound on the output length, enabling conservative yet adaptive scheduling.

SLO-aware Resource Scheduling. Satisfying SLO requirements has long been a central challenge in resource scheduling. In networking, Karuna [15] performs deadline-aware flow scheduling to reserve bandwidth for best-effort flows, while QCLIMB [39] uses QRF models to predict lower bounds on flow sizes for flow scheduling. CASSINI [63] schedules ML training traffic across jobs to reduce network contention, and Caladan [23] mitigates resource contention in hyperthreads to reduce OS tail latency. More broadly, cluster and big-data schedulers such as Quincy [30], Sparrow [57], YARN [73], and Borg [74], as well as fairness-oriented policies like Dominant Resource Fairness [27], operate at coarse job or stage granularity with relatively stable workload assumptions, focusing on fairness, utilization, or average completion time. In contrast, JITServe targets fine-grained, latency-sensitive LLM inference, where request execution times are highly variable and partially unknown, requiring inference-specific scheduling that reasons about token-level progress and uncertainty. AdaServe [42] supports customizable SLOs through fine-grained speculative decoding, while JITServe complements it by orchestrating a broader range of heterogeneous SLO requirements across diverse LLM request types to maximize service goodput without precise request information.

9 Conclusion

We introduce JITServe, an LLM request scheduler designed to maximize service goodput. JITServe conservatively estimates request characteristics and incrementally refines these estimates. It employs a novel *grouped margin goodput maximization* algorithm that determines each request's minimum serving bandwidth needed while prioritizing requests with high grouped goodput payoff relative to their bandwidth when forming batches. Our evaluations across a variety of LLM applications and models demonstrate substantial improvements across a wide range of LLMs and applications.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ganesh Ananthanarayanan, for their constructive and insightful feedback. This work was supported in part by grants from Cisco, Google, Lambda, and by awards from NVIDIA and AMD Academic Programs. It also utilized the Delta system at the National Center for Supercomputing Applications (NCSA) through allocation CIS240236 from the ACCESS program.

References

[1] Bentoml: Key metrics for llm inference. https://bentoml.com/llm/inference-optimization/llm-inference-metrics.

[2] Deepsearcher: Open source deep research alternative. https://github.com/zilliztech/deep-searcher.

[3] Flash-decoding for long-context inference. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[4] Introducing perplexity deep research. https://www.perplexity.ai/hub/blog/introducing-perplexity-deep-research.

[5] Open deep research: An ai-powered research assistant. https://github.com/dzhng/deep-research.

[6] Openai: Introducing next-generation audio models in the api. https://openai.com/index/introducing-our-next-generation-audio-models/.

[7] Openai: The official python library for the openai api. https://github.com/openai/openai-python.

[8] vllm: A high-throughput and memory-efficient inference and serving engine for llms. https://github.com/vllm-project/vllm.

[9] Openai api pricing. https://openai.com/api/pricing/, 2025.

[10] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *OSDI*, pages 117–134, 2024.

[11] aisera. Ai customer service. https://aisera.com/products/ai-customer-service/, 2025.

[12] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[13] Amzon. Amazon q - generative ai assistant, 2024.

[14] Fabian Biester, Mohamed Abdelaal, and Daniel Del Gaudio. Llmclean: Context-aware tabular data cleaning via llm-generated ofds. In *European Conference on Advances in Databases and Information Systems*, pages 68–78. Springer, 2024.

[15] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 174–187, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. Slos-serve: Optimized serving of multi-slo llms, 2025.

[17] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. *SIGCOMM*, 2014.

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

[19] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[20] OpenAI DeepResearch. https://openai.com/index/introducing-deep-research/, 2025.

[21] DeepWisdom. Metagpt. https://www.deepwisdom.ai/metagpt, 2023.

[22] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *PPoPP*, pages 389–402, 2021.

[23] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*, 2020.

[24] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank. *NeurIPS*, 2024.

[25] Walter Gautschi. *Numerical analysis*. Springer Science & Business Media, 2011.

[26] Google Gemini. https://gemini.google.com/, 2025.

[27] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*, 2011.

[28] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

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

[30] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[31] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. ExtMem: Enabling Application-Aware virtual memory management for Data-Intensive applications. In *ATC*, 2024.

[32] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, Xiyang Wang, Nosayba El-Sayed, Jidong Zhai, Weiguo Liu, and Wei Xue. Automatic, Application-Aware I/O forwarding resource allocation. In *FAST*, 2019.

[33] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models

for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[34] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S^3: Increasing gpu utilization during generative inference for higher throughput. *NeurIPS*, 36:18015–18027, 2023.

[35] Jarod Kintz. Amortized analysis. 2018.

[36] Nils M Kriege, Fredrik D Johansson, and Christopher Morris. A survey on graph kernels. *Applied Network Science*, 5:1–42, 2020.

[37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, pages 611–626, 2023.

[38] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *OSDI*, pages 155–172, 2024.

[39] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. Flow scheduling with imprecise knowledge. In *NSDI*, 2024.

[40] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *OSDI*, pages 663–679, 2023.

[41] Zhuoyan Li, Hangxiao Zhu, Zhuoran Lu, and Ming Yin. Synthetic data generation with large language models for text classification: Potential and limitations. *arXiv preprint arXiv:2310.07849*, 2023.

[42] Zikun Li, Zhuofu Chen, Remi Delacourt, Gabriele Oliaro, Zeyu Wang, Qinghan Chen, Shuhuai Lin, April Yang, Zhihao Zhang, Zhuoming Chen, Sean Lai, Xupeng Miao, and Zhihao Jia. Adaserve: Slo-customized llm serving with fine-grained speculative decoding. *arXiv preprint arXiv: 2501.12162*, 2025.

[43] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *ODSI*, pages 929–945, 2024.

[44] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024.

[45] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for llm agents as general programs. In *arXiv:2502.13965*, 2025.

[46] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[47] Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

[48] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *ASPLOS*, pages 1112–1127, 2024.

[49] Mihran Miroyan*, Tsung-Han Wu*, Logan Kenneth King, Tianle Li, Anastasios N. Angelopoulos, Wei-Lin Chiang, Narges Norouzi, and Joseph E. Gonzalez. Introducing the search arena: Evaluating search-enabled ai, April 2025.

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

[51] James Newling and FranÃğois Fleuret. K-medoids for k-means seeding. In *NIPS*, 2017.

[52] Greg Ridgeway Nicolai Meinshausen. Quantile regression forests. *Journal of Machine Learning Research 7 (2006) 983-999*, 2006.

[53] Rui Ning, Wei Zhang, and Fan Lai. Packinfer: Compute- and i/o-efficient attention for batched llm inference. In *arXiv: 2602.06072*, 2026.

[54] NVIDIA. FasterTransformer. https://github.com/NVIDIA/FasterTransformer, 2023.

[55] OpenAI. Introducing chatgpt. https://openai.com/index/chatgpt/, 2022.

[56] OpenAI. Batch api. https://platform.openai.com/docs/guides/batch, 2024.

[57] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 69–84, 2013.

[58] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[59] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ISCA*, pages 118–132. IEEE, 2024.

[60] Shuofei Qiao, Runnan Fang, Zhisong Qiu, Xiaobin Wang, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Benchmarking agentic workflow generation. *arXiv preprint arXiv:2410.07869*, 2024.

[61] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. Power-aware deep learning model serving with Îij-Serve. In *ATC*, 2024.

[62] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Efficient interactive llm serving with proxy model-based sequence length prediction. *arXiv preprint arXiv:2404.08509*, 2024.

[63] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware job scheduling in machine learning clusters. In *NSDI*, 2024.

[64] Vighnesh Sachidananda and Anirudh Sivaraman. Erlang: Application-aware autoscaling for cloud microservices. In *EuroSys*, 2024.

[65] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[66] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *OSDI*, pages 965–988, 2024.

[67] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. Building ai agents for autonomous clouds: Challenges and design principles. In *SoCC*, 2024.

[68] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *OSDI*, pages 173–191, 2024.

[69] Ting Sun, Penghan Wang, and Fan Lai. Hygen: Efficient llm serving via elastic online-offline request co-location. In *NeurIPS*, 2025.

[70] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of llm-based applications with ayo. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1302–1316, 2025.

[71] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7, 2023.

[72] Qwen Team. Qwen3 technical report, 2025.

[73] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[74] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[75] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing cloud servers for lower carbon. In *ISCA*, 2024.

[76] Jiayin Wang, Weizhi Ma, Peijie Sun, Min Zhang, and Jian-Yun Nie. Understanding user experience in large language model interactions. *arXiv preprint arXiv:2401.08329*, 2024.

[77] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

[78] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. Intent-driven network management with multi-agent llms: The confucius framework. In *SIGCOMM*, 2025.

[79] Zhibin Wang, Shipeng Li, Yuhang Zhou, Xue Li, Rong Gu, Nguyen Cam-Tu, Chen Tian, and Sheng Zhong. Revisiting slo and goodput metrics in llm serving. In *arXiv: 2410.14257*, 2024.

[80] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

[81] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *OSDI*, pages 911–927, 2024.

[82] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[83] Chang Xiao and Brenda Yang. Streaming, fast and slow: Cognitive load-aware streaming for efficient llm serving. In *UIST*, 2025.

[84] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[85] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*, 36, 2024.

[86] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. Llm as a system service on mobile devices. *arXiv preprint arXiv:2403.11805*, 2024.

[87] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *OSDI*, pages 521–538, 2022.

[88] Zhaoyang Yu, Minghua Ma, Chaoyun Zhang, Si Qin, Yu Kang, Chetan Bansal, Saravan Rajmohan, Yingnong Dang, Changhua Pei, Dan Pei, Qingwei Lin, and Dongmei Zhang. Monitorassistant: Simplifying cloud service monitoring via large language models. In *FSE*, 2024.

[89] Yao Zhang, Zijian Ma, Yunpu Ma, Zhen Han, Yu Wu, and Volker Tresp. Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23378–23386, 2025.

[90] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. Wildchat: 1m chatgpt interaction logs in the wild, 2024.

[91] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. Lmsys-chat-1m: A large-scale real-world llm conversation dataset, 2023.

[92] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In *NeurIPS*, 2024.

[93] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *NeurIPS*, 36, 2024.

[94] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *OSDI*, pages 193–210, 2024.

LLM Applications	Real-Time	Direct Use	Content-Based
Code generation	**33.1%–42.9%**	25.7%–35.0%	27.1%–36.2%
Report generation	**34.5%–43.6%**	31.2%–40.5%	20.7%–29.0%
Deep research	34.8%–42.6%	**43.1%–51.2%**	11.5%–17.2%
Real-time translation	26.0%–34.8%	**37.6%–46.7%**	23.3%–31.9%
Batch data processing	13.8%–21.0%	**42.9%–52.4%**	30.7%–40.0%
Reasoning task	24.5%–36.8%	**41.1%–53.9%**	16.3%–27.5%

Table 3: ***Bootstrap 95% Confidence Intervals of User Interaction Preferences. Each cell reports the lower–upper bound of the estimated proportion (1,000 bootstrap resamples) for a given workload and interaction mode.***

A User Study Methodology

Participants self-identified as: 65.1% AI application users and 34.9% AI application developers. The majority (74.4%) reported using LLM tools multiple times per day, indicating experienced respondents. For each workload category, we computed the proportion of respondents preferring: (i) Real-time streaming interaction, (ii) waiting for full completion, and (iii) context-dependent behavior. Percentages reported in Table 1 are normalized over valid responses.

Bootstrap 95% Confidence Intervals To assess the robustness of user preference distributions reported in Table 1, we performed both bootstrap resampling and χ^2 significance tests. We computed 95% confidence intervals using bootstrap resampling (1,000 runs with replacement) for each (workload, action) pair. Table 3 reports the lower and upper bounds of the estimated preference proportions.

Across all workloads, the confidence intervals are sufficiently tight to confirm that the observed heterogeneity in responsiveness preferences is statistically stable rather than driven by sampling noise. For example, real-time translation exhibits a strong preference for streaming interaction, while batch data processing and reasoning tasks consistently favor direct completion, with non-overlapping or weakly overlapping intervals across action categories.

χ^2 Test Across Workloads To further evaluate whether preference distributions differ across workload types, we conducted χ^2 tests over the joint distribution of (workload $\times$ action), considering three action categories (Real-Time, Direct Use, Content-Based).

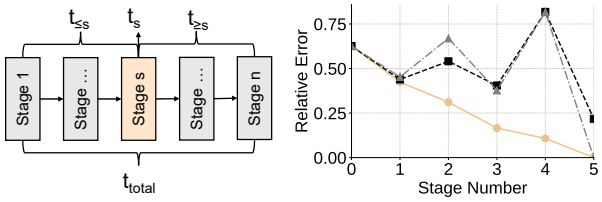
Results (Table 4) indicate that preference distributions are not uniform across workloads. In particular, several workloads, including code generation, deep research, and batch data processing, exhibit statistically significant deviations ($p < 0.01$), while others show no significant difference. This confirms that heterogeneous SLO expectations are systematically associated with task characteristics, rather than being randomly distributed across applications or driven by a specific user subgroup.

LLM Applications	χ^2	p-value
Code generation	21.40	2×10^{-5}
Report generation	9.47	9×10^{-3}
Deep Research	52.97	3×10^{-12}
Real-time translation	1.25	5×10^{-1}
Batch data processing	59.63	1×10^{-13}
Reasoning task	1.70	4×10^{-1}

Table 4: ***χ^2 Test Across LLM Workloads. Each row reports the χ^2 statistic and corresponding p-value when comparing the preference distribution of a given workload against the aggregated distribution over all workloads.***

B Pattern-Graph Matching Alternatives

For completeness, we also considered two alternative pattern-matching formulations (§4.1): setting D_s proportional to t_s/t_{total}, and setting D_s proportional to $t_s/t_{\geq s}$, where $t_{\geq s}$ is the accumulated time from stage s to the end. However, as shown in Figure 22(b), which reports relative error under online graph matching using traces from deepresearch requests, our design (orange curve) achieves substantially higher estimation accuracy than these alternatives.



(a) Illustration of $\phi(s)$.

(b) Estimation accuracy of different design.

Figure 22: ***Illustration and impact of different sub-deadline formulations.***

C Symbols and Notations

We will use the following notations for each request k: *input length* $L_i(k)$, *output length* $L_o(k)$, *start time* $s(k)$, *end time* $e(k)$, *computing time* $t_{\text{comp}}(k) > 0$, *service level objective (SLO) time* $t_{\text{SLO}}(k) > 0$, *remaining computing time* $t^{\text{r}}_{\text{comp}}(k) > 0$, *remaining time to service level objective (SLO)* $t^{\text{r}}_{\text{SLO}}(k) > 0$, *base goodput*

$$R(k) \;:=\; \omega_i L_i(k) + \omega_o L_o(k) \tag{1}$$

and *scheduling indicator*

$$I(k) \;:=\; \frac{R(k)}{t^{\text{r}}_{\text{comp}}(k) + \varepsilon} \tag{2}$$

together with $t^{\text{r}}_{\text{SLO}} - t^{\text{r}}_{\text{comp}}(k) \geq 0$ scheduling filter and $\varepsilon > 0$ to avoid division by 0 error. In our setting, a request will

only realize its goodput if and only if it completes by its SLO; otherwise, it will realize 0 goodput; We σ to denote a schedule, a sequence of served requests (R, t_R), where t_R refers to the served time of request set R in schedule σ. Note that since we allow preemption, some requests may get preempted and never complete; we use σ^c to denote the set of completed requests in σ and $\sigma^p \subset \sigma$ to denote the set of preempted requests. For a request k, denote by C_k its completion time (if it completes under σ), and by

$$\mathsf{Goodput}(\sigma) := \sum_{k: C_k \le t_{\mathrm{SLO}}(k)} R(k) \tag{3}$$

the realized total on-time goodput of schedule σ.

D Complexity Analysis

D.1 Optimal Scheduling

Theorem D.1 (NP-Hardness of optimal scheduling)**.** *Consider the following description of the previous LLM serving problem: given $n \in \mathbb{Z}_{>0}$ identical serving slots and a finite set of requests R, each request $r \in R$ specified by a computing time $t_{\mathrm{comp}}(k) > 0$, a start time $s(k)$, an SLO time $t_{\mathrm{SLO}}(k)$, and a goodput $R(k) > 0$ that is realized if and only if request r completes by its SLO on time, decide a schedule σ such that*

$$\sigma = \arg\max_{\sigma} \ \mathsf{Goodput}(\sigma) \tag{4}$$

Proof. It is well-known that the Multiple Knapsack Problem is NP-hard. We will reduce the Multiple Knapsack Problem to the LLM serving problem to show that the latter is NP-hard.

Multiple Knapsack Problem Given n knapsacks, each with a capacity $\mathcal{C}$, and a set of items I, each item $i \in I$ has a size $w_i > 0$ and a value $v_i > 0$, and a target value Ψ, determine whether there is a subset of items that can be assigned to the n knapsacks such that each knapsack's total size does not exceed $\mathcal{C}$ and the total value is at least Ψ.

Reduction Construction Given an instance of the Multiple Knapsack Problem, we construct an instance of the LLM serving problem as follows:

- Set the number of serving slots n equal to the number of knapsacks.
- For each item $i \in I$, create a request r_i with:

$$t_{\mathrm{comp}}(r_i) = w_i, \quad s(r_i) = 0, \quad t_{\mathrm{SLO}}(r_i) = C, \quad R(r_i) = v_i.$$

- Set the target total goodput to Ψ.

Correctness of reduction ($\Rightarrow$) If there is a solution to the Multiple Knapsack Problem, i.e., there exists a way to assign the items to the knapsacks such that each knapsack's total size does not exceed $\mathcal{C}$ and the total value is at least Ψ, then we can construct a valid schedule for the LLM serving problem. Each knapsack corresponds to a serving slot, and each item corresponds to a request. Since each knapsack has capacity $\mathcal{C}$, and the total size of items assigned to each knapsack is at most $\mathcal{C}$, the total processing time of requests in each slot does not exceed $\mathcal{C}$, and all requests are completed on time. Therefore, the total goodput is at least Ψ.

Correctness of reduction ($\Leftarrow$) Conversely, if there is a solution to the LLM serving problem such that the total goodput is at least $\mathcal{C}$, then we can assign items to knapsacks such that each knapsack's total size does not exceed $\mathcal{C}$, and the total value of the selected items is at least Ψ.

Thus, solving the LLM serving problem is equivalent to solving the Multiple Knapsack Problem. Since the Multiple Knapsack Problem is NP-hard, the LLM serving problem is also NP-hard. □

E Competitive Ratio Analysis

E.1 Analysis of Popular Scheduling Policies

E.1.1 Analysis of Earliest Deadline First Scheduling

Theorem E.1 (Non-competitiveness of EDF)**.** *The scheduling of Earliest Deadline First (EDF) is not competitive when compared with the optimal oracle scheduler: for any $r > 0$, there exists an input sequence σ such that*

$$\frac{\mathsf{Goodput}(\mathrm{OPT})}{\mathsf{Goodput}(\mathrm{EDF})} > r \tag{5}$$

Proof. We construct an input sequence σ consisting of multiple requests. Let $T > 0$ be a fixed time, and let N be a large positive integer. Define $\delta = \dfrac{T}{N+1}$. The request sequence includes:

- One request A that arrives at time 0, with computing time $t_{\mathrm{comp}}(A) = T$ and SLO time $t_{\mathrm{SLO}}(A) = T$, and goodput $R(A) = M$, where M is a large positive number to be chosen later.
- N requests B_i for $i = 0, 1, \cdots, N-1$, each arriving at time $t_i = i \cdot \delta$, with computing time $t_{\mathrm{comp}}(B_i) = \delta$ and SLO time $t_{\mathrm{SLO}}(B_i) = T + \delta$, and goodput $R(B_i) = 1$.

EDF scheduling. At time 0, request A arrives. Almost immediately, request B_0 arrives at time 0 with SLO time $\delta < T$, so EDF preempts A to serve B_0, which completes at time δ. At time δ, request B_1 arrives with SLO time $2 \cdot \delta < T$, so EDF preempts A to serve B_1, which completes at time $2 \cdot \delta$. This process continues: at each time $i \cdot \delta$, EDF schedules B_i, which completes at time $(i+1) \cdot \delta$. The last request B_{N-1} is scheduled at time $(N-1) \cdot \delta$ and completes at time $N \cdot \delta$. At time $N \cdot \delta$, no more B requests are available. EDF then schedules request A. However, the current time is $N \cdot \delta = \dfrac{N \cdot T}{N+1} < T$, and A requires computing time T. Thus, A completes at time $N \cdot \delta + T > T$, missing its SLO. Therefore, A contributes 0 goodput. And the total goodput for EDF is the sum of goodputs from all B requests: $\mathsf{Goodput}(\mathrm{EDF}) = N \times 1 = N$.

Optimal scheduling. It is clear to see that OPT will ignore all B_i requests and schedule request A starting at time 0. Since $t_{\text{comp}}(A) = T$ and its SLO is at time T, A completes exactly at time T, yielding $\mathsf{Goodput}(\text{OPT}) = M$.

Competitive ratio. Combining two cases above, we have the inverted competitive ratio

$$\frac{\mathsf{Goodput}(\text{OPT})}{\mathsf{Goodput}(\text{EDF})} = \frac{M}{N} \tag{6}$$

For any $r > 0$, choose $M > \frac{N}{r}$. Then $\frac{M}{N} > r$, proving the theorem. □

Remark. The classical Earliest Deadline First (EDF) policy is goodput-agnostic and therefore susceptible to adversarial workload constructions. In particular, an attacker can inject a stream of low-goodput requests whose *deadlines (SLOs)* are only marginally earlier than those of high-value jobs. EDF will systematically favor these low-value, tight-SLO requests, repeatedly preempting or delaying lucrative work and thereby degrading system-level utility.

E.1.2 Analysis of Shortest Job First Scheduling

Theorem E.2 (Non-competitiveness of SJF)**.** *The scheduling of Shortest Job First (SJF) is not competitive when compared with the optimal oracle scheduler: for any $r > 0$, there exists an input sequence σ such that*

$$\frac{\mathsf{Goodput}(\text{OPT})}{\mathsf{Goodput}(\text{SJF})} > r \tag{7}$$

Proof. We consturct an input sequence σ consisting of multiple requests. Let $T > 0$ be a fixed time, and let N be a large positive integer. Define $\delta = \frac{T}{N+1}$. The sequence includes:

- One request A that arrives at time 0, with computing time $t_{\text{comp}}(A) = T$ and SLO time $t_{\text{SLO}}(A) = T$, and goodput $R(A) = M$, where M is a large positive number to be chosen later.
- N requests B_i for $i = 0, 1, \cdots, N-1$, each arriving at time $t_i = i \cdot \delta$, with computing time $t_{\text{comp}}(B_i) = \delta$ and SLO time $t_{\text{SLO}}(B_i) = \delta$, and goodput $R(B_i) = 1$.

SJF scheduling. At time 0, request A arrives. Almost immediately, request B_0 arrives at time 0 with computing time $\delta < T$, so SJF preempts A to serve B_0, which completes at time δ. At time δ, request B_1 arrives with computing time $\delta < T$, so SJF serves B_1, which completes at time $2 \cdot \delta$. This process continues: at each time $i \cdot \delta$, SJF schedules B_i, which completes at time $(i+1) \cdot \delta$. At time $N \cdot \delta$, no more B requests are available. SJF then schedules request A. However, the current time is $N \cdot \delta = \frac{N \cdot T}{N+1} < T$, and A requires computing time T. Thus, request A completes at time $N \cdot \delta + T > T$, missing its SLO. Therefore, request A contributes 0 goodput.

Optimal scheduling. OPT will ignore all B_i requests and schedule request A starting at time 0. Since $t_{\text{comp}}(A) = T$ and its SLO is at time T, request A completes exactly at time T, yielding $\mathsf{Goodput}(\text{OPT}) = M$.

Competitive ratio. Combining two cases above, we have the inverted competitive ratio

$$\frac{\mathsf{Goodput}(\text{OPT})}{\mathsf{Goodput}(\text{SJF})} = \frac{M}{N} \tag{8}$$

For any $r > 0$, choose $M > \frac{N}{r}$. Then $\frac{M}{N} > r$, proving the theorem. □

Remark. Similarly, Shortest Job First (SJF) is indifferent to goodput and can be exploited by workloads populated with many low-goodput jobs of slightly shorter *computing times*. SJF will preferentially execute these short, low-value tasks, crowding out requests that are only marginally longer yet yield much higher goodputs, which leads to poor aggregate performance.

E.2 Analysis of JITServe Scheduling

We set the following condition as our additional preemption threshold: a newly considered request A may preempt the currently running request B if

$$\frac{R(A)}{R(B)} > 1 + \delta \qquad \text{for } \delta > 0 \tag{9}$$

We set this preemption threshold to avoid possible preemption overhead, and only a request with higher goodput may interrupt the current request.

Lemma 1 (Constant competitiveness of JITServe without GMAX)**.** *The scheduling of JITServe without GMAX is constant competitive when compared with the optimal oracle scheduler: there exists $r > 0$ such that*

$$\frac{\mathsf{Goodput}(\mathit{JITServe})}{\mathsf{Goodput}(\text{OPT})} \geq r \tag{10}$$

Proof. We use standard competitive analysis to evaluate our scheduling algorithm. And we will proceed our proof by using the *credit charging* technique from the amortized analysis. We first map the goodput of OPT to the requests served by JITServe via a carefully designed fractional credit charging mapping that respects their relative time overlap. Then we use the preemption-chain amortization to bound the total goodput of JITServe by the goodput of completed requests. And finally we combine the two pieces together to get the final competitive ratio.

Charging credits. We charge credits from each request $U \in \sigma_S$ to its corresponding request $V \in \sigma_*$, and we denote this credit charging rule as $f(U,V)$. In our setting, the systemâĂŹs service capacity $\mathfrak{C}$ is modeled as $\mathfrak{C}$ parallel service slots. For

expositional clarity, we first analyze the single slot case, i.e., schedules with $|R| = 1$. The extension to multiple slots is immediate, since the analysis applies independently to each slot. To better differentiate symbols and notations, we use U to denote the request in JITServe's schedule σ_S, and V to denote the request in the optimal schedule σ_*. Without loss of generality, we define the properties of a *chargable credit mapping* as follows:

- **Property 1:** For any request $U \in \sigma_S$, its aggregated charged credits to all requests in σ_* should be equal to the goodput of U, i.e., we have:

$$\sum_{V \in \sigma_*} f(U,V) = R(U), \quad \forall U \in \sigma_S \tag{11}$$

- **Property 2:** The aggregated charged credit from all $U \in \sigma_S^c$ to all $V \in \sigma_*$ must be greater than or equal to a constant portion of the aggregated goodput of $V \in \sigma_*$:

$$\sum_{V \in \sigma_*} \sum_{U \in \sigma_S^c} f(U,V) \geq r \sum_{V \in \sigma_*} R(V) \tag{12}$$

  where $r \in [0,1]$ is a constant.

Note that for any chargable credit mapping with the above two properties, we have the following lemma:

Lemma 2. *For any chargable credit mapping $f : (U,V) \to \mathbb{R}$ with a constant portion factor r in **Property 2**, we have:*

$$\frac{\mathsf{Goodput}(\sigma_S)}{\mathsf{Goodput}(\sigma_*)} \geq r \tag{13}$$

Proof of lemma.

$$\begin{aligned}
\mathsf{Goodput}(\sigma_S) &= \sum_{U \in \sigma_S^c} R(U) \\
&= \sum_{U \in \sigma_S^c} \sum_{V \in \sigma_*} f(U,V) \\
&= \sum_{V \in \sigma_*} \sum_{U \in \sigma_S^c} f(U,V) \\
&\geq \sum_{V \in \sigma_*} rR(V) \\
&= r \sum_{V \in \sigma_*} R(V) \\
&\geq r \cdot \mathsf{Goodput}(\sigma_*)
\end{aligned}$$

□

The key challenge here lies in constructing such a chargable credit mapping and quantifying such a good constant r.

With lemma 2, we are now ready define such credit charging mapping, which assigns to each ordered pair (U,V) a nonnegative charged credit $f(U,V)$ according to the following rules:

- **Rule 1:** If $s_S(V) \leq s_*(U)$, and $t^{\mathrm{r}}_{\mathrm{comp}}(V) > t^{\mathrm{r}}_{\mathrm{comp}}(U)$, we set $f(U,V) \coloneqq \alpha \cdot R(U)$.
- **Rule 2:** If $s_S(V) \leq s_*(U)$, and $t^{\mathrm{r}}_{\mathrm{comp}}(V) \leq t^{\mathrm{r}}_{\mathrm{comp}}(U)$, we set $f(U,V) \coloneqq \beta \cdot \dfrac{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon} \cdot R(U)$.
- **Rule 3:** If V and U share the same request, we set $f(U,V) \coloneqq \gamma \cdot (1+\delta)^3 \cdot R(U)$.
- **Rule 4:** Finally, if the aggregated credit mapped to V is less than $R(U)$, we assign the residual $R(U)$ to any arbitrary $V \in \sigma_*$.

It's clear to see that if we want this credit charging mapping f to be a *chargable* credit mapping, we will need to fix these three constants $\alpha \geq 0, \beta \geq 0, \gamma \geq 0$ with an additional condition that $\alpha+\beta+\gamma \leq 1$.

Lemma 3. *The credit charging mapping f satisties the above **Property 1** of a chargable credit mapping.*

Proof of lemma. For now, we assume that our credit charging mapping is confined to the above Rule 1, 2, and 3, and we have:

$$\begin{aligned}
&\sum_{V \in \sigma_*} g(U,V) \\
=\ &\sum_{\substack{V: s_*(V) \geq s_S(U) \\ t_{\mathrm{SLO}}(V) > t_{\mathrm{SLO}}(U)}} \alpha \cdot R(U) \\
&+ \sum_{\substack{V: s_*(V) \geq s_S(U) \\ t_{\mathrm{SLO}}(V) \leq t_{\mathrm{SLO}}(U)}} \beta \cdot \frac{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon} \cdot R(U) \\
&+ \sum_{V: V=U} \gamma \cdot R(U) \\
\leq\ &\sum_{\substack{V: s_*(V) \geq s_S(U) \\ t_{\mathrm{SLO}}(V) > t_{\mathrm{SLO}}(U)}} \alpha \cdot R(U) + \sum_{\substack{V: s_*(V) \geq s_S(U) \\ t_{\mathrm{SLO}}(V) \leq t_{\mathrm{SLO}}(U)}} \beta \cdot R(U) \\
&+ \sum_{V: V=U} \gamma \cdot R(U) \\
=\ &(\alpha+\beta+\gamma) \cdot R(U) \\
\leq\ &R(U)
\end{aligned}$$

To ensure **Property 1**, we can simply follow the Rule 4 to assign the residual credit to any arbitrary V. Therefore, we have:

$$\begin{aligned}\sum_{V\in\sigma_*} f(U,V) &= \sum_{V\in\sigma_*}\Big(g(U,V)+h(U,V)\Big)\\ &= \sum_{V\in\sigma_*} g(U,V)+\sum_{V\in\sigma_*} h(U,V)\\ &= (\alpha+\beta+\gamma)\cdot R(U)+(1-\alpha-\beta-\gamma)\cdot R(U)\\ &= R(U)\end{aligned}$$

where $h(U,V)\geq 0$ denotes the residual credit assigned by Rule 4 in the credit mapping f. □

Per-rule lower bounds. The remaining tasks now are all about how to bound a good constant r in **Property 2**. Fortunately, with the *credit charging* mapping technique, we can now focus on bounding the credit contribution to V from each $U\in\sigma_S$. Note that there's a brand-new challenge to address: there are two possible reasons why V may not be running at time $s_*(V)$. As a concrete example, consider the case where U starts at time t_U and V starts at time $t_V=t_U+\varepsilon>t_U$. On the one hand, V may be unable to preempt the request because U that is currently running. On the other hand, request V may have already ended in σ_S so there is no way for V to be served again in σ_S. We denote the unfinished request V as V^U, and the completed request V as V^C.

V^U Analysis In V^U analysis, we only consider the credit charging Rule 1 and 2. If V^U starts later than some request U, then V^U must be blocked by request U, hence we have the following two cases.

Case 1: We have $s_S(V)\leq s_*(U)$ and $t^{\mathrm{r}}_{\mathrm{comp}}(V)>t^{\mathrm{r}}_{\mathrm{comp}}(U)$. And therefore Rule 1 can be applied. According to the preemption threshold, we have two possible cases:

- **Case 1a:** V is not served because

$$\frac{R(V)}{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}\leq\frac{R(U)}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon} \tag{14}$$

Conbined with **Rule 1** and Eq. 14, we have:

$$\sum_{U\in\sigma_S} f(U,V)\geq\alpha\cdot R(U) \tag{15}$$

$$\geq\alpha\cdot\frac{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}\cdot R(V) \tag{16}$$

$$\geq\alpha\cdot R(V) \tag{17}$$

- **Case 1b:** V is not served because

$$\frac{R(V)}{R(U)}\leq 1+\delta \tag{18}$$

Combined with **Rule 1** and Eq. 18, we have:

$$\sum_{U\in\sigma_S} f(U,V)\geq\alpha\cdot R(U) \tag{19}$$

$$\geq\alpha\cdot\frac{R(V)}{1+\delta} \tag{20}$$

Case 2: We have $s_S(V)\leq s_*(U)$, and $t_{\mathrm{SLO}}(V)\leq t_{\mathrm{SLO}}(U)$. And therefore Rule 2 can be applied. According to the preemption threshold, we have two possible cases:

- **Case 2a:** V is not served because

$$\frac{R(V)}{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}\leq\frac{R(U)}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon} \tag{21}$$

Conbined with **Rule 2** and Eq. 21, we have:

$$\sum_{U\in\sigma_S} f(U,V)\geq\beta\cdot\frac{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon}\cdot R(U) \tag{22}$$

$$\geq\beta\cdot\left(\frac{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon}\right) \tag{23}$$

$$\cdot\left(\frac{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}\right)\cdot R(V) \tag{24}$$

$$\geq\beta\cdot R(V) \tag{25}$$

- **Case 2b:** V is not served because

$$\frac{R(V)}{R(U)}\leq 1+\delta \tag{26}$$

Combined with **Rule 2** and Eq. 26, we have:

$$\sum_{U\in\sigma_S} f(U,V)\geq\beta\cdot\frac{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\varepsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\varepsilon}\cdot R(U) \tag{27}$$

$$\geq\beta\cdot R(U) \tag{28}$$

$$\geq\beta\cdot\frac{R(V)}{1+\delta} \tag{29}$$

V^C Analysis Note that V^C must have been completed in σ_S. So based on the charging Rule 3, any request V must have been charged with a value of $\gamma\cdot R(U)$. Therefore, we have:

$$\sum_{U\in\sigma_S} f(U,V)\geq\gamma\cdot(1+\delta)^3\cdot R(U) \tag{30}$$

$$=\gamma\cdot(1+\delta)^3\cdot R(V) \tag{31}$$

Combining all the above cases, i.e., Eqs. 17, 20, 25, and 29, we have the following inequality:

$$\sum_{U\in\sigma_S} f(U,V) \geq \min\Big(\alpha, \frac{\alpha}{1+\delta}, \beta, \frac{\beta}{1+\delta}, \gamma\cdot(1+\delta)^3\Big)\cdot R(V) \tag{32}$$

Preemption chains. Based on the previous preemption threshold, we can now bound the total goodput of requests served by JITServe by the goodput of completed requests. Note that we can now partition the requests served by JITServe into chains:

$$U_1 \prec U_2 \prec \cdots \prec U_m,$$

where U_{k+1} directly preempts U_k. By the preemption threshold, we have

$$R(U_{k+1}) \;>\; (1+\delta)\cdot R(U_k) \tag{33}$$

hence by the property of geometric series,

$$\sum_{k=1}^{n} R(U_k) \leq \frac{1+\delta}{\delta}\cdot R(U_n) \tag{34}$$

Moreover, every chain terminates at a request $U_n \in \sigma_S^c$ (the last request in the chain completes on time). Summing over chains yields:

$$\sum_{U\in\sigma_S} R(U) \leq \frac{1+\delta}{\delta}\sum_{I\in\sigma_S^c} R(I) \tag{35}$$

$$= \frac{1+\delta}{\delta}\cdot \mathsf{Goodput}(\sigma_S) \tag{36}$$

Amortized charging bound. With Eq. 35, we have:

$$\mathsf{Goodput}(\sigma_S) \geq \frac{\delta}{1+\delta}\cdot\sum_{U\in\sigma_S} R(U) \tag{37}$$

$$= \frac{\delta}{1+\delta}\cdot\sum_{U\in\sigma_S}\sum_{V\in\sigma_*} f(U,V) \tag{38}$$

$$= \frac{\delta}{1+\delta}\cdot\sum_{V\in\sigma_*}\sum_{U\in\sigma_S} f(U,V) \tag{39}$$

Combining with the per-request bound Eq. 32, we have:

$$\mathsf{Goodput}(\sigma_S) \geq \frac{\delta}{1+\delta}\cdot\mathfrak{A}(\delta,\alpha,\beta,\gamma)\cdot\sum_{V\in\sigma_*} R(V) \tag{40}$$

$$= \frac{\delta}{1+\delta}\cdot\mathfrak{A}(\delta,\alpha,\beta,\gamma)\cdot\mathsf{Goodput}(\sigma_*) \tag{41}$$

where

$$\mathfrak{A}(\delta,\alpha,\beta,\gamma) \;=\; \min\Big(\alpha, \frac{\alpha}{1+\delta}, \beta, \frac{\beta}{1+\delta}, \gamma\cdot(1+\delta)^3\Big) \tag{42}$$

Therefore, we can conclude that there exists a constant competitive ratio for JITServe:

$$\mathfrak{B}(\delta,\alpha,\beta,\gamma) \;=\; \frac{\delta}{1+\delta}\cdot\min\Big(\frac{\alpha}{1+\delta}, \frac{\beta}{1+\delta}, \gamma\cdot(1+\delta)^3\Big) \tag{43}$$

Optimize objective function. It is clear that the maximum lower bound can be achieved by optimizing over $\delta,\alpha,\beta,\gamma$ under the constraints $\delta>0$, $\alpha\geq 0, \beta\geq 0, \gamma\geq 0$, and $\alpha+\beta+\gamma\leq 1$, i.e., formally, we need to solve the optimization problem to attain the maximum competitive ratio:

$$\max_{\delta,\alpha,\beta,\gamma} \quad \mathfrak{B}(\delta,\alpha,\beta,\gamma) \tag{44}$$

$$\text{s.t.} \quad \alpha+\beta+\gamma\leq 1 \tag{45}$$

$$\alpha\geq 0, \beta\geq 0, \gamma\geq 0 \tag{46}$$

$$\delta > 0 \tag{47}$$

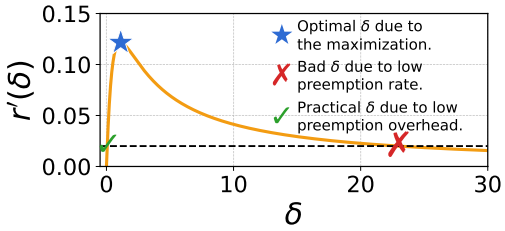


Figure 23: ***Competitive ratio*** $r'(\delta)$ ***versus preemption threshold*** δ

Solving these above optimization problem using *numerical analysis* [25] yields an optimal performance guarantee of JITServe $r'(\delta)\approx\frac{1}{8.13}$. Note that as we discussed in Section 4, preemption can improve the performance of our online algorithm JITServe, but it also introduces practical overhead. Figure 23 illustrates the trade-off between preemption overhead and the competitive ratio: the x-axis is the preemption threshold δ, and the y-axis is the performance guarantee of JITServe. To balance objective maximization and implementation overhead, we choose a moderate threshold of $\delta=10\%$, which slightly relaxes the bound yet yields a performance guarantee that remains acceptable in practice.

□

Theorem E.3 (Constant competitiveness of JITServe with GMAX)**.** *The scheduling of JITServe with GMAX is constant competitive when compared with the optimal oracle scheduler: there exists* $r>0$ *such that*

$$\frac{\mathsf{Goodput}(JITServe)}{\mathsf{Goodput}(\mathrm{OPT})} \;\geq\; r \tag{48}$$

Proof. We now continue from the above lemma to complete the proof. Note that we now have already had a performance guarantee for our JITServe scheduling algorighm under no-GMAX scenario. We are now ready to prove that our JITServe scheduling together with GMAX also has a performance guarantee.

Top-p Filtering We are now ready to prove that GMAX will only introduce a uniform $(1-\varepsilon)$-loss surrogate for each served request. Fix any batching decision time. Let $R(r(b))$ denote the b-th largest goodput value among all currently available requests. By the GMAX rule, the candidate set is

$$\mathcal{T} := \{W : R(W) \geq p \cdot R(r(b))\}. \tag{49}$$

Consider any request U that σ_S would serve (or continue to serve) at this time. Because σ_S maintains batch size b, necessarily $R(U) \geq R(r(b))$. Hence there exists some $U' \in \mathcal{T}$ with

$$R(U') \;\geq\; pR(r(b)) \;\geq\; pR(U). \tag{50}$$

Intuitively, U' is a *surrogate* for U that is always almost as valuable in terms of its goodput value; the length adjacency constraint in GMAX only determines *which* b requests inside $\mathcal{T}$ are taken, but it never drives any selected $R(\cdot)$ below the threshold $p \cdot R(r(b))$. Therefore, replacing U by U' in any lower-bound argument causes at most a multiplicative p degradation.

Putting together. Combining Eq. 43 from Lemma 1 and Eq. 50, we now have

$$\mathsf{Goodput}(\sigma_S) \geq \frac{p \cdot \delta}{1+\delta} \cdot \mathfrak{B}(\delta, \alpha, \beta, \gamma) \cdot \mathsf{Goodput}(\sigma_*) \tag{51}$$

Solving these above optimization problem using *numerical analysis* [25] under our experimental setting yields an optimal performance guarantee of JITServe $r(\delta) \approx \frac{1}{8.557}$.

□