QoServe: Breaking the Silos of LLM Inference Serving

Kanishk Goel
Microsoft Research
Bengaluru, India
t-kangoel@microsoft.com

Jayashree Mohan
Microsoft Research
Bengaluru, India
jamohan@microsoft.com

Nipun Kwatra
Microsoft Research
Bengaluru, India
nkwatra@microsoft.com

Ravi Shreyas Anupindi
Microsoft Research
Bengaluru, India
ravianupindi@microsoft.com

Ramachandran Ramjee
Microsoft Research
Bengaluru, India
ramjee@microsoft.com

Abstract

The widespread adoption of Large Language Models (LLMs) has enabled diverse applications with very different latency requirements. Existing LLM serving frameworks rely on siloed infrastructure with coarse-grained workload segregation — interactive and batch — leading to inefficient resource utilization and limited support for fine-grained Quality-of-Service (QoS) differentiation.

We present QoServe, a novel QoS-driven inference serving system that enables efficient co-scheduling of diverse workloads on shared infrastructure. QoServe introduces fine-grained QoS classification allowing applications to specify precise latency requirements, and dynamically adapts scheduling decisions based on real-time system state. Leveraging the predictable execution characteristics of LLM inference, QoServe implements dynamic chunking to improve overall throughput while maintaining strict QoS guarantees. Additionally, QoServe introduces hybrid prioritization to balance fairness and efficiency, and employs selective request relegation for graceful service degradation during overloads. Our evaluation demonstrates that QoServe increases serving capacity by 23% compared to current siloed deployments, while maintaining QoS guarantees on an A100 cluster, and improves per-replica goodput by up to 2.4x compared to Sarathi on a shared cluster. Notably, under extreme load, our system reduces SLO violations by an order of magnitude compared to current strategies.

CCS Concepts: • **General and reference** → *Reliability*; *Metrics*; • **Computing methodologies** → *Neural networks*; • **Software and its engineering** → **Software reliability**; **Scheduling**.

Keywords: Large Language Models; Quality of Service; Inference serving; Scheduling; Reliability

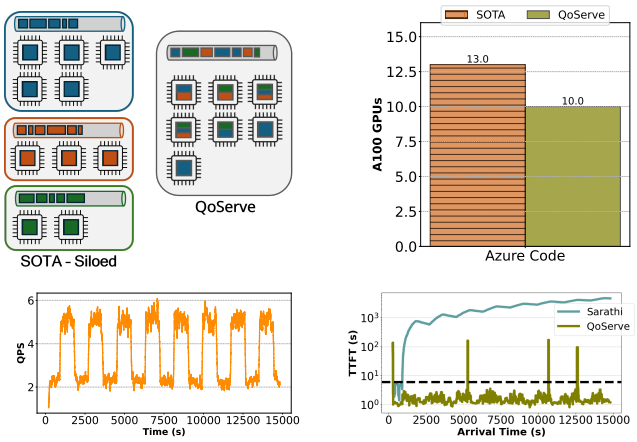


Figure 1. Efficiency of QoServe under uniform load and transient overload. (top left) Illustration of QoServe co-scheduling vs current siloed deployments. (top right) A100 GPUs needed to serve a fixed load of 35QPS while meeting the QoS targets of requests divided equally among 3 QoS tiers in a real cluster. QoServe improves efficiency by 23% compared to the state-of-the-art Sarathi [4] siloed deployment. (bottom left) Bursty overload scenario. (bottom right) QoServe maintains low latency while SOTA scheduling succumbs to cascading deadline violations under bursty loads.

ACM Reference Format:
Kanishk Goel, Jayashree Mohan, Nipun Kwatra, Ravi Shreyas Anupindi, and Ramachandran Ramjee. 2026. QoServe: Breaking the Silos of LLM Inference Serving. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790206

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790206

1 Introduction

Large language models (LLMs) have transformed applications across diverse domains including conversational assistants, coding assistants, content generation, and summarization. These applications can have very different latency requirements — for example, autocomplete coding assistants

demand responses within milliseconds, while summarization tasks can reasonably tolerate longer latencies. As LLM deployments scale to serve billions of users and diverse applications, inference serving systems must efficiently handle this diverse spectrum of latency requirements while ensuring high GPU utilization.

Current LLM serving solutions primarily adopt a coarse-grained categorization, segregating requests into two broad service classes: latency-sensitive interactive applications, and throughput-oriented batch processing, and serve them independently [9]. Interactive requests are typically served with smaller prefill chunks [4] to minimize latency, but that can result in relatively higher operational costs due to reduced throughput (e.g., 28% lower as shown in Figure 4). Batch requests, on the other hand, employ larger chunks to achieve higher throughput as latency is not a constraint. This siloed deployment, however, creates other inefficiencies: it leads to significant GPU resource under-utilization, as workload demands fluctuate across the two classes. Moreover, such partitioning inhibits the introduction of more QoS classes with fine-grained latency requirements, as doing so further exacerbates the partitioning inefficiencies.

Furthermore, current inference systems struggle under load fluctuations and overload conditions. Typical scheduling mechanisms such as first-come-first-served (FCFS) indiscriminately delay all incoming requests under overload, degrading user experience across the board. Alternatively, naïve throttling approaches reject all new incoming requests when reaching capacity, ignoring their QoS requirements or relative priorities. Neither strategy adequately manages the complex trade-offs between throughput, latency, and fairness during such demand surges.

In this paper, we present QoServe, a QoS-driven LLM inference serving system that addresses these limitations through two key ideas. First, QoServe supports fine-grained QoS classes which allows applications to precisely specify their latency requirements. Multiple QoS classes are served efficiently by *co-scheduling requests with diverse QoS targets on a shared rather than siloed infrastructure*. Second, QoServe implements a hybrid prioritization and an eager relegation policy that *allows graceful service degradation during overload conditions*. Figure 1 compares QoServe to state-of-the-art Sarathi-Serve [4] siloed deployment, demonstrating significant performance improvements.

Efficiently supporting multiple QoS classes on a shared serving instance poses significant challenges. One approach is to use the smallest chunk size necessary to meet the latency constraint of the strictest QoS class on all serving instances. However, this would result in low throughput [4] and high cost for all service classes. Instead, QoServe leverages the unique execution characteristics of LLM inference – particularly the distinct prefill and decode phases and the inherent predictability of the prefill phase – to dynamically adjust chunk sizes based on the observed system state and individual QoS targets. Co-serving multiple QoS classes allows us to exploit *deadline slack* of requests with relaxed latency requirements to schedule bursts of larger chunk sizes, thereby increasing throughput opportunistically.

For managing overload conditions gracefully, QoServe employs a hybrid prioritization and an eager relegation policy. Simple overload handling approaches like shortest-job-first (SJF) manage overload by prioritizing short requests. This helps reduce load due to the quadratic dependence of request length on LLM system load [17]. However, SJF neglects the QoS requirements of longer jobs, leading to SLO violations even at low load (Figure 2). On the other hand, Earlier Deadline First (EDF) scheduling is optimal under low load but suffers excessive violations even when load is slightly higher than capacity. Thus, QoServe introduces a hybrid policy that smoothly interpolates between EDF and SJF, allowing deployments to minimize SLO violations across both low and high load. Additionally, QoServe proactively employs *eager relegation*, selectively degrading service for a small subset of requests to ensure stable performance, even under extreme load conditions. In multi-QoS scenarios, QoServe leverages application-provided hints about request importance, such as whether a request originates from a free or paid tier, to perform relegation. This ensures that lower-priority requests are affected first during overload conditions, allowing the system to maintain QoS for the majority of high-priority requests. Our evaluations show that during significant overload scenarios (50% above capacity), QoServe consistently meets latency targets for over 95% of requests, translating into substantial cost savings and enhanced user experience across diverse applications relying on LLM infrastructure.

Our work makes the following key contributions:

1. We develop a QoS-aware adaptive scheduling algorithm that exploits the unique characteristics of LLM inference to co-schedule requests belonging to multiple QoS classes on shared infrastructure, improving throughput while maintaining latency guarantees.
2. We design and implement a hybrid prioritization and eager relegation policy that minimizes SLO violations under both optimal load and overload conditions.
3. We evaluate QoServe across workloads and scenarios, demonstrating up to 32% higher serving capacity while meeting QoS guarantees compared to baseline.

The rest of the paper is structured as follows. (§2) outlines the need for QoS-based serving systems and (§3) details the architecture and implementation of QoServe. (§4) presents our evaluation methodology and results.

2 Background and Motivation

2.1 LLM Inference

Large language model (LLM) inference is fundamentally different from traditional computing workloads, characterized

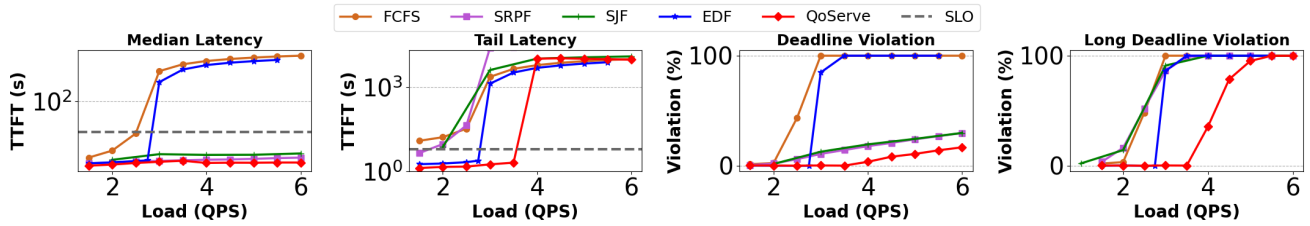


Figure 2. Comparison of traditional policies for multi-SLA scheduling. The graphs plot the latency and violations in the strictest QoS class. FCFS breaks down very quickly because urgent requests can be stalled by non-urgent ones. Deadline-aware policies like EDF are better than FCFS, but cannot gracefully degrade at high loads because of intense queue buildup. SJF/SRPF on the other hand can maintain QoS in the median case but violates SLOs of the majority of long jobs even at a low load of 2.5 QPS. QoServe interpolates smoothly between SJF and EDF and minimizes violations across all load conditions.

by two distinct computational phases that significantly impact system design: the prefill and decode stages. During the prefill phase, the entire input prompt is processed simultaneously, making it computationally intensive. The subsequent decode phase generates output tokens auto-regressively, with each token's generation depending on the previously generated tokens.

Scheduling. In this work, we assume co-located LLM inference scheduling as seen in popular serving frameworks like vLLM [11] and SGLang [21] where prefills and decodes of a request are executed on the same replica using chunked prefills [4] for better serving efficiency. Chunked prefills split a prefill request into equal-sized chunks, allowing for efficient batching and scheduling without pausing ongoing decodes. This approach helps balance the trade-off between throughput and latency, and is used as a standard scheduling practice in production systems [18].

Latency metrics. LLM inference encompasses three primary latency metrics, which serve as critical performance indicators across different application types:

1. **Time to First Token (TTFT).** This metric captures the initial response latency, measuring the duration from request submission to generating the first output token. For interactive applications like chatbots and coding assistants, TTFT is crucial as it directly influences user perception of system responsiveness.
2. **Time Between Tokens (TBT).** This metric measures the interval between the generation of consecutive output tokens of a request, and affects the overall perceived fluidity of the response which is particularly important for interactive applications where users expect a smooth, uninterrupted stream of generated content.
3. **Time to Last Token (TTLT).** This metric focuses on the total time required to complete the entire generation process. TTLT is particularly relevant for non-interactive, batch-oriented applications such as document summarization, comprehensive research analysis, or offline content generation. In these scenarios, the overall completion time matters more than the speed of initial response or token-by-token generation.

The application's nature determines which of these metrics take priority. User-facing, interactive applications critically depend on both TTFT and TBT, as these metrics directly impact user experience and perceived system responsiveness. In contrast, non-interactive applications primarily concern themselves with TTLT, prioritizing the total time to generate a complete output over the speed of initial token generation.

2.2 Production Deployment Landscape

Due to the fundamental differences in workload characteristics and performance requirements between these application types, current industrial practices for LLM inference deployment predominantly employ a siloed infrastructure model [9], maintaining two distinct GPU clusters: (1) a dedicated fleet for latency-sensitive, interactive requests, and (2) a separate cluster for batch processing and background jobs.

Overload management. When faced with traffic exceeding capacity, current systems employ limited and often ineffective overload management techniques.

1. Rate Limiting: These mechanisms simply reject excess requests without considering their relative importance or potential impact.
2. Short Request Prioritization: These techniques favor shorter requests, which can unfairly disadvantage longer but potentially more important queries.

Such approaches are unable to provide application-aware or graceful service degradation, resulting in either uniform performance degradation across workloads or complete rejection of a class of requests without any fairness guarantees.

2.3 Deployment Challenges

Current LLM deployments create significant operational inefficiencies due to the siloed infrastructure model.

Resource provisioning and utilization. As workload demands fluctuate, dedicated clusters often operate well below their maximum capacity, resulting in substantial resource underutilization. An interactive cluster might be overwhelmed

during peak hours, while a batch processing fleet remains largely idle, leading to inefficient computational resource allocation. The complexity intensifies when supporting applications with multiple different latency requirements. Each unique performance profile potentially necessitates a dedicated infrastructure cluster, which can increase operational complexity significantly. What begins as a straightforward architectural decision quickly transforms into a management challenge, with each new cluster introducing additional capacity provisioning challenges and monitoring overhead.

Lack of graceful service degradation. Existing mechanisms for overload management, such as user rate limiting and prioritizing short requests, are often unfair and not application-aware, and thus lack an ability to gracefully degrade QoS. These techniques can lead to poor user experiences and inefficient resource utilization.

2.4 Analysis of Multi-SLA scheduling policies

A practical approach to mitigating the operational complexities and resource inefficiencies of siloed infrastructure is to co-schedule requests from various applications within a unified cluster. In this section, we examine the effects of traditional scheduling policies from the literature on multi-tenant scheduling and assess their performance for LLM inference across three key dimensions: latency, SLO violations, and the fairness of SLO violations. This analysis highlights the necessity for a novel multi-tenant, SLO-aware scheduling policy tailored for LLM inference.

Scheduling policies. We compare four different scheduling policies from the literature for multi-tenant systems. First-Come-First-Served (FCFS) represents the most basic approach, processing requests in the order they arrive. More advanced policies include Shortest Job First (SJF), which prioritizes jobs with the shortest expected execution time, and Shortest Remaining Prompt First (SRPF), which continuously re-evaluates and preempts jobs to minimize overall waiting time, based on the outstanding prompt tokens to be processed. Finally, Earliest Deadline First (EDF) schedules jobs based on their impending deadlines.

Figure 2 compares the multiple scheduling policies and plots the (a) median and (b) p99 latency of requests in the system, (c) percentage of requests that violated their SLO, and (d) the number of long requests (requests with prompt length in the 90th percentile of the dataset) that violated their SLO. Despite their theoretical foundations, we observe that these scheduling approaches fundamentally struggle when applied to large language model (LLM) inference workloads. QoServe exploits the unique computational characteristics of LLMs — including variable input complexity, distinct prefill and decode phases, and the predictability of the prefill phase to devise an SLO-aware scheduling policy which minimizes latency and SLO violations while maximizing throughput, as we show in our evaluations (figs. 10 and 11).

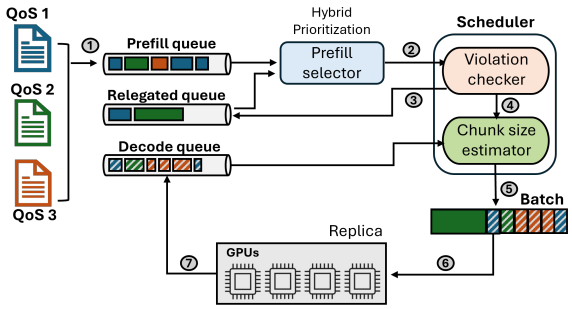


Figure 3. Overview of QoServe

Summary. In this paper, we address these critical infrastructure challenges by introducing a QoS-aware serving framework, QoServe. Our system transforms LLM inference serving from a static, siloed approach to a dynamic, application-aware computational system. By introducing sophisticated service level objective (SLO) management, QoServe enables more efficient, responsive, and cost-effective infrastructure for next-generation AI applications.

3 QoServe: Design and Implementation

QoServe is designed to efficiently manage concurrent LLM inference requests with diverse QoS requirements, while maximizing resource utilization across the shared infrastructure. We address the limitations outlined earlier by dynamically adapting scheduling decisions based on real-time system state and QoS targets of the in-flight requests.

3.1 Overview

The architecture of QoServe is shown in Figure 3. A request in QoServe can be in one of three queues — 1) prefill queue, 2) decode queue, or 3) relegated queue. ① When a request enters the system, it is put into the prefill queue. In each iteration, QoServe constructs a batch consisting of all requests in the decode queue and a prefill-chunk from a request in the prefill queue. The prefill selector uses *hybrid prioritization* to select the prefill request for the current batch. ② The violation checker module validates that the chosen request has not already violated (or will not violate) its QoS targets in the current iteration. ③ If it does, it is eagerly moved into the relegated queue and a different prefill request is chosen. The relegated requests are serviced opportunistically during periods of lower system load, ensuring eventual completion without permanent rejection; while enabling graceful degradation under overload conditions. ④ A lightweight predictor is then used to estimate the latency of the batch to make sure that the QoS targets are not violated, while maximizing the chunk size for efficiency. ⑤ A mixed batch of prefill and decode tokens is constructed using the chosen prefill chunk and the requests in the decode queue, which is then

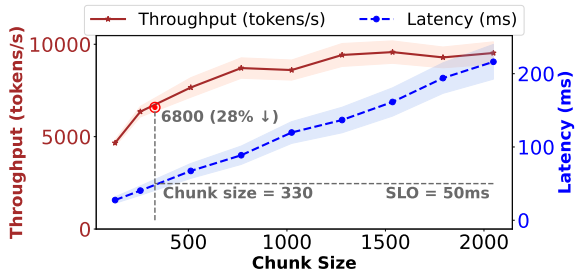


Figure 4. Performance characteristics as a function of chunk size, showing the throughput-latency tradeoff.

⑥ dispatched to the execution engine on the GPU for processing. ⑦ Once the prefill portion of a request is completed, it is moved to the decode queue, and subsequent iterations continue.

3.2 QoS Classes and Deadlines

QoServe defines two QoS classes: interactive and non interactive. Interactive requests use two SLOs — TTFT (time to first token) and TBT (token-by-token latency), which ensure immediate responsiveness and consistent pacing. Non-interactive requests have a single TTLT (total latency) target, focused on overall completion. Although, we define two QoS classes, the application owner is free to specify their custom SLO targets within the class, allowing for flexibility and customization to specific application needs as shown in Table 3.

The deadline for each request is determined based on its QoS class. For the interactive QoS class, following the approach in [2], the deadline for the first token is defined as:

$$D_{first} = t_{arrival} + SLO_{TTFT}, \quad (1)$$

while subsequent tokens' deadlines are calculated using:

$$D_n = t_{arrival} + SLO_{TTFT} + (n-1) \cdot SLO_{TBT}, \quad (2)$$

where n is the token position. For non-interactive requests a deadline is set only for the full completion of the request as:

$$D_{total} = t_{arrival} + SLO_{TTLT} \quad (3)$$

Once we have defined the deadlines for each request, QoServe scheduling aims to minimize deadline violations while maximizing throughput.

3.3 Dynamic Chunking

State-of-the-art LLM inference serving frameworks [4, 11] serve requests using chunked-prefills, where each iteration processes a fixed number of tokens (called chunk size), which includes both prefill and decode tokens from different requests using fused prefill-decode MLP to improve the compute efficiency of memory-bound decode phase [5]. However, this involves a fundamental trade-off between throughput and latency – a larger chunk results in better throughput but increases the TBT of the decodes in the batch. This is illustrated in Figure 4.

A naïve approach for co-scheduling jobs of different QoS classes with deadlines on TTFT, TBT, and TTLT would be to use the smallest chunk size necessary to meet the latency constraint of the strictest QoS class. However, this results in low throughput and high cost for all service classes.

QoServe employs *dynamic chunking* to opportunistically maximize the chunk size for the prefill request by exploiting any slack in the deadlines of the requests being currently serviced. For each request in the decode queue, we define slack as the difference between the deadline for the next token (Eq. 2) and current time. Using this slack and characteristics of the requests in decode phase, we calculate the chunk size which maximizes throughput under the given latency budget. We elaborate on the design in Section 3.6.1

3.4 QoServe Scheduling

While dynamic chunking allows us to choose an optimal chunk size for a prefill request, we also need to decide which request from the prefill queue should be processed in the current scheduling iteration.

Hybrid Prioritization. As shown in Figure 2, existing scheduling policies struggle with LLM workloads at higher loads. For example, EDF which prioritizes requests with earlier deadline has very low deadline violation rates (Figure 2(c)) at low loads, but the violation rates spike to almost 100% once the load exceeds a certain threshold. On the other hand, policies which prioritize short work requests — SRPF and SJF — handle higher loads much better but are worse than EDF at lower loads. Further, SRPF and SJF achieve this at the expense of unfairly penalizing long jobs (Figure 2(d)) without any regard to the request priorities. To handle varying load conditions which are common in production services and maintain fairness across requests, our first key insight is a *hybrid prioritization* scheme which interpolates between SRPF and EDF. This allows us to get EDF characteristics at low loads, and leverage SRPF semantics under overload conditions while maintaining fairness.

To implement this scheduling, QoServe smoothly interpolates between EDF and SRPF to compute the priority of a request. For interactive requests, the priority is computed by taking a linear combination of the TTFT deadline (this incorporates EDF semantics) and the estimated time taken which will be needed to process the remaining prefills (this incorporates SRPF semantics) of the request as:

$$P^i = t^i_{arrival} + SLO^i_{TTFT} + \alpha * Prefill^i_{rem}. \quad (4)$$

Note that we only consider the TTFT deadline, as TBT deadlines are maintained by our dynamic chunking scheme. For non-interactive requests, the priority is computed as

$$P^i = t^i_{arrival} + SLO^i_{TTLT} + \alpha * (Prefill^i_{rem} + Decode^i_{rem}), \quad (5)$$

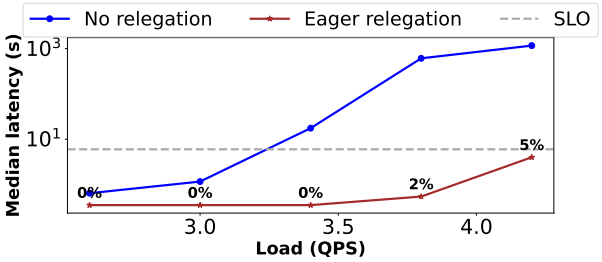


Figure 5. Proactively relegating a small percentage of requests enormously helps in maintaining the quality of service for the median request in the system, which otherwise grows exponentially due to a cascade of violations.

where $Decode_{rem}$ indicates the time to compute all the decode tokens. Since decode length is unknown in LLM inference, this introduces a challenge in modeling the priority of non-interactive requests. We address this with a simple insight — for non-interactive jobs, the TTLT deadline is typically much greater than the actual processing time. Therefore, given an application, we can use historic information on the decode tokens generated by that application and over-approximate it by two standard deviations. We show in (§4.4.1) that this simple prediction sufficiently captures the priority of non-interactive jobs.

Eager Relegation. Our *hybrid prioritization* strikes a balance between minimizing deadline violations and fairness. However, as shown in Figure 5, under overload conditions, QoServe (or any scheduling policy) still cannot service all incoming requests at the desired QoS SLOs. Our second key insight is that by *eagerly relegating* a small fraction of requests that we know will miss their deadlines, one can provide stable performance for the majority, enabling graceful service degradation under overload conditions. The key idea is simple — if a request has already violated its TTFT / TTLT deadline, or is about to violate it in the current iteration, then QoServe de-prioritizes this request into a relegated queue. In multi-tenant deployments, we also use application hints such as free vs paid tier to preferentially relegate low-priority requests to ensure stability of service to the high priority ones. Only when there are no more low-priority requests, QoServe proactively relegates high-priority requests that have violated their deadlines to prevent cascading deadline violations. This enables graceful degradation of service even under extreme load. As shown in Figure 5, by relegating just 5% of the requests, we can maintain latency SLOs even under very high overload conditions.

Selective Preemption. Note that our *hybrid prioritization* scheduling can preempt an in-flight request for which a few prefill chunks have already been processed to instead service a new request with strict QoS target (see eq 4). Preemption

Algorithm 1 Dynamic Batch Creation Algorithm

```
function CREATE_BATCH
    selected_jobs ← GET_ALL_DECODES
    batch_decode_context ← GET_DECODE_CONTEXT(selected_jobs)
    num_decodes ← selected_jobs
    min_decode_slack ← GET_MIN_SLACK(selected_jobs)
    // Below invokes the predictor model to find dynamic chunk size
    C ← GET_PREFILL_BUDGET(num_decodes, batch_decode_context, min_decode_slack)
    job_queue ← PRIORITY_QUEUE(COMPARATOR)
    prefill_token_count ← 0
    while prefill_token_count < C do
        top_job ← job_queue.TOP
        if WILL_VIOLATE(top_job) then
            UPDATE_RELEGATE_STATUS(top_job, true)
            job_queue.PUSH(top_job)
            continue
        else
            curr_job_tokens ← min(C − prefill_token_count, REM_TOKENS(top))
            prefill_token_count ← prefill_token_count + curr_job_tokens
            top_job.prefill_tokens_taken ← curr_job_tokens
            selected_jobs.APPEND(top_job)
            job_queue.POP
        end if
    end while
    PROCESS_BATCH(selected_jobs)
end function
function COMPARATOR(job1, job2)
    if job1.drop_status ≠ job2.drop_status then
        return job1.drop_status < job2.drop_status
    end if
    priority1 ← job1.arrival_time + job1.TTFT_SLO + α × job1.rem_prefill_tokens
    priority2 ← job2.arrival_time + job2.TTFT_SLO + α × job2.rem_prefill_tokens
    return priority1 < priority2
end function
```

is a desirable capability as it avoids head-of-line blocking of small interactive requests behind long batch requests. However, in LLM serving, the memory overhead of preemption can be significant as the KV-cache of requests can be large. To avoid this, QoServe uses *selective preemption*, where we preempt a request to accommodate another with a higher priority only if (1) the in-flight request is in the prefill queue (i.e., requests in the decode queue are never preempted), and (2) preempting that request for an iteration does not lead to deadline violation. We do not preempt requests in the decode queue as TBT targets are typically strict (10s of ms), and thus preempting them significantly increases the chances of TBT violation. This also ensures that the KV-cache for each request remains in the GPU for the shortest necessary duration, thereby minimizing memory pressure. The pseudocode for hybrid batch creation and prioritization in QoServe is presented in Algorithm 1.

3.5 An Illustrative Example

Figure 6 illustrates QoServe with an example of five requests (*A*–*E*) across 3 QoS buckets. *A* is an interactive request while others are non-interactive. State-of-the-art LLM schedulers like vLLM [11] and Sarathi [4] will execute each iteration using a fixed chunk size and process requests in arrival order (FCFS). Our solution introduces two key improvements.

First, we prioritize requests based on their QoS targets using our hybrid prioritization, which will prioritize request *A* before *D* due to its earlier deadline. Second, we dynamically adjust chunk sizes based on accumulated slack. For example,

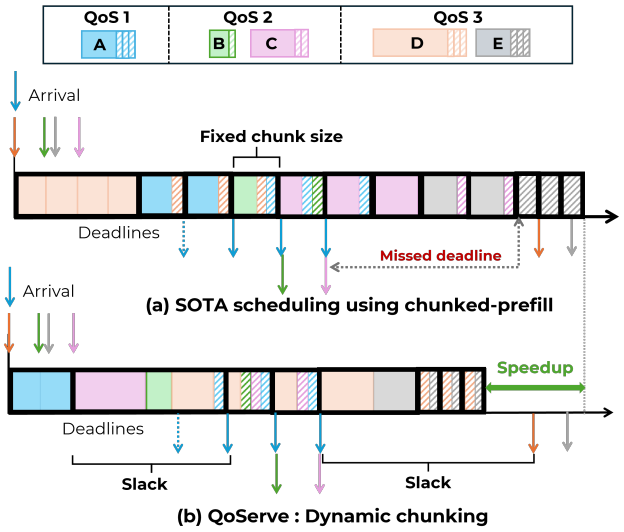


Figure 6. An illustration of how QoServe improves throughput using dynamic chunking compared to SOTA scheduling.

after A's prefill phase completes earlier than its deadline, it accumulates significant slack before its next token is due. We exploit this slack by dynamically increasing the chunk size, adding more prefill tokens from requests B and D (which have the earliest deadlines in the queue), thereby improving throughput without violating any ongoing request deadlines.

When the interactive job A enters its decode phase, we revert to the original smaller chunk size necessary to meet its TBT, though we still exploit slack accumulated if decoding completes faster than predicted. Once A completes and no remaining requests impose strict TBT constraints, we again increase chunk size to maximize throughput while respecting the TTLT deadlines of ongoing requests. This approach effectively leverages the deterministic execution characteristics of LLMs to dynamically optimize chunk sizes during runtime, balancing throughput and deadline requirements.

3.6 Implementation

We implemented QoServe by extending the Sarathi scheduler [4], which is built on top of the vLLM inference system [11]. The implementation focuses on enhancing the scheduler component while maintaining compatibility with vLLM's efficient tensor parallelism and PagedAttention mechanisms. We extended the vLLM API to associate each inference request with its corresponding QoS requirements (TTFT, TBT, and/or TTLT) and priority level during request submission. The hybrid prioritization policy is implemented using a priority queue that incorporates both deadline proximity and estimated processing time, with the interpolation factor α configurable as a deployment parameter. For non-interactive requests, we maintain a running history of token generation patterns per application to estimate the expected decode length. α is a configurable hyperparameter. For fixed-QPS runs, we perform an offline sweep of α values from 0 to 10 and select the value that minimizes SLO violations. An α of 8 ms/token provided the best trade-off, reducing violations without affecting tail latency. For variable-QPS, we employ load-adaptive tuning. At low loads, we set $\alpha = 1$ ms/token as Figure 14 shows that smaller values achieve comparable violation rates and median latency while limiting tail latency. This optimizes tail latency at low loads while minimizing deadline violations at high loads. To support multi-tenant deployments, we add a priority field to each request that enables relegation decisions based on application hints such as free-tier versus premium users.

3.6.1 Dynamic chunking batch predictor. Given the statistics of the requests in a batch, e.g. number of requests, context lengths, etc., the dynamic chunk size predictor determines the optimal chunk size that maximizes throughput while adhering to the latency constraints. For this, we train a lightweight random forest model which predicts the execution time of a given batch. The model is trained on latency profiles of MLP and attention operation collected at varying chunk sizes, batch sizes as well as context lengths. To collect this data, we use a lightweight harness exposed by an inference simulator Vidur [3], and collect profiles for each model, hardware, and parallelism configuration of interest.

The prediction runs on the CPU and incurs a negligible overhead with < 10% error margin on chunk size prediction. We tune the model to err on the side of under-predicting chunk size. While this may scarcely miss on fully maximizing throughput; it ensures no inadvertent latency increase due to over-prediction. Modifying the chunk size at runtime incurs no additional cost — simply requiring pulling appropriate number of tokens from the pending prefill queue. (§4.1.4) illustrates how dynamic chunking adapts to varying slack under load and dynamically tunes chunk size per iteration.

4 Evaluation

Our evaluation aims to answer the following questions.

1. What is the improvement due to QoServe in the serving capacity while meeting specified QoS SLOs at a cluster scale (§4.1.1), with PD (Prefill-Decode) colocation (§4.1.2), and with PD disaggregation (§4.1.3), and the impact of dynamic chunking (§4.1.4)?
2. What is the impact of QoServe on request latencies and deadline violations under high load conditions (§4.2)?
3. How does QoServe react to transient load spikes (§4.3)?
4. What is the independent impact of the different optimizations and design choices used in QoServe, impact of varying workload compositions and SLOs (§4.4)?
5. How does QoServe empirically and qualitatively compare to other relevant concurrent work (§4.5)?

Model	GPU (TP)	Attention
Llama3-8B	A100 - 80GB (TP1)	GQA
Qwen-7B	A100 - 80GB (TP2)	MHA
Llama3-70B	H100 - 80GB (TP4)	GQA

Table 1. Model configurations and hardware setup

Dataset	Prompt tokens		Decode tokens	
	p50	p90	p50	p90
ShareGPT	1730	5696	415	834
Azure Conv	928	3830	41	342
Azure Code	1930	6251	8	43

Table 2. Datasets used in evaluation

Models and Hardware. We evaluate QoServe across three different models, tensor parallel (TP) degrees, two hardware platforms, and attention mechanisms to demonstrate the diversity and generality of our results across models and hardware configurations, as shown in Table 1. Our evaluation spans three different datasets with varying ratios of prefill to decode tokens, as shown in Table 2.

Workloads and QoS Tiers. For workloads, we use popular open-source datasets such as ShareGPT [19] and coding and conversation production traces from multiple LLM inference services in Azure [15]. Request arrival times are generated using a Poisson distribution, [4, 13], while maintaining the prefill and decode token counts of the respective traces. To emulate different applications, we divide the dataset into three equal parts, and assign each part with a different application type and the corresponding QoS bucket and SLO. We consider three QoS buckets: one interactive and two non-interactive, as shown in Table 3.

We selected the SLO targets to be representative of three key production workloads at a large cloud provider (<retracted>): Q1: O(ms) – interactive responses (e.g. chat applications), Q2: O(minutes) — user-facing but relaxed SLO (e.g. video summaries), and Q3: O(hours) – batch processing (e.g. email insights). For interactive applications, the SLOs track the TTFT and TBT latency metrics, while for non-interactive applications we only track TTLT. In the first set of experiments, we assume an equal mix of requests from these three representative application categories (33% each). Furthermore, to demonstrate resilience to the choice of workload split and SLOs, we also evaluate QoServe with varying workload composition and SLO targets in (§4.4.2).

Baselines. We built QoServe on top of Sarathi-Serve [4], which itself extends vLLM [11]. Our evaluation includes several baseline configurations: (1) **Sarathi-Silo (SOTA)**, the State-of-the-art siloed deployment where each QoS bucket is assigned an independent GPU cluster with each replica running a Sarathi scheduler (2) **Sarathi-FCFS**, which co-schedules requests across all QoS Tiers on a unified cluster

QoS bucket	Request ratio	Interactive		Non-interactive
		TTFT(s)	TBT(ms)	TTLT(s)
Q1	33.33%	6	50	-
Q2	33.33%	-	-	600
Q3	33.33%	-	-	1800

Table 3. QoS classes and workload composition

using Sarathi with FCFS policy, and (3) **Sarathi-EDF**, which again co-schedules but also imparts deadline-awareness during scheduling by using the Earliest Deadline First policy on Sarathi. The strictest QoS bucket with 50ms TBT deadline uses a chunk size of 256, while the other two QoS classes use a large chunk size of 2K to maximize throughput in the siloed baselines. For shared cluster baselines, the chunk size chosen is 256, to meet the TBT targets of the strictest tier.

By default, all experiments are run with PD colocation with chunking enabled, except in (§4.1.3) where the experiments are run with PD disaggregation. We evaluate different scheduling policies within the same serving framework (vLLM) to isolate algorithmic improvements from implementation artifacts. This approach ensures fair comparison by eliminating performance variations due to different system implementations. Since Sarathi demonstrates superior throughput over vanilla vLLM through chunking, we do not present the non-chunked vLLM baseline.

Setup. We first evaluate QoServe under uniform load conditions to identify the impact of our design on goodput (§4.1) as well as on latency and SLO violations (§4.2). Next, we evaluate how QoServe performs under transient spikes in load (§4.3), and finally we perform detailed ablation of our individual techniques (§4.4).

4.1 Capacity and Goodput at Regular Load

4.1.1 Cluster-scale evaluation. We evaluate QoServe over a cluster of 16 A100 GPUs (4 nodes, 4 GPUs per node) with pairwise NVLink and 80GB memory per GPU. Table 4 presents the results for serving the Az-Code trace at 35 QPS across 360K requests (equally split among 3 QoS classes as shown in Table 3) using Llama3-8B. The silo baseline allocates dedicated replicas based on capacity estimation from per-replica throughput for each QoS tier: 7 replicas for Q1 and 3 each for Q2 and Q3, totaling 13 GPUs. In contrast, QoServe meets the latency SLOs at each tier, with no deadline violations using 10 mixed-workload replicas. Both deployments use round-robin load balancing across replicas.

QoServe achieves comparable p99 latencies and no deadline violations with 23% fewer GPUs. To validate QoServe's resource efficiency, we reduce the silo allocation to match QoServe's GPU count (6,2,2 replicas), which causes violations to surge to 60.4%. An alternate 4,3,3 allocation faces approximately 95% Q1 violations with p99 latency of 385.36s.

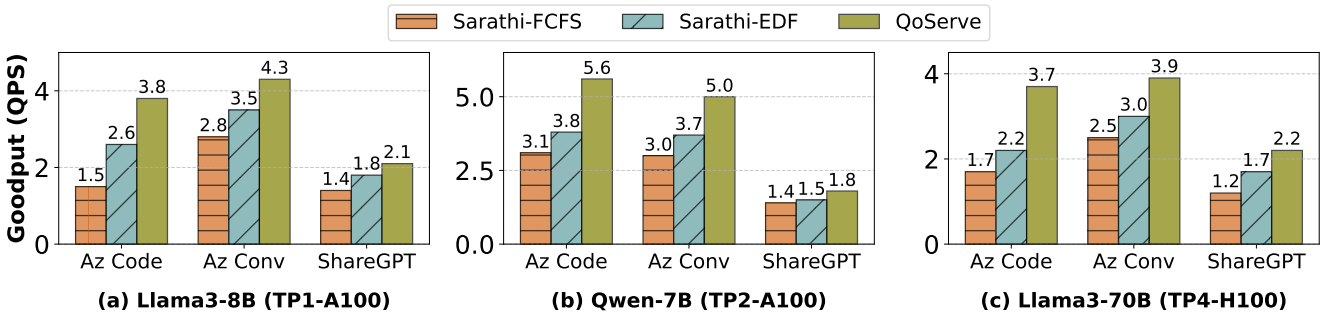


Figure 7. Maximum goodput per replica in a shared cluster across models, hardware, and datasets

Scheme (GPUs)	Total GPUs	p99 Latency in s (SLO) Q1(6s)	Q2(600s)	Q3(1800s)	Overall violations
Silo-(7,3,3)	13	3.09	172.84	171.11	0.24%
Silo-(6,2,2)	10	11.39	4681.56	4678.17	**60.4%**
QoServe-(10)	10	3.38	55.79	204.61	0%

Table 4. Cluster-scale experiments

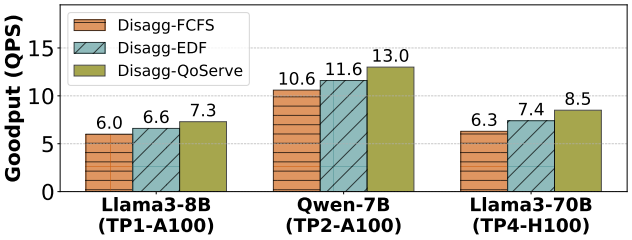


Figure 8. Goodput with PD disaggregation

QoServe achieves better resource efficiency than siloed deployments by maximizing throughput while meeting QoS targets through dynamic chunking. Since the lengths of requests (prompt length as well as number of decode tokens) can vary over time, even at uniform QPS the compute load on the serving system varies. QoServe benefits from dynamically increasing chunk size by exploiting any deadline slack of the non-interactive QoS requests as well as any slack of interactive requests during lower load. On the other hand, the siloed replicas serving the strict QoS requests are limited by the small chunk sizes required to meet the TBT constraints resulting in lower efficiency.

4.1.2 Goodput under PD colocation. We measure the system's goodput, which we define as the number of requests served per replica per second while meeting the latency targets (p99). We allow at most 1% of total requests to violate their deadlines. For these single replica experiments, we compare against the Sarathi-FCFS and Sarathi-EDF baselines. Figure 7 shows the goodput while serving requests over a 4-hour period across three different datasets and models listed in Table 1. As shown, QoServe achieves 1.5x to 2.4x higher

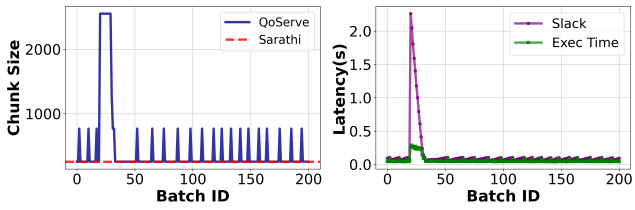


Figure 9. Chunk sizes in QoServe using dynamic chunking

goodput compared to Sarathi-FCFS and 20–40% higher goodput than Sarathi-EDF. These performance benefits stem from a combination of dynamic chunking, hybrid prioritization, and eager relegation. (§4.4) examines the contribution of each of these techniques.

4.1.3 Goodput under PD disaggregation. QoServe's techniques of hybrid prioritization and eager relegation are directly applicable to the prefill nodes of disaggregated serving. We evaluate QoServe on the PD disaggregated mode of vLLM [1] using the Az-conv trace with identical QoS classes from Table 3. We set a large chunk size of 8K as default as we are not constrained by TBT in the prefill nodes for disaggregated serving. As done in the colocated case, we use the Sarathi-FCFS and Sarathi-EDF baselines for these experiments and report the maximum goodput supported per (prefill) replica. In all deployments, the number of decode replicas and their SLO attainment is identical as they work with a maximum batch size that meets the strictest TBT. Efficiently supporting different TBT SLOs in the decode nodes is left to future work. As shown in Figure 8, across models, hardware and parallelism, QoServe achieves better prefill goodput (QPS) compared to the baselines. This directly translates to fewer required prefill nodes in disaggregated serving. The throughput gains are lower compared to PD colocation because we are unable to exploit dynamic chunking here, because of the large baseline chunk size.

4.1.4 Dynamic chunking. Figure 9 shows the effectiveness of dynamic chunking by analyzing chunk sizes and batch latency relative to accumulated slack for Az-conv trace

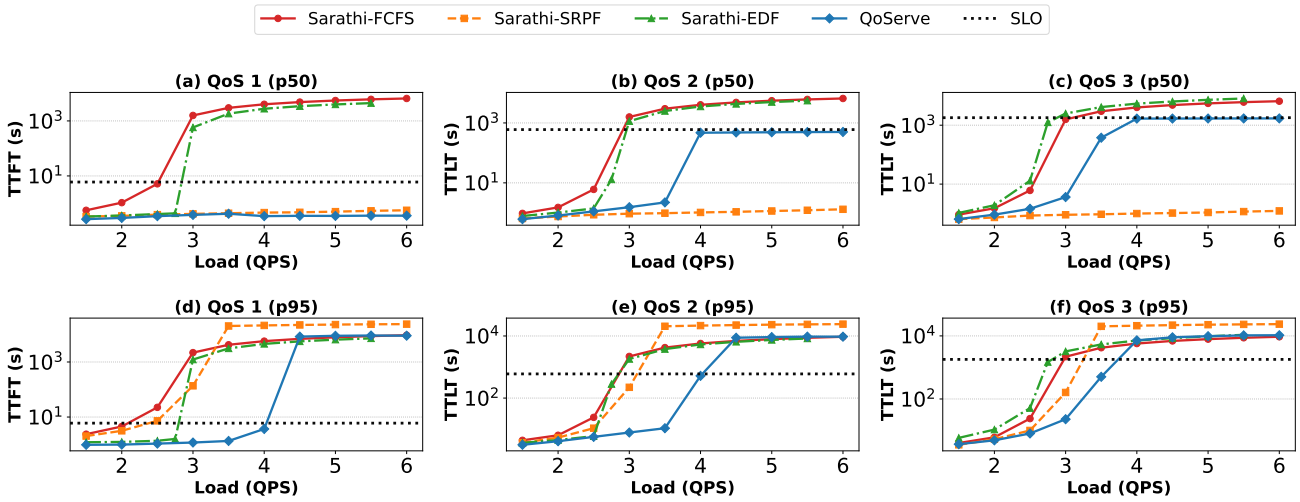


Figure 10. Latency of requests across the three QoS buckets as we vary load in the system

on Llama3-8B for 200 consecutive iterations. As shown, when high slack accumulates across requests, QoServe uses an increased chunk size until slack is exhausted. Dynamic chunking achieves 20% throughput improvement by exploiting latency slack to optimize chunk sizes. We set the chunk size maximizing the throughput using the performance profile in Figure 4. As throughput saturates around 2500, we choose that as the maximum chunk size. Note that 2500 chunk size delivers 2× higher throughput compared to the default 256 chunk size mandated by tight TBT constraints.

4.2 Latency and SLO violations under Overload

We comprehensively evaluate system behavior at various loads by comparing QoServe against baselines on a shared cluster. We measure three key parameters: (1) median and p95 latency (TTFT, TBT, and TTLT) across all requests, (2) percentage of deadline violations across all SLO buckets, and (3) deadline violations in each SLO bucket and violations categorized by request length to assess scheduling fairness. For this evaluation, we add another baseline, Sarathi-SRPF which prioritizes jobs with the lowest pending prefill tokens.

Latency. Figure 10 shows the median and p95 latency across all requests for Llama3-8B on the Azure-Code dataset. As load exceeds the optimal operating point, queuing delay increases because the system cannot process requests as fast as they arrive. This causes a sharp increase in latency for all requests. While this happens in every system, the point where scheduling delay becomes unreasonably large defines the maximum serviceable load. We omit TBT plots since across all schemes, the average TBT violations was less than 0.1%, by virtue of carefully chosen chunk size.
There are several takeaways from these graphs.

- The State-of-the-art Sarathi-FCFS scheduler ignores individual request deadlines. As load increases, SLOs for jobs with stricter QoS requirements are violated. At heavy overloads, head-of-line blocking causes denial of service to all requests.
- Adding deadline awareness through mechanisms like EDF (Sarathi-EDF) better maintains QoS than Sarathi-FCFS, but doesn't scale well with load because we must sacrifice throughput to meet SLOs for the strictest QoS tier. It also degenerates at high loads similar to FCFS due to head-of-line blocking.
- Schedulers that prioritize short jobs like Sarathi-SRPF maintain good median latency at the expense of tail latency. The p95 latency, however, grows unboundedly because SRPF ignores longer requests. Since it is not deadline-aware it prioritizes minimizing latency across requests of all SLO-buckets, which could have otherwise been used to prioritize those with stringent SLOs.
- QoServe handles up to 40% higher load while meeting tail latency SLOs in each QoS bucket compared to baselines. Notably, QoServe's hybrid prioritization smoothly balances between deadline prioritization (EDF) and length prioritization (SRPF), achieving low median latency without drastically increasing tail latency.

Deadline violations. For the same workload, Figure 11(a) shows the overall percentage of SLO violations across all requests as load varies. QoServe maintains zero deadline violations for up to 30% higher load than the next-best scheme, Sarathi-EDF. Even at extreme overloads, QoServe has the fewest deadline violations compared to all other shared-cluster scheduling policies. These lower deadline violation result in the higher goodput we saw in (§4.1.2).

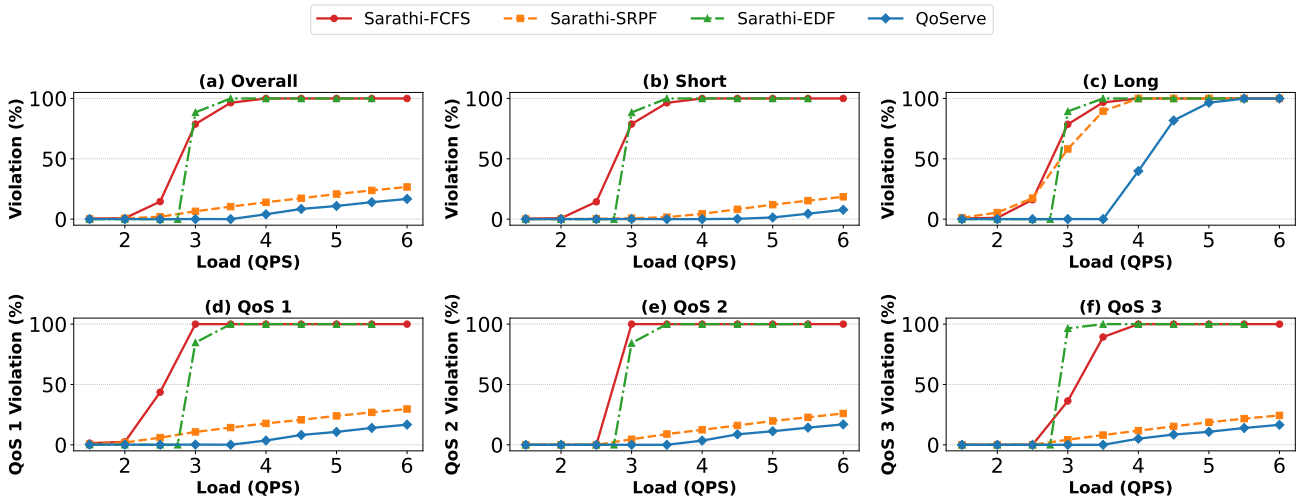


Figure 11. Deadline violations across all jobs, split by request length and QoS buckets

Finally, we analyze whether deadline violations are distributed fairly across request lengths and different QoS buckets. Figure 11(b,c) plot the deadline violations by request length (combined across all QoS buckets). We classify requests as long if their prompt token count is greater than or equal to the 90th percentile, and short otherwise. Our analysis reveals three key patterns:

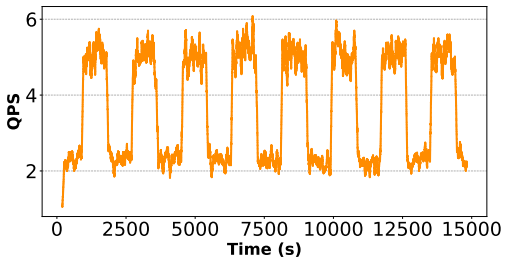
- Sarathi-FCFS and Sarathi-EDF violate SLOs for short and long requests at similar rates. These schedulers do not differentiate between request lengths. At high loads when head-of-line blocking occurs, all requests violate SLOs due to a cascade effect.
- Sarathi-SRPF shows a very high ratio of violations for long versus short jobs, and ignores all long requests beyond certain load. Even at very low loads (<2 QPS), when other schedulers have no deadline violations, Sarathi-SRPF unnecessarily deprioritizes long requests and misses their deadlines. This approach is not only unfair but also counterproductive in real-world settings where request importance doesn't correlate with length.
- QoServe achieves balance between these extremes. It does not deprioritize long requests under normal conditions. During overload, it adjusts the α parameter (§3) to incorporate SRPF-like behavior. This approach allows QoServe to maintain fairness at reasonable loads while gracefully degrading service as load increases.

Figure 11d-f plots the split of deadline violations across the three constituent QoS buckets. We observe that Sarathi-FCFS first violates requests in the strictest QoS bucket and then continues to the less strict buckets. This happens because Sarathi-FCFS is deadline unaware, and due to head-of-line blocking, it violates requests with the shortest deadlines when they get blocked by other requests. Sarathi-EDF equally misses deadlines across all tiers because it treats all requests equally with respect to their individual deadlines. Sarathi-SRPF shows a pattern similar to Sarathi-FCFS, violating the strictest tier first due to being deadline unaware. However, it has fewer overall violations by ignoring long jobs, which frees up capacity for the larger proportion of short requests. On the contrary, QoServe combines the best of these strategies via hybrid prioritization, and achieves fewer overall violations than even Sarathi-SRPF.

4.3 Transient Overload Scenario

We evaluate whether QoServe can gracefully degrade service during transient overload by running an end-to-end evaluation with diurnal load patterns. Load in the system varies dynamically between low (QPS:2.0) and high (QPS:5) points every 15 minutes over a total of 4 hours as shown in Figure 12(a). This workload pattern models realistic diurnal request rate variations typically observed over a weekly cycle in production, compressed into a shorter evaluation timeframe to facilitate the 4-hour experimental duration. This pattern incorporates a 2.5× peak-to-trough ratio consistent with request rate variability documented in LLM production traces [9]. To evaluate QoServe handling of requests with multiple priorities, we mark a random set of 20% of requests in each QoS bucket as low priority, based on application hints. The remaining 80% of requests in each bucket are marked as high priority or Important.

Figure 12(b) shows the overall deadline violations observed in the system. While the baselines collapse under this load and violate deadlines for all requests, QoServe misses deadlines for no important tasks and only 8.75% of all requests. This improvement comes from leveraging application hints to perform eager relegation. Additionally, the throughput gains from dynamic chunking and hybrid prioritization help QoServe sustain higher loads.



Scheme	Violations (%)				
	Overall	Important	QoS 1	QoS 2	QoS 3
Sarathi-FCFS	81.88	81.96	97.13	89.14	59.57
Sarathi-EDF	84.12	84.09	79.3	83.27	89.77
QoServe	8.64	0	16.03	9.98	0

Figure 12. Workload with varying QPS and overall deadline violations across different schemes

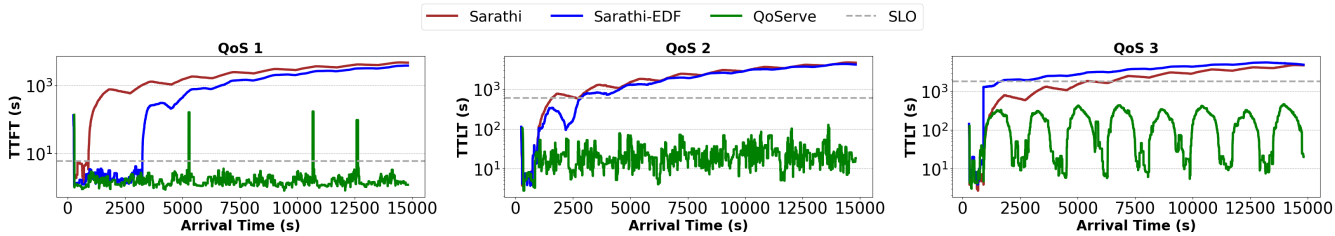


Figure 13. Rolling average of p99 latency of all high-priority requests during a dynamic workload with varying request rates

Figure 13 plots the rolling p99 latency (over 60s windows) of all requests in the system for the three QoS buckets. We see that the baseline Sarathi-FCFS fails to sustain performance during the first request burst. It cannot recover from the queueing delay and enters request denial mode beyond that point for all classes. While Sarathi-EDF sustains the first burst and absorbs some of it until the second peak, it succumbs to queueing delay beyond this point. QoServe handles both high and low load periods, meeting the latency SLOs for a large majority of requests (all important requests and 92% of all requests). For relegated requests in QoServe, the maximum latency observed was no more than 3900s, while the maximum latency in baselines reached 5582s. Across all requests, irrespective of whether they are relegated, QoServe has better tail latency compared to the baselines. These results demonstrate graceful service degradation—proactively dropping a few requests during overload to maintain service levels for the majority, thereby eliminating cascading effects. In fact, the p50 rolling average for QoServe remains much more uniform and resilient to load changes.

4.4 Ablation Studies

4.4.1 Impact of various techniques. We now examine how each component of our system design affects throughput and SLO violations. For this analysis, we tag all requests as important and evaluate three design elements—dynamic chunking, hybrid prioritization, and eager relegation—starting with the Sarathi-EDF baseline. Table 5 shows that dynamic chunking provides a 20% boost in throughput, while eager relegation adds another 9%. The impact of hybrid prioritization appears marginal in the optimal load scenario but becomes significant at high load.

Config	Optimal Load		High load (QPS=6)	
	QPS	% gain	% viol	% impr.
Sarathi-EDF	2.75	-	100	-
QoServe (DC)	3.3	**20%**	74	**26%**
QoServe (DC+ER)	3.6	**9%**	26	**68%**
QoServe (DC+ER+HP)	3.65	**1.4%**	16	**32%**

Table 5. Impact of QoServe's optimizations. (DC:Dynamic Chunking, ER:Eager Relegation, HP:Hybrid Prioritization)

To further illustrate the impact of hybrid prioritization, Figure 14 plots the median latency and percentage of deadline violations as we vary system load across three different values of α, our hybrid prioritization parameter. As α increases, the system increasingly deprioritizes longer requests. This significantly reduces median latency for all requests but comes at the cost of violating deadlines for most long requests. This demonstrates the importance of tuning this parameter as load increases to strike a balance between low median latency and fair service for long requests.

4.4.2 Varying workload composition and SLOs. We now evaluate QoServe's robustness across diverse SLO configurations and workload compositions.

Workload mix. We evaluate QoServe under skewed workload distributions: 70-15-15 (interactive dominant) and 15-15-70 (batch-dominant) at 4.5 QPS. Table 6 presents p99 latencies and SLO violations. While baseline systems fail to meet

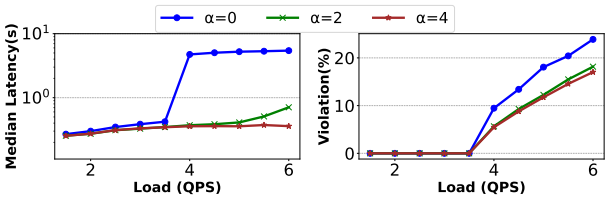
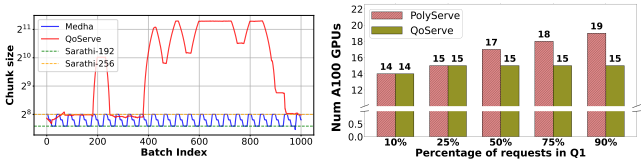


Figure 14. Varying the hybrid prioritization parameter



(a) Comparison to Medha **(b)** Comparison to PolyServe

Figure 15. Comparison with concurrent work

Scheme	Median latency (in s) Q1(6)	Q2(600)	Q3(1800)	% Violations
Composition: 70-15-15				
Sarathi-FCFS	4835.2	4843.6	4825.7	100%
Sarathi-EDF	3354.3	4214.7	6033.6	98%
QoServe	0.77	425.31	1625.15	5%
Composition: 15-15-70				
Sarathi-FCFS	4835.2	4843.6	4825.7	100%
Sarathi-EDF	1800.75	2621.5	4436.4	82.78%
QoServe	0.779	4.7	1027.5	0.5%

Table 6. Latency across workload compositions.

SLO targets under these loads, QoServe maintains SLO compliance across all tiers through a combination of strategic relegation (0.5-5% of requests), deadline-aware scheduling, and increased throughput via dynamic chunking.

Varying SLO. We modify the SLO targets to (3s, 50ms), (6s, 50ms), and (1000s) for Q1, Q2, and Q3 respectively, with equal request distribution. This configuration increases the proportion of interactive workloads compared to previous experiments. On the Azure-Conv trace with Llama3-8B, QoServe achieves 5 QPS goodput while Sarathi-EDF sustains only 3.7 QPS — a 26% performance degradation.

4.5 Comparison to concurrent work

Although the concurrent work discussed in Section 5 are not open-source, we provide best-effort comparisons with three representative approaches.

4.5.1 Medha - Adaptive chunking. Medha [6] uses adaptive chunking that starts with large chunks and progressively shrinks to maintain consistent TBT as attention overhead increases in later chunked iterations. We implement the dynamic chunking policy from Medha within our framework. Figure 15a compares chunk size choices between QoServe and Medha across 1000 consecutive batches using a synthetic trace (10K prefill tokens, 500 decode tokens per request) on Llama3-8B since chunking overhead is negligible for the median prompt lengths (<5K tokens) in our evaluation datasets.

While Medha progressively reduces chunk sizes within a prefill, it is unaware of slack accumulated by the current batch of requests. In contrast, QoServe opportunistically increases chunk sizes when slack becomes available, as demonstrated in Figure 15a. For fairness, we evaluate QoServe with only dynamic chunking under FCFS scheduling, disabling all other techniques. Compared to Medha's adaptive chunking (also under FCFS), this isolated setup yields a 23% goodput improvement (0.32 vs. 0.26 QPS), showing the gains arise solely from the chunking strategy.

4.5.2 PolyServe. PolyServe [22] is a multi-SLO scheduling system designed to serve interactive workloads with diverse TBT requirements. It partitions requests into separate deployments based on TBT SLO categories, employing dedicated resources and autoscaling for each deployment. It uses a similar adaptive chunking policy as Medha. We do not implement autoscaling and request migration in PolyServe or QoServe to compare only the core techniques.

We compare PolyServe against QoServe's collocated approach using PolyServe's evaluation methodology. Our experimental setup comprises two interactive job categories: Q1: 50ms TBT and Q2: 100ms TBT, both maintaining 6s TTFT SLOs. We determine the A100 GPU requirements for serving Llama3-8B on Azure Conversation traces at 50 QPS total load, varying the distribution of requests between QoS classes. We calculate GPU requirements by determining maximum per-replica goodput for each QoS class, then computing total resources needed for specific load configurations.

Figure 15b presents the comparative capacity requirements as we vary request composition across the two TBT classes for PolyServe and QoServe. Compared to PolyServe, QoServe always has lower resource requirement due to colocation of requests which allows exploiting prefill slack and improving throughput using dynamic chunking.

4.5.3 SLOs-Serve. We provide a qualitative comparison with SLOs-Serve [8], which employs periodic dynamic programming to optimize scheduling across all active and queued requests. Despite sharing similar multi-QoS objectives with QoServe, SLOs-Serve's $O(NN_{new}M)$ scheduling complexity (where N represents running requests, N_{new} denotes queued requests, and M indicates KV blocks) exhibits poor scalability. In comparison, QoServe requires $O(\log(N_{new}))$ time to choose the prefill tokens to schedule from the priority queue. In their paper, SLOs-Serve is evaluated with MHA models (which limits the KV-cache size) with constrained batch sizes on 40GB GPUs, where the scheduling overheads would be lower; whereas QoServe efficiently scales to larger model configurations and distributed deployments.

5 Related work

Model-level optimization. Approaches like AlpaServe [12] focus on optimal model parallelism strategies to meet varying SLO requirements across different models. Learned Best-Effort LLM Serving [10] employs deep reinforcement learning to route requests across multiple models for accuracy-latency optimization. Unlike these approaches that optimize across models, deployments, or sacrifice accuracy, QoServe focuses on exploiting temporal slack within a single model to maximize throughput without compromising quality.

Adaptive chunking. Systems such as Medha [6] and PolyServe [22] introduce a variant of dynamic chunking that progressively reduces chunk size to maintain consistent TBT as attention overhead increases in the later chunked iterations of Sarathi [4]. Unlike QoServe, they assume fixed TBT targets and don't exploit SLO slack across requests.

Workload collocation. ConServe [14] advocates collocated serving by prioritizing interactive jobs and adding offline tasks when latency permits, using reactive preemption during load surges. However, its binary interactive-offline classification is inadequate for multi-QoS scenarios where all requests have definite SLO requirements. SageServe [9] proposes a simple reactive heuristic, adding batch jobs when interactive replica load drops below 60%. Without hybrid prioritization and slack awareness, SageServe's SLO-agnostic approach would exhibit high SLO violations similar to naive collocation. Unlike these, QoServe employs fine-grained proactive scheduling across multiple QoS classes.

Multi-SLO scheduling. SLOs-Serve [8], uses periodic dynamic programming over all running and queued requests. While conceptually similar to QoServe's multi-QoS goals, SLOs-Serve's scheduling complexity makes it significantly less scalable than QoServe for large batch sizes and multi-replica deployments. PolyServe [22] advocates binning requests with different TBT constraints into independent deployments with appropriate autoscaling. Unlike QoServe's unified serving approach, PolyServe's deployment isolation prevents cross-QoS slack exploitation and requires separate resource provisioning for each SLO class.Tempo [20] adopts an SLO-aware approach that leverages conservative output length prediction and online refinement to maximize service gain, in contrast to QoServe's slack-aware dynamic chunking and hybrid prioritization for efficient QoS co-scheduling.

6 Conclusion

We address the challenge of co-scheduling multiple QoS classes in LLM inference serving, and graceful service degradation during overload. We achieve this using three key techniques: (1) dynamic chunking to opportunistically maximize throughput while meeting latency targets, (2) hybrid prioritization to strike a balance between maintaining low median latency and fairness in serving longer requests, and (3) eager relegation to enable graceful service degradation. Our evaluation shows that QoServe significantly improves QoS attainment compared to State-of-the-art LLM serving systems, particularly under high load. As LLMs power more applications with varying performance needs, we believe that techniques supporting multiple QoS classes will become essential for production deployments.

7 Acknowledgments

We thank our shepherd David Lo, along with the anonymous ASPLOS reviewers, for their valuable feedback.

References

[1] [n. d.]. vLLM: Easy, fast, and cheap LLM serving for everyone. https://github.com/vllm-project/vllm.

[2] Amey Agrawal, Anmol Agarwal, Nitin Kedia, Jayashree Mohan, Souvik Kundu, Nipun Kwatra, Ramachandran Ramjee, and Alexey Tumanov. 2024. Etalon: Holistic Performance Evaluation Framework for LLM Inference Systems. *arXiv e-prints*, Article arXiv:2407.07000 (July 2024), arXiv:2407.07000 pages. https://doi.org/10.48550/arXiv.2407.07000 arXiv:2407.07000 [cs.LG]

[3] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A Large-Scale Simulation Framework For LLM Inference. *Proceedings of The Seventh Annual Conference on Machine Learning and Systems, 2024, Santa Clara* (2024).

[4] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[5] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 [cs.LG]

[6] Amey Agrawal, Haoran Qiu, Junda Chen, Íñigo Goiri, Chaojie Zhang, Rayyan Shahid, Ramachandran Ramjee, Alexey Tumanov, and Esha Choukse. 2025. Medha: Efficiently Serving Multi-Million Context Length LLM Inference Requests Without Approximations. arXiv:2409.17264 [cs.LG] https://arxiv.org/abs/2409.17264

[7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[8] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. 2025. SLOs-Serve: Optimized Serving of Multi-SLO LLMs. arXiv:2504.08784 [cs.DC] https://arxiv.org/abs/2504.08784

[9] Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St. Amant, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. 2025. SageServe: Optimizing LLM Serving on Cloud Data Centers with Forecast Aware Auto-Scaling. arXiv:2502.14617 [cs.DC] https://arxiv.org/abs/2502.14617

[10] Siddharth Jha, Coleman Hooper, Xiaoxuan Liu, Sehoon Kim, and Kurt Keutzer. 2024. Learned Best-Effort LLM Serving.

arXiv:2401.07886 [cs.LG] https://arxiv.org/abs/2401.07886

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[12] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. arXiv:2302.11665 [cs.LG] https://arxiv.org/abs/2302.11665

[13] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. arXiv:2405.04437 [cs.LG] https://arxiv.org/abs/2405.04437

[14] Yifan Qiao, Shu Anzai, Shan Yu, Haoran Ma, Yang Wang, Miryung Kim, and Harry Xu. 2024. ConServe: Harvesting GPUs for Low-Latency and High-Throughput Large Language Model Serving. arXiv:2410.01228 [cs.DC] https://arxiv.org/abs/2410.01228

[15] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. arXiv:2408.00741 [cs.AI] https://arxiv.org/abs/2408.00741

[16] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[18] vLLM Project. 2024. [RFC] Upstream Chunked Prefill #3130. https://github.com/vllm-project/vllm/issues/3130.

[19] Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2024. OpenChat: Advancing Open-source Language Models with Mixed-Quality Data. arXiv:2309.11235 [cs.CL] https://arxiv.org/abs/2309.11235

[20] Wei Zhang, Zhiyu Wu, Yi Mu, Banruo Liu, Myungjin Lee, and Fan Lai. 2025. Tempo: Application-aware LLM Serving with Mixed SLO Requirements. arXiv:2504.20068 [cs.DC] https://arxiv.org/abs/2504.20068

[21] 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. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

[22] Kan Zhu, Haiyang Shi, Le Xu, Jiaxin Shan, Arvind Krishnamurthy, Baris Kasikci, and Liguang Xie. 2025. PolyServe: Efficient Multi-SLO Serving at Scale. arXiv:2507.17769 [cs.DC] https://arxiv.org/abs/2507.17769

A Artifact Appendix

A.1 Abstract

QoServe is a QoS-driven LLM inference serving framework that enables efficient co-scheduling of requests across multiple QoS classes on shared infrastructure. This artifact contains the source code, datasets, and scripts to reproduce key results from our paper (Figures 7, 8, 10, 11). QoServe is built on Sarathi-Serve [4], extending vLLM [1] with deadline-aware scheduling capabilities.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Hybrid prioritization, dynamic chunking, eager relegation for multi-SLO LLM serving
- **Model:** Llama3-8B [16], Llama3-70B [16], Qwen-7B [7]
- **Data set:** ShareGPT [19], Azure Conversation traces [15], Azure Code traces [15]
- **Run-time environment:** Ubuntu 20.04+, CUDA 12.1, Python 3.10, PyTorch 2.3.0
- **Hardware:** 4×A100 80GB GPUs (preferred for Llama3-8B and Qwen-7B experiments). Minimum: 2×A100 with pairwise NVLink for Qwen-7B (TP2). For Llama3-70B: 4×H100 80GB GPUs with NVLink (TP4). Tiny scripts available for single A100 GPU.
- **Metrics:** TTFT, TBT, TTLT, goodput (QPS), deadline violations (%)
- **Experiments:** Maximum goodput per replica, latency and deadline violations under varying load
- **How much disk space required?:** ~150 GB
- **How much time to prepare workflow?:** 20–60 minutes
- **How much time to complete experiments?:** ~61 hours on 4×A100 (Figure 7: 13h, Figure 8: 8h, Figures 10–11: 40h)
- **Publicly available?:** Yes, on GitHub and Zenodo.
- **Archived (DOI)?:** Yes, DOI: 10.5281/zenodo.18218177

A.3 Description

A.3.1 How to access. The artifact is publicly available on GitHub and archived on Zenodo with DOI: 10.5281/zenodo.18218177.

A.3.2 Hardware dependencies. The evaluation scripts assume a 4×A100 80GB node with NVLink. Qwen-7B requires minimum 2×A100 with pairwise NVLink (TP2). Llama3-70B requires 4×H100 80GB GPUs with NVLink (TP4). For resource-constrained environments, tiny scripts are provided that run on a single A100 GPU with reduced execution times while preserving core trends.

A.3.3 Software dependencies. Python 3.10, PyTorch 2.3.0, CUDA 12.1, FlashInfer 0.1.1. See README for complete dependency list and installation instructions.

A.3.4 Data sets. ShareGPT [19] and Azure production traces (Code, Conversation) [15]. Scripts automatically download datasets from public Azure blob storage.

A.3.5 Models. Llama3-8B, Qwen-7B, Llama3-70B accessed via Hugging Face (requires HF token with model access).

A.4 Installation

Refer to README for detailed installation instructions. Summary: create conda environment, install dependencies via pip, download datasets, export `HF_TOKEN`, and configure GPU clocks for reproducibility.

A.5 Experiment workflow

The artifact provides automated bash scripts for each figure:

- `tester.sh`: Quick validation run (~5 minutes)
- `fig7.sh`: Goodput with PD colocation (13 hours, Llama3-8B TP1 only)
- `fig7_tiny.sh`: Goodput with PD colocation (8 hours, single A100)
- `fig8.sh`: Goodput with PD disaggregation (8 hours, Llama3-8B TP1 only)
- `fig10_11.sh`: Latency and violations under load (40 hours)
- `fig10_11_tiny.sh`: Latency and violations under load (11 hours, single A100)

Results are saved to `benchmark_output/` (raw logs) and `paper_plots/` (plots and graphs).

A.6 Evaluation and expected results

Figure 7 (Goodput, PD colocation): QoServe achieves 1.5–2.4× higher goodput than Sarathi-FCFS and 20–40% over Sarathi-EDF. Current artifact reproduces Llama3-8B TP1 on Azure Code Trace.

Figure 8 (Goodput, PD disaggregation): QoServe shows consistent goodput improvements across disaggregated serving. Current artifact reproduces Llama3-8B TP1 on Azure Conv. Trace.

Figures 10–11 (Latency and violations): QoServe is capable of handling significantly higher load while meeting tail latency SLOs and reduces deadline violations by an order of magnitude under overload. Note: artifact uses fewer requests and coarser QPS sweep than paper for manageable runtime; trends and relative performance remain consistent. The sweep is done on Llama3-8B TP1 for the Azure Code Trace.

A.7 Experiment customization

Key parameters can be modified in source files:

- QoS tiers: `sequence.py` (line 11)
- Hybrid prioritization α: `sequence.py` (line 79)
- Scheduling logic: `deadline_scheduler.py`

Refer to README for implementation details and customization guide.

A.8 Notes

GPU clock locking (described in README) is essential for reproducible measurements. The current artifact uses a smaller number of requests for manageable runtime; the original paper evaluated on significantly larger request volumes for stronger statistical significance. This may lead to variations in reproducing exact numerical results, though relative performance trends and conclusions remain consistent.