SLO-AWARE OPTIMIZATION AND STATEFUL ORCHESTRATION FOR LLM SYSTEMS

BY

ZHIYU WU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2025

Urbana, Illinois

Adviser:

Assistant Professor Fan Lai

ABSTRACT

The rapid evolution of Large Language Models (LLMs) has shifted the focus of AI infrastructure from simple text generation to complex, multi-turn agentic workflows. As these applications become increasingly sensitive to latency and dependencies, existing serving systems—which primarily optimize for aggregate throughput—fail to meet application-specific Service Level Objectives (SLOs). Furthermore, as workloads evolve into multi-agent systems (MAS), the lack of robust state management and error recovery in current runtimes creates a bottleneck for reliable orchestration.

This thesis addresses these challenges by proposing a comprehensive optimization of the LLM runtime stack. First, we present Concord, an SLO-aware serving system designed to maximize service "goodput" (the rate of requests served within strict performance goals) under imprecise request information. Concord employs a novel iterative scheduling algorithm and Criticality-Aware Length Matching ($CALM$) to dynamically refine resource allocation as generation progresses. Evaluation across diverse realistic workloads, including chat, deep research, and agentic pipelines, demonstrates that Concord improves service goodput by 1.4×–6.3× and achieves 28.5%–83.2% resource savings compared to state-of-the-art designs.

Building upon this optimized serving layer, the thesis concludes by exploring the future of Stateful Agent Orchestration. We propose the design of an ML Agent Compiler, a runtime environment akin to a JVM for agents. This proposed framework addresses the limitations of current stateless orchestration by introducing graph-based checkpointing, forking engines, and deduplication of partial executions. Together, these works chart a path toward a unified, efficient, and fault-tolerant infrastructure for the next generation of AI applications.

To my family.
To my fiancée.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Professor Fan Lai, for his continuous support throughout my master's studies and research. His patience, enthusiasm, and immense knowledge have been invaluable. I have had the privilege of working with Fan for more than two years, during which he consistently provided hands-on research guidance and offered thoughtful advice on my career development. His dedication, passion, vision, and honest feedback have profoundly shaped my research mindset—teaching me to pursue systems that are clean and simple in design, yet novel, solid, and impactful in solutions. Fan has been far more than a mentor to me; he has been like a senior friend or older brother in academia, guiding and supporting me as I develop my own path. My master's studies would not have progressed so smoothly without his invaluable mentorship.

I also thank my wonderful lab mates: Wei Zhang, Yi Mu, Rui Ning, Banruo Liu, Jiahao Fang, Sean Nian, Qilong Feng, Jeff B, and Anagha Balaji. It has been a pleasure sharing unforgettable moments, especially during submission deadlines. Special thanks to Wei Zhang; without his help, this thesis would not have been possible. I feel truly fortunate to have had the opportunity to work with him.

I am grateful to my undergraduate mentors at the University of Michigan, including Professor Mosharaf Chowdhury, Dr. Jiachen Liu, and Jae-Won Chung. Their guidance laid a strong foundation for my academic growth. I also thank my industry collaborators, Dr. Myungjin Lee and Nikhil Sarda, for their rigorous approach and insightful feedback, and my internship mentors, Xiao Yu and Yifan Pi, for their support during the summer.

I extend my thanks to all my friends at UIUC, especially Jiahao Fang, for their companionship and support. Life at UIUC would not have been nearly as colorful or enjoyable without you. I am also thankful to the faculty and staff of the UIUC Department of Computer Science for their support.

Finally, I am deeply grateful to my family for their unconditional love. To my parents, thank you for always believing in me, supporting me emotionally and financially, and encouraging me to pursue my passions. Your dedication has given me the confidence to chase my goals. To my fiancée, Yuchen Wang—sharing so many important moments and milestones with you has been one of my greatest joys. Thank you for your patience, understanding, and love, especially during stressful periods of my studies. Your support has been my greatest strength and has made every challenge more manageable. I am truly grateful to walk this journey with you and look forward to our future together.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

As large language models (LLMs) enable language-driven interaction between humans and intelligent agents, modern applications are increasingly transcending conventional chatbot scenarios like ChatGPT. Today, AI infrastructure must support complex, multi-turn workflows where LLMs integrate with external tools (e.g., AI-assisted coding platforms [1] and autonomous web agents [2]). This shift makes it crucial to ensure responsiveness not just for human readers, but for system reliability—avoiding external timeouts [3] and preventing degraded user experience [4].

Increasingly, these applications issue dependent LLM requests to enhance problem-solving, such as response aggregation in test-time scaling [5] or coordination in multi-agent systems (MAS) [6]. This introduces two fundamental challenges to the runtime stack: massive generation token volumes[1] and complex, evolving execution dependencies.

The expanding landscape of LLM applications has introduced increasingly diverse Service-Level Objectives (SLOs) for responsiveness. Our analysis of millions of LLM requests across real-world applications—corroborated by user studies and discussions with service providers—reveals that requests fall into three dominant patterns: (i) Latency-sensitive requests: Prioritize per-token latency metrics such as Time-To-First-Token (TTFT) and Time-Between-Tokens (TBT), typical in interactive chatbots where streaming speed dictates user satisfaction [4, 7]. (ii) Deadline-sensitive requests: Require low End-to-End Latency (E2EL) to return a complete response quickly. These are common in cloud AIOps [8], batch-processing APIs [9], or agent interactions where slow generation triggers tool timeouts [10]. (iii) Compound requests: Consist of multiple dependent LLM calls to complete a single task, such as hierarchical reasoning [11] or deep research planning.

Current serving systems are ill-equipped to handle this heterogeneity. Existing schedulers largely optimize for aggregate throughput (tokens per second) or average latency [12]. We prove that these objective functions can yield arbitrarily poor application-level performance. For example, a scheduler might delay a deadline-sensitive request to batched efficiently with a long document summary. While this maximizes the GPU's arithmetic intensity, it causes the deadline-sensitive request to fail its SLO. In an agentic world, "bandwidth" consumed by a request that misses its deadline is bandwidth wasted.

To align infrastructure with application needs, this thesis proposes that the system must optimize for Service Goodput—defined as the total volume of requests that strictly meet their individual SLOs.

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

We introduce Concord, an SLO-aware serving system built on a novel lazy commitment scheduling principle. The central insight of Concord is that perfect scheduling information (e.g., exactly how many tokens a model will generate) is unavailable at request arrival. Instead of relying on static assumptions, Concord leverages imprecise information—such as probabilistic upper bounds on response length and estimated dependency graph structures—to make conservative initial allocations. As generation progresses, Concord engages in Progressive Refinement. It continuously updates its estimates based on the tokens actually generated. This allows the scheduler to dynamically "harvest" slack: if a request is generating faster or shorter than predicted, its reserved GPU time is immediately freed and reallocated to other pending requests.

To realize this, we developed the Criticality-Aware Length Matching (*CALM*) algorithm. This algorithm solves the dual challenge of scheduling individual requests (an NP-hard packing problem) and composing efficient batches for the GPU. *CALM* prioritizes requests based on their "criticality"—the buffer they have before violating their SLO. It ensures that every request receives the minimum sufficient bandwidth to survive the current time step, while strategically batching requests of similar lengths to minimize the "padding waste" inherent in transformer attention mechanisms.

We implemented Concord atop vLLM [13], ensuring it remains compatible with standard APIs. Extensive evaluation on workloads ranging from chatbots to deep research agents demonstrates that Concord improves service goodput by 1.4×–6.3× and achieves 28.5%–83.2% resource savings compared to state-of-the-art schedulers.

We implement Concord atop vLLM [13], preserving API compatibility to support a wide range of applications (§5). We evaluate Concord on diverse LLMs and real-world workloads, including chatbots [14], deep research [15], and agentic workflows [16]. Our results show that Concord improves service goodput by 1.4×–6.3×, achieving 28.5%–83.2% resource savings for equivalent goodput, while achieving near-oracle performance (§6).

While Concord solves the problem of serving tokens efficiently, the evolution toward autonomous agents exposes a new bottleneck at the orchestration layer. As agents move from passive analysis to active execution (e.g., SWE-Agent [17] or App-World [18]), they become stateful. However, "state" in this advanced context transcends the mere conversation history or Key-Value (KV) cache residing in GPU memory. It encompasses the System and Environment State: modifications to the local file system, installed Python packages, active shell processes, and environment variables. Current agent runtimes (like LangChain or AutoGen) are effectively "stateless" regarding the environment; they cannot easily rollback a file write or clone a running shell process if an agent creates a new branch of reasoning.

This thesis concludes by proposing the ML Agent Compiler as a future direction. En-

visioned as a runtime environment analogous to a JVM for Agents, this system introduces a "Fork Engine" capable of managing these environmental side-effects. By implementing Copy-on-Write (CoW) mechanisms for file systems and snapshotting for shell environments, the ML Agent Compiler aims to allow agents to fork, backtrack, and merge execution paths without corrupting the underlying environment. This would allow the infrastructure to treat complex, stateful agent workflows with the same elasticity and fault tolerance that Concord provides for token generation.

In summary, our contributions are:

1. We perform real-world studies of LLM services, including user surveys and discussions with providers, introducing a new characterization of request patterns (§2);

2. We design a novel scheduler and the *CALM* algorithm that estimate, refine, and exploit imprecise request information to maximize goodput with provable performance (§3-§4);

3. We demonstrate Concord's significant improvements in application-level performance across real workloads (§6).

4. Future of Agent Runtimes: We propose the architecture for an ML Agent Compiler, detailing how graph-based dependency management and state forking can eliminate redundancy and ensure fault tolerance in next-generation multi-agent systems (§8).

CHAPTER 2: BACKGROUND AND 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 [19], WildChat [20]), agentic AI systems (MetaGPT [21], GAIA [6]), and reasoning tasks (deep research [15, 22], math reasoning [11]). 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 2.1).

Latency-sensitive vs. Deadline-sensitive vs. Compound Requests. Our studies show that LLM requests can be increasingly categorized into three key patterns (Figure 2.1):

1. *Type 1: Latency-sensitive Requests.* They generate responses consumed in a streaming fashion. Representative applications include ChatGPT-style web services, AI-powered customer support [23], and real-time speech-to-text services [24]. 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 [4, 25].

2. *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 [8, 10]), data cleaning [26], large codebase generation [27], and batch processing APIs at OpenAI and Gemini [9].

3. *Type 3: Compound Requests.* These involve multiple dependent LLM calls, often forming graph-structured execution dependencies [28, 29, 30], with the requirement that

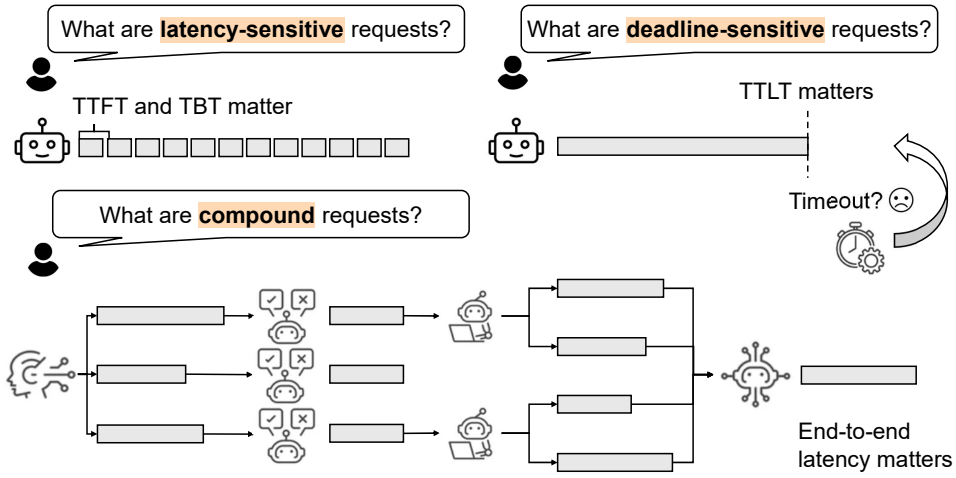


Figure 2.1: Illustration of three request patterns.

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 2.1: 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.

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 [11], multi-agent workflows [30], hierarchical planning scenarios such as deep research [31, 32], and reinforcement learning from human feedback pipelines where multiple responses must be generated per prompt [33].

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 [25, 34] and our discussions with service providers. Further, certain requests may not impose explicit SLOs (e.g., synthetic data generation [35]), yet do not want to suffer starvation.

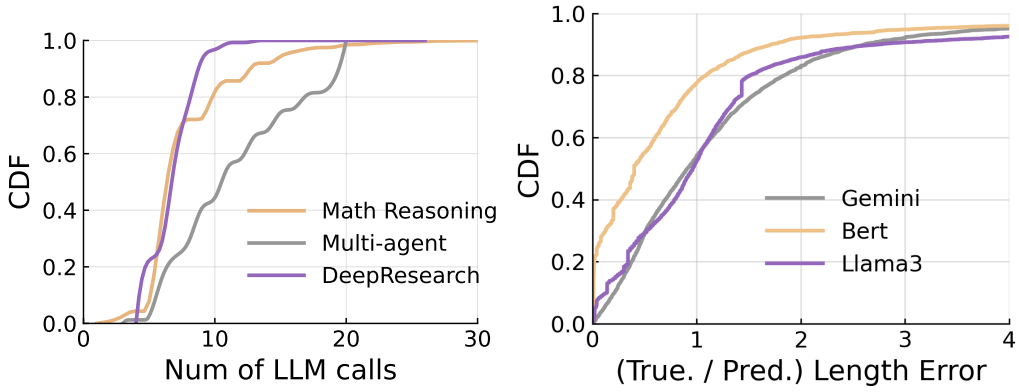
Beyond workload and application heterogeneity, SLO requirements also vary across users. As shown in Table 2.1, 30.5% of users prefer minimizing E2EL in code generation for rapid testing and direct execution, whereas 38.1% favor streaming code delivery to facilitate interactive reading and comprehension. Even within streaming use cases, users exhibit different reading speeds, translating into heterogeneous TBT requirements [4, 25, 36]. Batch APIs provide differentiated response guarantees across pricing tiers, leading to distinct deadline constraints [37, 38]. Agentic workflows also diverge in their downstream integrations (e.g., triggering automated remediation or updating monitoring dashboards), resulting in varied E2EL requirements [39].

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 [40, 41], computing [42], and storage [43, 44], 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 [45]. Second, modern LLM workloads often exhibit complex and unpredictable execution dependencies. As shown in Figure 2.2(a), applications such as multi-agent workflows [16], test-time scaling for reasoning [11], 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 [46]).



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

Figure 2.2: (a) LLM workloads increasingly involve varying numbers of subrequests (LLM calls) in a request. (b) Predicting response length remains highly inaccurate.

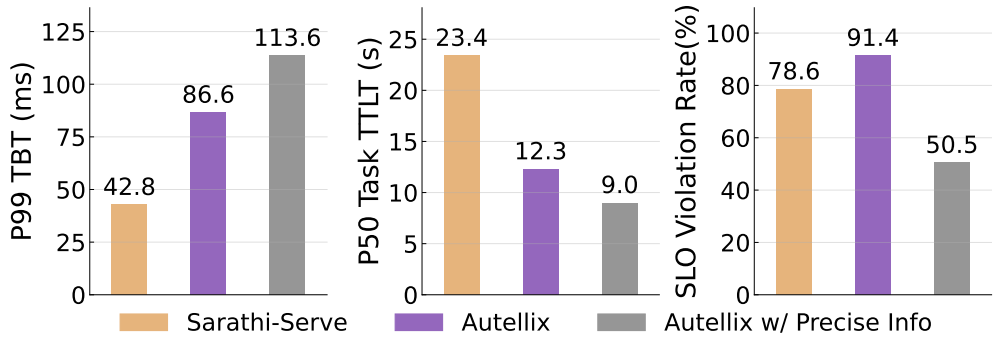


Figure 2.3: Existing advances face significant performance drops due to growing LLM request diversity.

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 [47]. As shown in Figure 2.2(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 [48]) 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 [49] or Least-Attained Service (LAS) First as in Autellix [30]. However, we theoretically prove that they can lead to arbitrarily poor goodput

(Appendix §D.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 2.3 shows that while Autellix improves average E2EL compared to Sarathi-Serve [12], 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 2.3, 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 2.3 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:

1. *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.

2. *Goodput Efficiency*: Maximize service goodput and resource utilization for providers, while remaining robust to runtime dynamics with provable performance guarantees.

3. *Deployability and Scalability*: Integrate seamlessly with existing serving stacks and provide strong scaling capabilities (e.g., extending to multiple concurrent models).

CHAPTER 3: CONCORD OVERVIEW

We introduce Concord, 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. Concord 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. Concord 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 [50]:

1. *Latency-sensitive requests*: measured as the number of tokens delivered within the expected *timeline* [4, 25]. Specifically, token i counts toward goodput if it finishes by $TTFT_{SLO} + i \times TBT_{SLO}$.

2. *Deadline-sensitive requests*: measured as the total number of tokens, including input and output, if the request completes before its deadline; zero otherwise [4].

3. *Compound requests*: measured as the total number of tokens across all subrequests if the final generation completes by the E2EL deadline; zero otherwise [30].

Concord 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 Concord consistently improves various goodput objectives (§6.2), such as request-level goodput like maximizing the number of requests that meet their SLOs [51].

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

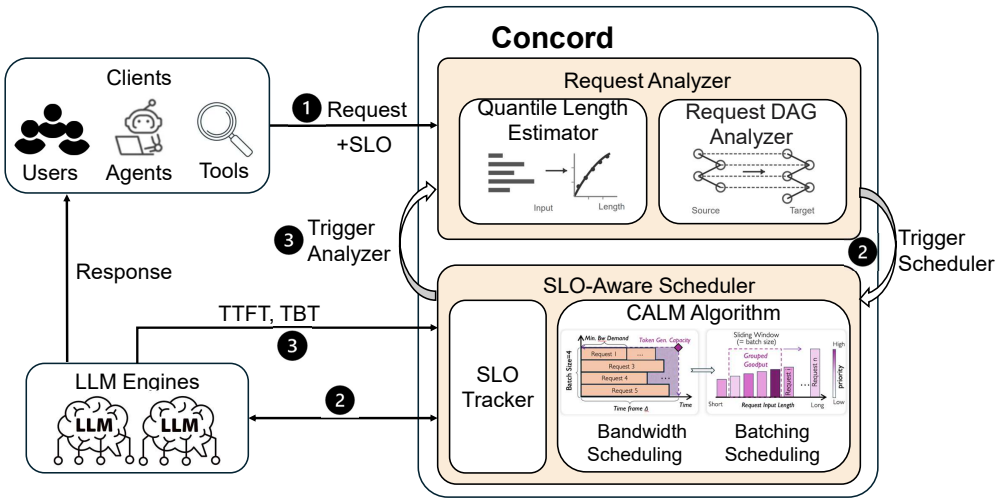


Figure 3.1: Concord system overview and workflow.

System Workflow. As illustrated in Figure 3.1, Concord operates as a middleware layer that aligns application-level performance needs with underlying execution backends (e.g., vLLM [13]). ❶ Upon request arrival, along with its SLO (specified by users [36] or developers [7]), 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.

CHAPTER 4: CONCORD DESIGN

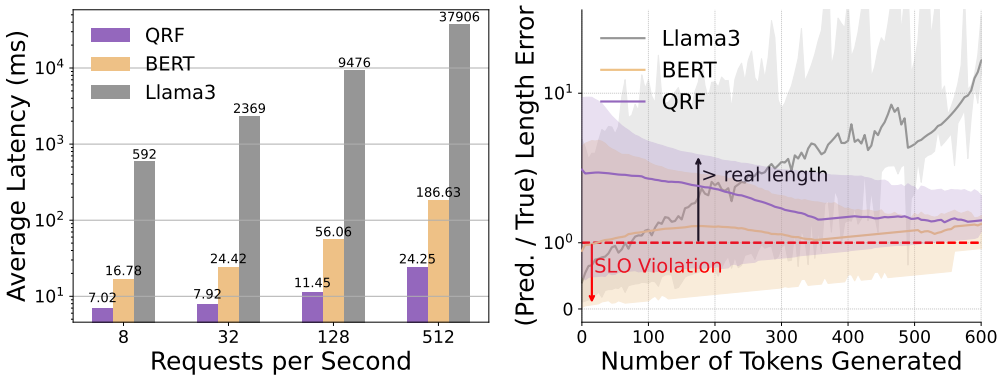
SLO-aware LLM serving must address three fundamental challenges: (1) *Generalizability*: proactively estimating and refining request information during execution to meet diverse SLOs (§4.1), which informs (2) *Goodput Efficiency*: scheduling individual requests and their batch compositions to maximize 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 Concord 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 [30] 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 D.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, Concord 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, Concord 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 [48] 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 4.1(a), each prediction takes only 7 ms, 7× faster than a fine-tuned



(a) Estimation overhead.

(b) Estimation accuracy.

Figure 4.1: (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).

BERT predictor [48].

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 4.1(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 [32]), where nodes represent LLM or tool invocations and edges encode their dependencies. Moreover, these dependency structures may evolve dynamically (e.g., to achieve sufficient response confidence [32]), 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

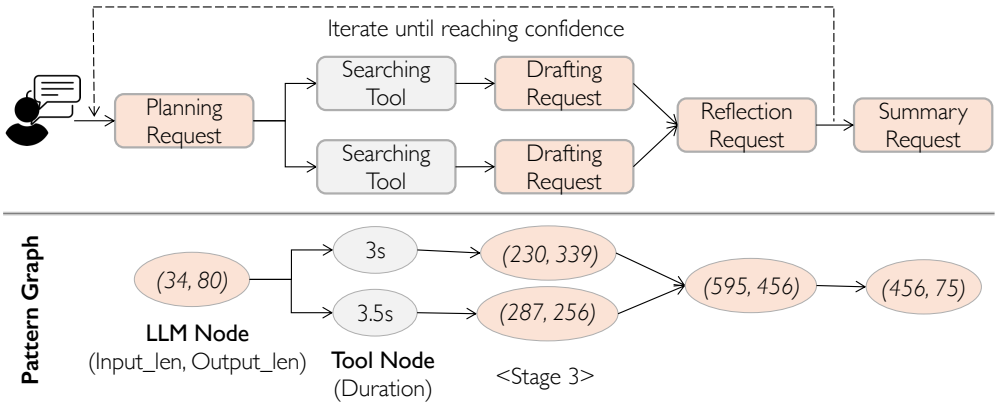


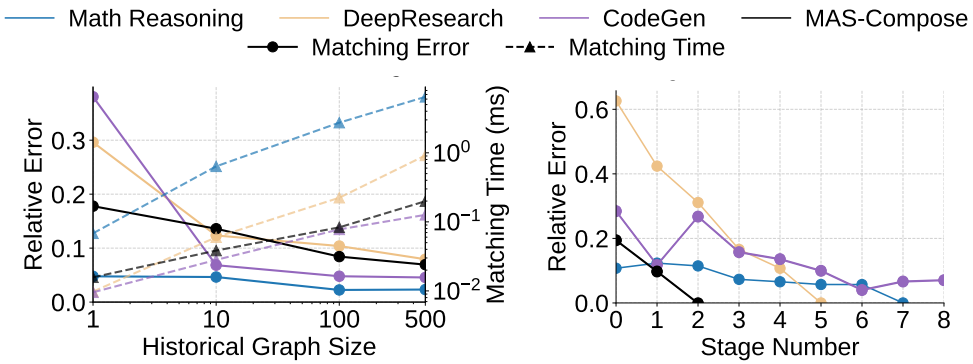
Figure 4.2: Example of Pattern Graph consisting of five stages. LLM node weighted by (input_len, output_len), tool node weighted by execution time.

pattern graph, without needing raw input/output plaintext. As illustrated in Figure 4.2, 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 [53] 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 A), offering greater robustness and accuracy by grouping previous stages' information.

To ensure efficiency, we cluster historical pattern graphs offline using a K-medoids mech-



(a) Impact of historical repo size.

(b) Impact of online matching.

Figure 4.3: (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., $t_s = 0$).

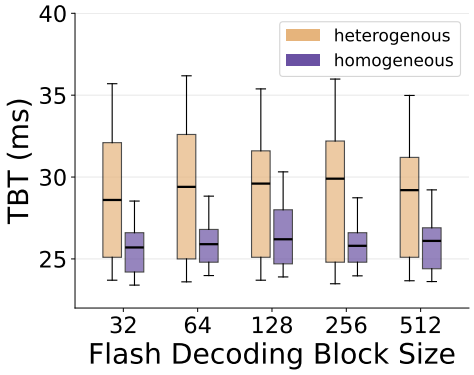


Figure 4.4: Batching requests with heterogeneous lengths slows down execution.

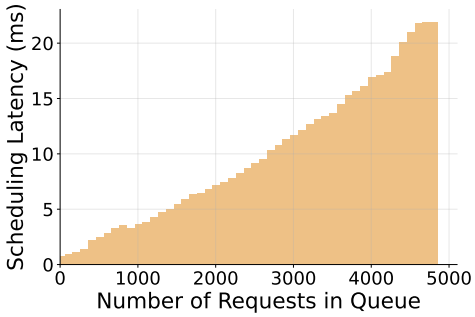


Figure 4.5: *CALM* scales efficiently to schedule thousands of concurrent requests.

anism [54], 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 4.3(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 4.3(b), the relative error in next-stage ratio estimation decreases as additional stage information becomes available, demonstrating progressive refinement.

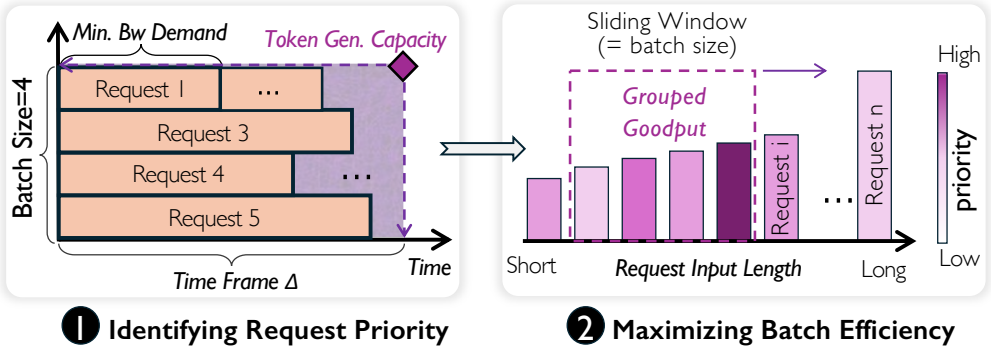


Figure 4.6: *CALM* (1) identifies the scheduling priority of each request from its bandwidth demand, and then (2) selects requests to maximize the grouped margin goodput and batch efficiency.

4.2 SLO-AWARE SCHEDULER WITH *CALM* 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 C.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 [55]. As shown in Figure 4.4, this inefficiency persists even with advanced kernels such as Flash Decoding [56].

Algorithm 4.1 summarizes how Concord addresses the online two-dimensional scheduling challenge with the *Grouped Margin Goodput Maximization* (*CALM*) 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 12-16). As generation progresses, both the Request Analyzer and the scheduler continuously refine request estimates and update scheduling decisions accordingly (Lines 9-10).

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

Algorithm 4.1: *CALM* Algorithm

Class *RequestAnalyzer*:
  Function AnalyzeRequest*(req)*:
    `// Capture minimum serving bandwidth each request needs to meet SLOs`
    req.len_rem ← PredictLength(req);
    req.bw ← $\frac{req.len\_rem}{\text{EstimateRemainingTime}(req)}$;
    req.goodput ← EstimateGoodput(req);
    req.priority ← $\frac{req.goodput}{req.bw}$;

Class *Scheduler*:
  Function Schedule*(batch_size, cutoff)*:
    Q ← GetRequestQueue(); **foreach** *req in Q* **do**
      AnalyzeRequest(req);
    bp ← BatchPriority(Q, batch_size);
    `// Step 1: candidate filtering by priority cutoff` p
    Candidates ← Q.Filter(req: req.priority ≥ bp × cutoff);
    `// Step 2: group by input length (sliding window)`
    Candidates.sort(req: req.input_length); BestGroup ← ∅; max_score ← $-\infty$;
    foreach *window* $\mathcal{G}$ *of size B in Candidates* **do**
      score ← $\sum_{r\in\mathcal{G}} r.priority$; **if** *score ¿ max_score* **then**
        BestGroup ← $\mathcal{G}$; max_score ← score;
    return BestGroup;

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 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), *CALM* amortizes bandwidth allocation over discrete scheduling frames of length Δ. As illustrated in Figure 4.6, 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, *CALM* uses a lightweight design that prioritizes requests by their *margin goodput per unit bandwidth*:

$$Priority(r) = \frac{goodput_\Delta(r)}{bw_\Delta(r)} = \frac{goodput(r)}{t_{\text{gen}}(r)}. \tag{4.1}$$

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, *CALM* 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 4.4). Next, *CALM* extends individual prioritization into a *grouped scheduling strategy* that jointly balances goodput payoff and length homogeneity.

As illustrated in Figure 4.6, let B denote the batch size. *CALM* 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, *CALM* 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 4.4). Fortunately, because LLM serving is long-running, *CALM* 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 relies on both future arrivals and progressively refined request information, introducing uncertainty that can lead to suboptimal online decisions. Preempting running requests to correct these errors, however, incurs non-trivial overheads (e.g., batch stalls and KV cache swaps), which may outweigh benefits if not carefully controlled.

Concord mitigates this with two safeguards. First, because preemption overhead scales with the KV cache size to be released, the resulting stall time is predictable given the KV cache size and I/O bandwidth. Concord estimates the potential goodput loss as: $goodput\_loss = stall\_duration \times token\_generation\_speed$, and performs preemption only when the projected gain from admitting a higher-priority request exceeds this cost, thereby ensuring a net benefit. Second, to prevent 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 Concord to correct scheduling decisions with negligible overhead ($< 1\%$) in practice (§6.2).

Achievable Guarantees. We next analyze the scheduling efficiency and quality of *CALM*: (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 4.5 shows that *CALM* 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 [57], we prove that *CALM* achieves a competitive performance guarantee relative to the even *offline* optimal scheduler with future request arrival information. A detailed proof is provided in Appendix D.3, specifically:

Theorem 4.1. Let $G_{CALM}(\mathcal{R})$ denote the goodput achieved by online *CALM* 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 $G_{CALM}(\mathcal{R}) \geq \frac{1}{8.56} \cdot G^{\star}(\mathcal{R})$.

4.3 CONCORD 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 Concord scales efficiently—its request estimation and refinement can run in parallel across requests, and *CALM* 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 *CALM* 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 *CALM*'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 Concord consistently achieves superior performance in multi-model deployments (§6.3.1).

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.

Concord can seamlessly incorporate additional objectives, such as fairness, with minimal changes. Given a developer-specified fairness function $\mathrm{Fair}(r)$, we redefine the request priority as $priority'(r) = (1-f) \cdot priority(r) + f \cdot \mathrm{Fair}(r)$, 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 Concord to enforce diverse fairness policies while offering flexible tradeoffs.

CHAPTER 5: IMPLEMENTATION

We implemented Concord atop vLLM [13] with about 2,800 lines of code, preserving its APIs for broad compatibility.

Execution Backend. Concord 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. Concord further inherits vLLM's prefix caching [58] 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. Concord extends the OpenAI API [59] 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, Concord enforces a maximum waiting_time (e.g., 5 seconds): requests unscheduled beyond it are dropped to prevent overload, ensuring predictable service behavior.

CHAPTER 6: EVALUATION

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

1. Concord 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);

2. Concord effectively balances performance across diverse request types, achieving strong P50 metrics and comparable P95 tail latency for all request patterns (§6.3);

3. Concord demonstrates robustness across different SLO requirements, workload compositions, and distributed settings, consistently outperforming baselines (§6.3.1).

6.1 EXPERIMENTAL SETUP

We evaluate Concord on a set of widely used LLMs with diverse architectures, including Llama-3.1-8B [60], Qwen2.5-14B [61], Qwen3-30B-MoE-A3B [62], and Llama-3.1-70B [60]. 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 [63] and LMsys-chat [19] datasets to build the Chatbot application. We further incorporate a long-context math reasoning application [11], a deep research application based on the Search Arena benchmark [64], and an agentic code generation application [16]. From LMsys-chat usage analysis [19], we extract the distribution of real-world use cases. Requests are tagged according to statistics from our user study (Table 2.1). For example, 38.1% of code generation requests are classified as latency-sensitive, while the deep research application is modeled as compound requests. Table 6.1 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 [65, 66], 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-

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 6.1: 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.

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.

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 Concord achieves consistent improvement across different settings of SLO requirements and workload compositions (§6.3.1).

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

1. *Autellix* [30]: Uses Program-level Least Attained Service scheduling (PLAS) to imitate SJF, optimizing request completion time (E2EL) for compound workloads.

2. *Learn to Rank (LTR)* [67]: Leverages an LLM prediction model to predict the relative response length ranking of requests and prioritize the smallest one, imitating SJF.

3. *vLLM* [65]: A recent advanced LLM serving backend with continuous batching [55] and PagedAttention [65]. Uses FCFS scheduling.

4. *Sarathi-Serve* [12]: Extends vLLM with chunked prefills to optimize TTFTs and TBTs for requests.

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

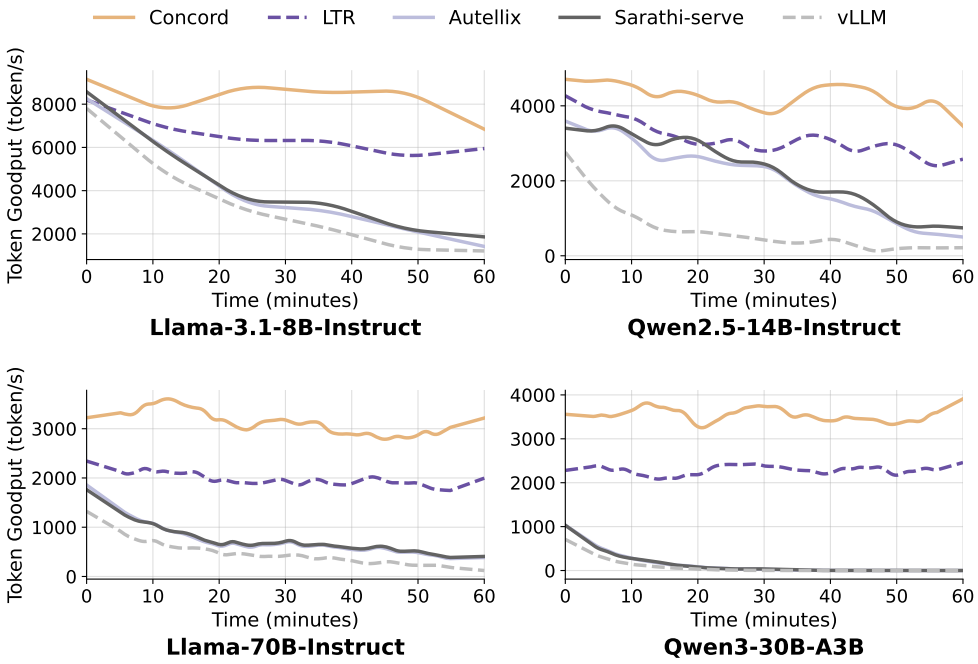


Figure 6.1: Service goodput over time in a one-hour online serving experiment. Concord achieves consistently high service good while the baselines suffer cascading SLO violations and degraded service goodput over time.

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.

Concord substantially improves service goodput. We first deploy Concord in a one-hour online experiment to evaluate its long-term performance. As shown in Figure 6.1, Concord 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 Concord'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. Concord mitigates this degradation by leveraging conservative upper-bound length

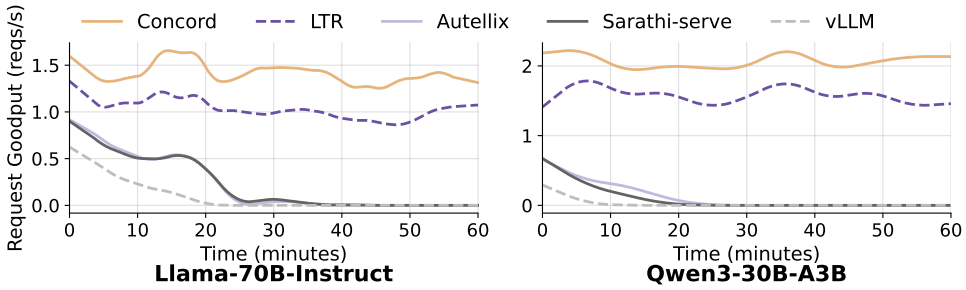


Figure 6.2: Concord achieves consistently better request-level SLO service goodput in online deployments.

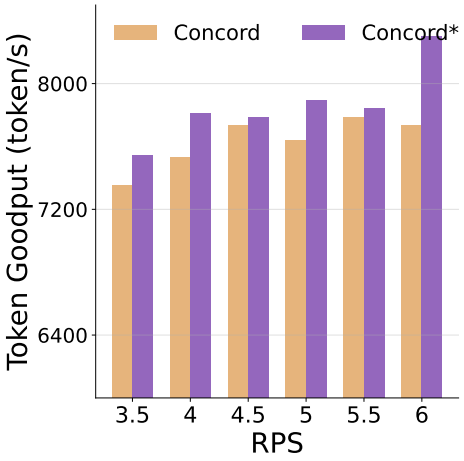


Figure 6.3: Concord achieves close-to-oracle performance.

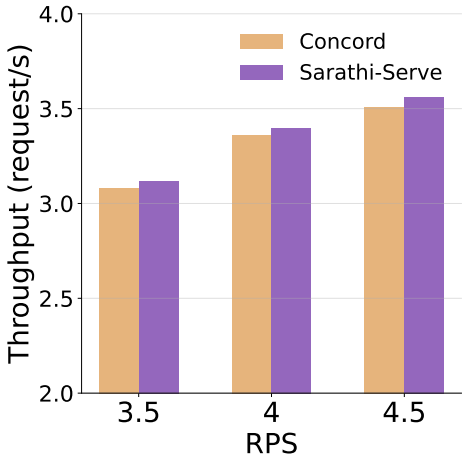


Figure 6.4: Concord introduces little overhead in throughput.

predictions and maximizing residual bandwidth for other requests, resulting in stable service goodput throughout the evaluation. Notably, Concord 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 6.2, Concord achieves 2.3×-4.5× higher goodput than LTR.

Concord achieves near-oracle performance. We further benchmark Concord against an oracle variant, Concord *, 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 6.3, Concord 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 informa-

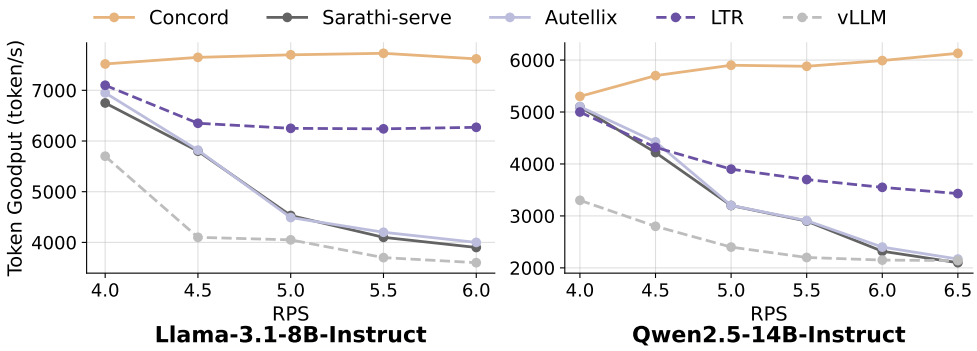


Figure 6.5: Concord sustains high goodput across request loads.

tion and progressively relaxing the conservatism to make near-optimal decisions. Together, these results demonstrate that Concord approaches the best achievable performance without strong assumptions.

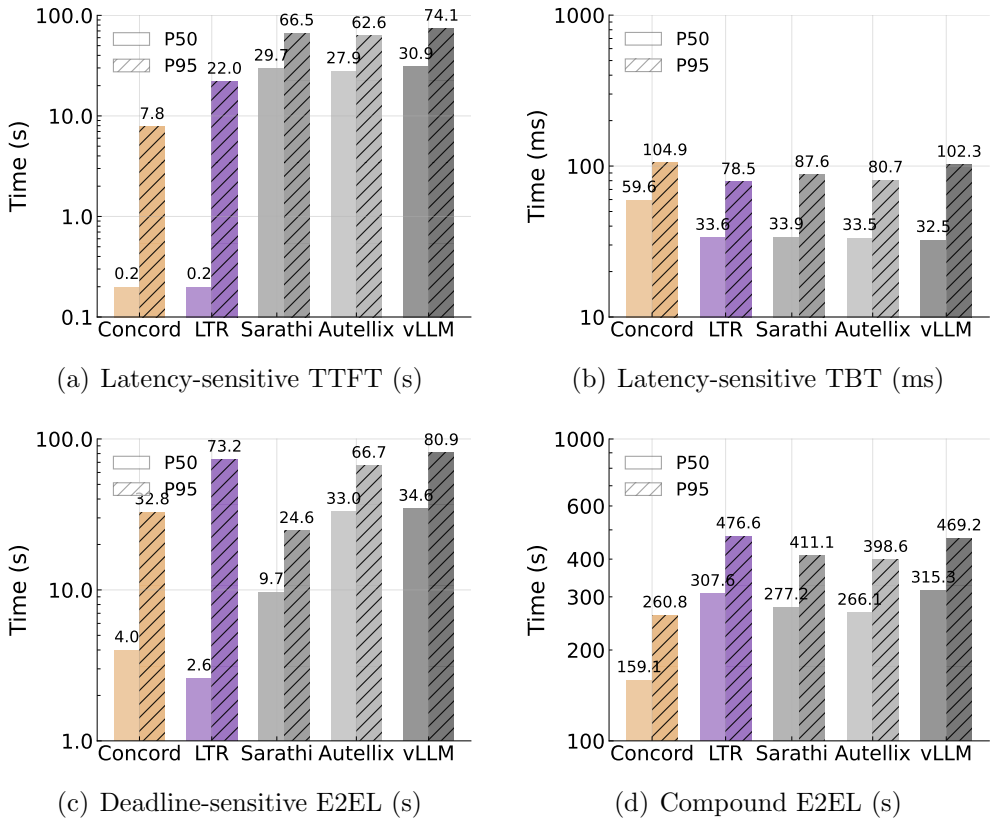
Concord does not hurt system throughput. A common concern with sophisticated scheduling is the potential throughput loss. To evaluate this, we compare Concord against Sarathi-Serve, which employs FIFO scheduling without preemption and thus represents a near upper-bound on serving throughput. As shown in Figure 6.4, Concord achieves comparable throughput to Sarathi-Serve across different request-per-second (RPS) settings, reaching 96%–98% of its performance. This demonstrates that Concord'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).

Concord sustains high goodput under load surges. To evaluate scalability and robustness, we measure service goodput under varying request arrival rates. As shown in Figure 6.5, all baselines exhibit sharp performance drop as system load increases due to contention. In contrast, Concord consistently achieves the highest goodput across all load levels by dynamically adjusting priorities and resource allocations.

As a result, Concord 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 Concord's performance by breaking it down into two key aspects: its ability to handle different request types, and its performance when key components are



(a) Latency-sensitive TTFT (s)

(b) Latency-sensitive TBT (ms)

(c) Deadline-sensitive E2EL (s)

(d) Compound E2EL (s)

Figure 6.6: 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.

ablated. By understanding these aspects, we gain deeper insights into Concord's effectiveness and how it performs in various scenarios.

Breakdown by Request Types. While we have demonstrated Concord's large goodput improvement, we next study how Concord handles diverse SLOs across different request types, using conventional performance metrics such as TTFT, TBT, and E2EL. As shown in Figure 6.6(a), Concord 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 6.6(b)). For deadline-sensitive and compound requests, Concord achieves favorable median E2EL (Figures 6.6(c) and 6.6(d)), ensuring that many requests meet their SLOs without oversubscribing bandwidth merely to minimize the average E2EL. Across all request types, Concord 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

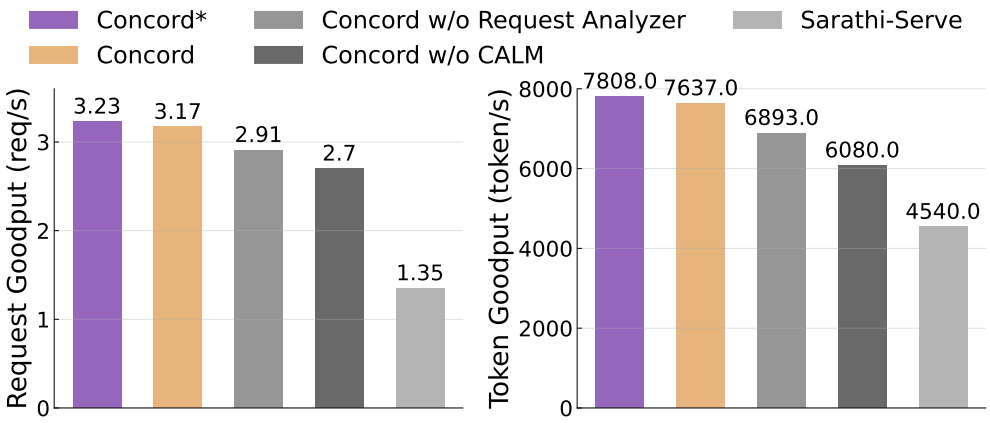


Figure 6.7: Request analyzer and *CALM* algorithm jointly contribute to Concord's high-quality, resilient serving.

of a small subset of requests. Our results show that Concord 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, Concord 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 2.3) 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 6.7 presents a component-level breakdown of Concord and several ablated variants: (1) Concord with precise knowledge (Concord *), (2) Concord without the Request Analyzer (falling back to average response length estimation), (3) Concord without *CALM* scheduling (replaced by SJF scheduling based on Request Analyzer estimates), and Sarathi-Serve. Beyond the near-oracle performance achieved by Concord in terms of SLO goodput, we observe that removing either the Request Analyzer or *CALM* results in a noticeable degradation in goodput, highlighting their essential roles in the design.

6.3.1 Sensitivity and Ablation Studies

Extending to Multiple Models. We next investigate Concord's performance when serving multiple model replicas with data parallelism. We scale the request arrival rates propor-

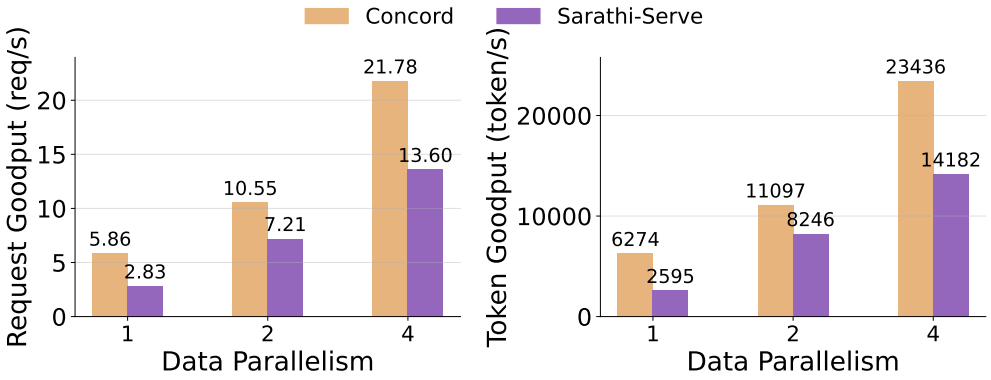


Figure 6.8: Concord scales effectively to multi-model deployments.

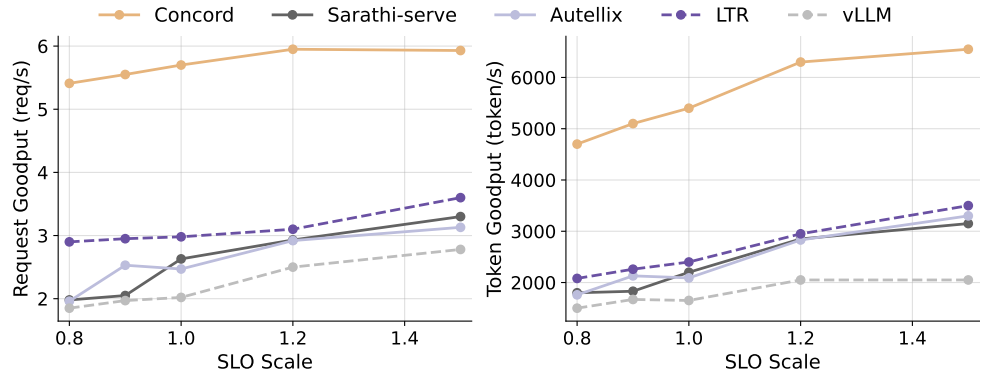


Figure 6.9: Concord outperforms across various SLO tightness.

tionally to the number of model replicas. As shown in Figure 6.8, while all systems achieve higher goodput with additional replicas, Concord consistently outperforms the baseline by 1.34×–2.42× across all configurations.

Impact of SLO Constraints. We evaluate how Concord 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 6.9, relaxing SLO constraints naturally improves the SLO goodput. Concord 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 6.10 shows that Concord consistently achieves higher service goodput than existing

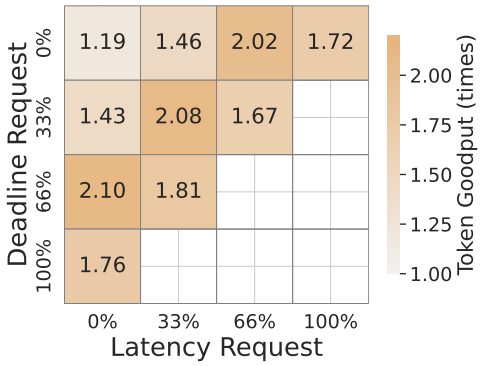


Figure 6.10: Concord maintains higher performance across varying workload compositions.

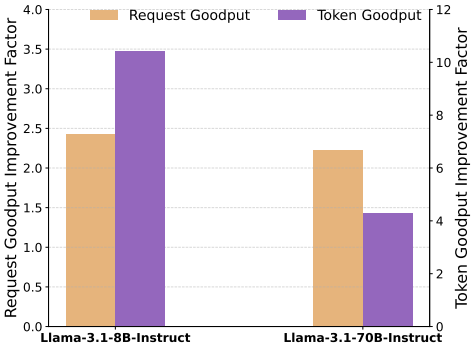


Figure 6.11: Concord outperforms earliest deadline first.

approaches, such as 1.8× goodput improvement in a setting with 33% latency- and 66% deadline-sensitive workloads. Notably, Concord outperforms Sarathi-Serve by 1.72× even for latency-sensitive-only requests, its intended design point.

Comparison to Earliest-Deadline First. We here summarize the performance comparison to the earlies-deadline first (EDF) policy. Figure 6.11 shows that Concord achieves better goodput since EDF does not account for the heterogeneous SLOs and goodput.

CHAPTER 7: RELATED WORKS

LLM Serving System. Recent advancements in LLM have led to the development of numerous inference systems. Orca [55], dLoRA [68], VTC [69], and FastServe [70] present varied strategies for batching, concurrent serving, and fairness. vLLM [65] and InfiniGen [71] focus on KV-cache management to improve memory utilization and throughput. DistServe [51], Sarathi-Serve [12], Llumnix [66], and Splitwise [72] capitalize on characteristics of the prefilling and decoding stages to minimize TTFT and TBT latency. Some other systems explore GPU kernel optimizations [73, 74, 75, 76, 77], model parallelism [78, 79], and preemptive scheduling [80]. However, most prior works primarily target single-request latency without accounting for the diverse requirements of different applications. Parrot [28] leverages the interconnections within LLM applications. Concord builds on these approaches, categorizing LLM applications into three types and addressing their SLO requirements collectively. Parrot [28] offers APIs to extract LLM request execution dependency. Concord builds on them, categorizing LLM applications and addressing their SLO requirements collectively.

LLM Output Length Prediction. Recent efforts such as TetriInfer [81], S^3[82], and u-Serve[83] 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). Concord 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 [41] performs deadline-aware flow scheduling to reserve bandwidth for best-effort flows, while QCLIMB [84] uses QRF models to predict lower bounds on flow sizes for flow scheduling. CASSINI [85] schedules ML training traffic across jobs to reduce network contention, and Caladan [86] mitigates resource contention in hyperthreads to reduce OS tail latency. AdaServe [7] supports customizable SLOs through fine-grained speculative decoding. Concord complements AdaServe by orchestrating a broader range of diverse SLO requirements across different LLM request types, aiming to maximize service goodput without request information.

CHAPTER 8: FUTURE WORK

From Optimized Serving to Stateful Orchestration While this thesis has presented Concord as a robust solution for the serving layer—optimizing token generation against strict SLOs—the rapid evolution of AI towards agentic workflows exposes a critical gap in the orchestration layer. Current frameworks excel at defining agent interactions but lack a sophisticated runtime to manage the complex, stateful side-effects of these agents. As agents evolve from passive chatbots to active operators that modify file systems, install packages, and execute code, the "state" of the application transcends the LLM's context window. It encompasses the entire environment configuration and execution history.

Proposed Solution: The ML Agent Compiler To address this, we propose the development of an ML Agent Compiler, a runtime system analogous to a Java Virtual Machine (JVM) for multi-agent systems (MAS). Just as a JVM abstracts memory management and platform dependencies, the ML Agent Compiler is designed to abstract and manage System and Environment State. The core of this proposed architecture is a runtime compiler that treats agent workflows as executable graphs. Unlike current implementations where error recovery often requires restarting a task from scratch, this engine aims to map the high-level Agent Workflow Graph to a low-level System State Graph. This mapping allows the runtime to track attribute propagation (distinguishing between mutable and revertible changes) and manage dependencies between agent actions and environmental side-effects.

Graph-Based Checkpointing and Deduplication A primary technical challenge in this domain is the cost of state management. Our proposed design introduces a mechanism for Graph-Based Checkpointing, enabling the system to "fork" and "merge" execution paths efficiently. By utilizing shared checkpoints for common states and implementing a liveness counter to deallocate unused resources, the system can support complex exploration strategies (e.g., Tree of Thoughts) without prohibitive overhead. Furthermore, the ML Agent Compiler will implement Lookahead Compilation and State Deduplication. Similar to lineage tracking in distributed data systems (e.g., Spark RDDs), the compiler will detect redundant partial executions—such as a tool usage pattern that has already been computed in a parallel branch—and reuse the cached result. This approach not only ensures fault tolerance via three layers of recovery methods but also guides cost-effective execution by speculating on the computational cost of forking versus re-computation. Through these mechanisms, we aim to bring the same level of rigorous optimization to agent orchestration that Concord brings to LLM serving.

CHAPTER 9: CONCLUSIONS

We introduce Concord, an LLM request scheduler designed to maximize service goodput. Concord 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.

REFERENCES

[1] Amzon, "Amazon q - generative ai assistant," 2024. [Online]. Available: https://aws.amazon.com/q/?nc1=h_ls

[2] Y. Zhang, Z. Ma, Y. Ma, Z. Han, Y. Wu, and V. 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*, vol. 39, no. 22, 2025, pp. 23 378–23 386.

[3] W. Yin, M. Xu, Y. Li, and X. Liu, "Llm as a system service on mobile devices," *arXiv preprint arXiv:2403.11805*, 2024.

[4] Z. Wang, S. Li, Y. Zhou, X. Li, R. Gu, N. Cam-Tu, C. Tian, and S. Zhong, "Revisiting slo and goodput metrics in llm serving," in *arXiv: 2410.14257*, 2024.

[5] N. Muennighoff, Z. Yang, W. Shi, X. L. Li, L. Fei-Fei, H. Hajishirzi, L. Zettlemoyer, P. Liang, E. Candès, and T. Hashimoto, "s1: Simple test-time scaling," *arXiv: 2501.19393*, 2025.

[6] G. Mialon, C. Fourrier, T. Wolf, Y. LeCun, and T. Scialom, "Gaia: a benchmark for general ai assistants," in *The Twelfth International Conference on Learning Representations*, 2023.

[7] Z. Li, Z. Chen, R. Delacourt, G. Oliaro, Z. Wang, Q. Chen, S. Lin, A. Yang, Z. Zhang, Z. Chen, S. Lai, X. Miao, and Z. Jia, "Adaserve: Slo-customized llm serving with fine-grained speculative decoding," *arXiv preprint arXiv: 2501.12162*, 2025.

[8] M. Shetty, Y. Chen, G. Somashekar, M. Ma, Y. Simmhan, X. Zhang, J. Mace, D. Vandevoorde, P. Las-Casas, S. M. Gupta, S. Nath, C. Bansal, and S. Rajmohan, "Building ai agents for autonomous clouds: Challenges and design principles," in *SoCC*, 2024.

[9] G. Gemini, 2025, https://gemini.google.com/.

[10] Z. Yu, M. Ma, C. Zhang, S. Qin, Y. Kang, C. Bansal, S. Rajmohan, Y. Dang, C. Pei, D. Pei, Q. Lin, and D. Zhang, "Monitorassistant: Simplifying cloud service monitoring via large language models," in *FSE*, 2024.

[11] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *NeurIPS*, vol. 36, 2024.

[12] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve," in *OSDI*, 2024, pp. 117–134.

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

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

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

[16] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu et al., "Autogen: Enabling next-gen llm applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.

[17] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[18] H. Trivedi, T. Khot, M. Hartmann, R. Manku, V. Dong, E. Li, S. Gupta, A. Sabharwal, and N. Balasubramanian, "AppWorld: A controllable world of apps and people for benchmarking interactive coding agents," in *ACL*, 2024.

[19] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing, J. E. Gonzalez, I. Stoica, and H. Zhang, "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," 2023.

[20] W. Zhao, X. Ren, J. Hessel, C. Cardie, Y. Choi, and Y. Deng, "Wildchat: 1m chatgpt interaction logs in the wild," 2024. [Online]. Available: https://arxiv.org/abs/2405.01470

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

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

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

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

[25] C. Xiao and B. Yang, "Streaming, fast and slow: Cognitive load-aware streaming for efficient llm serving," in *UIST*, 2025.

[26] F. Biester, M. Abdelaal, and D. Del Gaudio, "Llmclean: Context-aware tabular data cleaning via llm-generated ofds," in *European Conference on Advances in Databases and Information Systems*. Springer, 2024, pp. 68–78.

[27] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[28] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of LLM-based applications with semantic variable," in *ODSI*, 2024, pp. 929–945.

[29] X. Tan, Y. Jiang, Y. Yang, and H. 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*, 2025, pp. 1302–1316.

[30] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez, and I. Stoica, "Autellix: An efficient serving engine for llm agents as general programs," in *arXiv:2502.13965*, 2025.

[31] L. Wang, W. Xu, Y. Lan, Z. Hu, Y. Lan, R. K.-W. Lee, and E.-P. Lim, "Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models," *arXiv preprint arXiv:2305.04091*, 2023.

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

[33] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray et al., "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[34] J. Wang, W. Ma, P. Sun, M. Zhang, and J.-Y. Nie, "Understanding user experience in large language model interactions," *arXiv preprint arXiv:2401.08329*, 2024.

[35] Z. Li, H. Zhu, Z. Lu, and M. Yin, "Synthetic data generation with large language models for text classification: Potential and limitations," *arXiv preprint arXiv:2310.07849*, 2023.

[36] J. Liu, Z. Wu, J.-W. Chung, F. Lai, M. Lee, and M. Chowdhury, "Andes: Defining and enhancing quality-of-experience in llm-based text streaming services," *arXiv preprint arXiv:2404.16283*, 2024.

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

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

[39] S. Qiao, R. Fang, Z. Qiu, X. Wang, N. Zhang, Y. Jiang, P. Xie, F. Huang, and H. Chen, "Benchmarking agentic workflow generation," *arXiv preprint arXiv:2410.07869*, 2024.

[40] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," *SIGCOMM*, 2014.

[41] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with karuna," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2934872.2934888 pp. 174–187.

[42] V. Sachidananda and A. Sivaraman, "Erlang: Application-aware autoscaling for cloud microservices," in *EuroSys*, 2024.

[43] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, "Automatic, Application-Aware I/O forwarding resource allocation," in *FAST*, 2019.

[44] S. Jalalian, S. Patel, M. R. Hajidehi, M. Seltzer, and A. Fedorova, "ExtMem: Enabling Application-Aware virtual memory management for Data-Intensive applications," in *ATC*, 2024.

[45] J. Wang, D. S. Berger, F. Kazhamiaka, C. Irvene, C. Zhang, E. Choukse, K. Frost, R. Fonseca, B. Warrier, C. Bansal, J. Stern, R. Bianchini, and A. Sriraman, "Designing cloud servers for lower carbon," in *ISCA*, 2024.

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

[47] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang et al., "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 534–46 594, 2023.

[48] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. Kalbarczyk, T. Başar, and R. K. Iyer, "Power-aware deep learning model serving with μ-Serve," in *ATC*, 2024.

[49] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An llm-empowered llm inference pipeline," *NeurIPS*, vol. 36, 2024.

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

[51] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *OSDI*, 2024, pp. 193–210.

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

[53] N. M. Kriege, F. D. Johansson, and C. Morris, "A survey on graph kernels," *Applied Network Science*, vol. 5, pp. 1–42, 2020.

[54] J. Newling and F. Fleuret, "K-medoids for k-means seeding," in *NIPS*, 2017.

[55] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *OSDI*, 2022, pp. 521–538.

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

[57] J. Kintz, "Amortized analysis," 2018.

[58] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," in *NeurIPS*, 2024.

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

[60] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan et al., "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[61] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei et al., "Qwen2. 5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[62] Q. Team, "Qwen3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[63] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. 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*, vol. 3, no. 6, p. 7, 2023.

[64] M. Miroyan, T.-H. Wu, L. K. King, T. Li, A. N. Angelopoulos, W.-L. Chiang, N. Norouzi, and J. E. Gonzalez, "Introducing the search arena: Evaluating search-enabled ai," April 2025. [Online]. Available: https://blog.lmarena.ai/blog/2025/search-arena/

[65] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *SOSP*, 2023, pp. 611–626.

[66] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *OSDI*, 2024, pp. 173–191.

[67] Y. Fu, S. Zhu, R. Su, A. Qiao, I. Stoica, and H. Zhang, "Efficient llm scheduling by learning to rank," *NeurIPS*, 2024.

[68] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin, "dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving," in *OSDI*, 2024, pp. 911–927.

[69] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in serving large language models," in *OSDI*, 2024, pp. 965–988.

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

[71] W. Lee, J. Lee, J. Seo, and J. Sim, "InfiniGen: Efficient generative inference of large language models with dynamic KV cache management," in *OSDI*, 2024, pp. 155–172.

[72] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *ISCA*. IEEE, 2024, pp. 118–132.

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

[74] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "Turbotransformers: an efficient gpu serving system for transformer models," in *PPoPP*, 2021, pp. 389–402.

[75] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

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

[77] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *arXiv preprint arXiv:2407.08608*, 2024.

[78] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez et al., "{AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving," in *OSDI*, 2023, pp. 663–679.

[79] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. 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*. IEEE, 2022, pp. 1–15.

[80] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," in *ASPLOS*, 2024, pp. 1112–1127.

[81] C. Hu, H. Huang, L. Xu, X. Chen, J. Xu, S. Chen, H. Feng, C. Wang, S. Wang, Y. Bao et al., "Inference without interference: Disaggregate llm inference for mixed downstream workloads," *arXiv preprint arXiv:2401.11181*, 2024.

[82] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "S^3: Increasing gpu utilization during generative inference for higher throughput," *NeurIPS*, vol. 36, pp. 18 015–18 027, 2023.

[83] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Efficient interactive llm serving with proxy model-based sequence length prediction," *arXiv preprint arXiv:2404.08509*, 2024.

[84] W. Li, X. He, Y. Liu, K. Li, K. Chen, Z. Ge, Z. Guan, H. Qi, S. Zhang, and G. Liu, "Flow scheduling with imprecise knowledge," in *NSDI*, 2024.

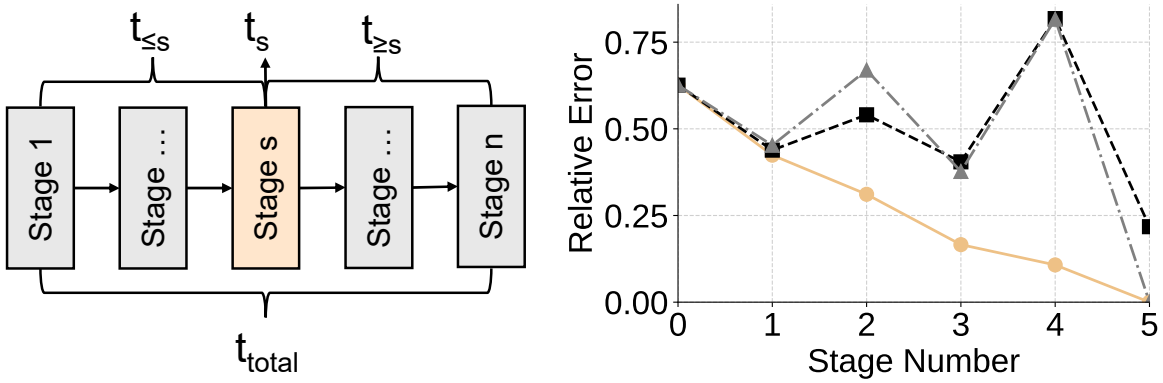
[85] S. Rajasekaran, M. Ghobadi, and A. Akella, "CASSINI: Network-Aware job scheduling in machine learning clusters," in *NSDI*, 2024.

[86] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *OSDI*, 2020.

[87] W. Gautschi, *Numerical analysis*. Springer Science & Business Media, 2011.

APPENDIX A: 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 A.1(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 A.1: Illustration and impact of different sub-deadline formulations.

APPENDIX B: NOTATIONS FOR THEORETICAL ANALYSIS

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*

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

and *scheduling indicator*

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

together with $t^{\text{r}}_{\text{SLO}} - t^{\text{r}}_{\text{comp}}(k) \geq 0$ scheduling filter and $\epsilon > 0$ 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

$$\mathsf{Goodput}(\sigma) \;:=\; \sum_{k:\, C_k \leq t_{\text{SLO}}(k)} R(k) \tag{B.3}$$

the realized total on-time goodput of schedule σ.

APPENDIX C: COMPLEX ANALYSIS

C.1 OPTIMAL SCHEDULING

Theorem C.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 $t_{\text{comp}}(k) > 0$, a start time $s(k)$, an SLO time $t_{\text{SLO}}(k)$, 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

$$\sigma = \arg\max_{\sigma} \ \mathsf{Goodput}(\sigma) \tag{C.1}$$

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:

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

$$t_{\text{comp}}(r_i) = w_i, \quad s(r_i) = 0, \quad t_{\text{SLO}}(r_i) = C, \quad R(r_i) = v_i. \tag{C.2}$$

3. 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. QED.

APPENDIX D: COMPETITIVE RATIO ANALYSIS

D.1 ANALYSIS OF POPULAR SCHEDULING POLICIES

D.1.1 Analysis of Earliest Deadline First Scheduling

Theorem D.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

$$\frac{\mathsf{Goodput}(\mathrm{OPT})}{\mathsf{Goodput}(\mathrm{EDF})} > r \tag{D.1}$$

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:

1. 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.
2. 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 $t_{\text{SLO}}(B_i) = T + \delta$, 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 $\mathsf{Goodput}(\mathrm{OPT}) = M$.

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

$$\frac{\mathsf{Goodput}(\mathrm{OPT})}{\mathsf{Goodput}(\mathrm{EDF})} = \frac{M}{N} \tag{D.2}$$

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

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.

D.2 ANALYSIS OF SHORTEST JOB FIRST SCHEDULING

Theorem D.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

$$\frac{\mathsf{Goodput}(\mathrm{OPT})}{\mathsf{Goodput}(\mathrm{SJF})} > r \tag{D.3}$$

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:

1. 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.

2. 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 $t_{\mathrm{SLO}}(B_i) = \delta$, 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 $N \cdot \delta = \dfrac{N \cdot T}{N+1} < T$, 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

$$\frac{\mathsf{Goodput}(\text{OPT})}{\mathsf{Goodput}(\text{SJF})} \;=\; \frac{M}{N} \tag{D.4}$$

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

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.

D.3 ANALYSIS OF CONCORD SCHEDULING

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

$$\frac{R(A)}{R(B)} \;>\; 1 + \delta \qquad \text{for } \delta > 0 \tag{D.5}$$

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

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

$$\frac{\mathsf{Goodput}(Concord)}{\mathsf{Goodput}(\mathrm{OPT})} \geq r \tag{D.6}$$

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 Concord 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 Concord 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 system's 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 Concord'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:

1. **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:

$$\sum_{V \in \sigma_*} f(U, V) = R(U), \quad \forall U \in \sigma_S \tag{D.7}$$

2. **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 $V \in \sigma_*$:

$$\sum_{V \in \sigma_*} \sum_{U \in \sigma_S^c} f(U, V) \geq r \sum_{V \in \sigma_*} R(V) \tag{D.8}$$

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

$$\frac{\mathsf{Goodput}(\sigma_S)}{\mathsf{Goodput}(\sigma_*)} \geq r \tag{D.9}$$

Proof of lemma.

$$\begin{aligned}
\mathsf{Goodput}(\sigma_S) &= \sum_{U \in \sigma_S^c} R(U) \\
&= \sum_{U \in \sigma_S^c} \sum_{V \in \sigma_*} f(U, V) \\
&= \sum_{V \in \sigma_*} \sum_{U \in \sigma_S^c} f(U, V) \\
&\geq \sum_{V \in \sigma_*} rR(V) \\
&= r \sum_{V \in \sigma_*} R(V) \\
&\geq r \cdot \mathsf{Goodput}(\sigma_*)
\end{aligned} \tag{D.10}$$

QED.

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

With lemma D.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:

1. **Rule 1:** If $s_S(V) \leq s_*(U)$, and $t^{\mathrm{r}}_{\mathrm{comp}}(V) > t^{\mathrm{r}}_{\mathrm{comp}}(U)$, we set $f(U, V) := \alpha \cdot R(U)$.
2. **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 $f(U, V) := \beta \cdot \dfrac{t^{\mathrm{r}}_{\mathrm{comp}}(V) + \epsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U) + \epsilon} \cdot R(U)$.
3. **Rule 3:** If V and U share the same request, we set $f(U, V) := \gamma \cdot (1 + \delta)^3 \cdot R(U)$.
4. **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 D.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:

$$\begin{aligned}
&\sum_{V\in\sigma_*} g(U,V) \\
=\ &\sum_{\substack{V:s_*(V)\geq s_S(U)\\ t_{\mathrm{SLO}}(V)>t_{\mathrm{SLO}}(U)}} \alpha\cdot R(U) \\
&+\sum_{\substack{V:s_*(V)\geq s_S(U)\\ t_{\mathrm{SLO}}(V)\leq t_{\mathrm{SLO}}(U)}} \beta\cdot\frac{t^{\mathrm{r}}_{\mathrm{comp}}(V)+\epsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(U)+\epsilon}\cdot R(U) \\
&+\sum_{V:V=U} \gamma\cdot R(U) \\
\leq\ &\sum_{\substack{V:s_*(V)\geq s_S(U)\\ t_{\mathrm{SLO}}(V)>t_{\mathrm{SLO}}(U)}} \alpha\cdot R(U) \ +\sum_{\substack{V:s_*(V)\geq s_S(U)\\ t_{\mathrm{SLO}}(V)\leq t_{\mathrm{SLO}}(U)}} \beta\cdot R(U) \\
&+\sum_{V:V=U} \gamma\cdot R(U) \\
=\ &(\alpha+\beta+\gamma)\cdot R(U) \\
\leq\ &R(U)
\end{aligned} \tag{D.11}$$

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

$$\begin{aligned}
\sum_{V\in\sigma_*} f(U,V) &= \sum_{V\in\sigma_*}\Big(g(U,V)+h(U,V)\Big) \\
&= \sum_{V\in\sigma_*} g(U,V)+\sum_{V\in\sigma_*} h(U,V) \\
&= (\alpha+\beta+\gamma)\cdot R(U)+(1-\alpha-\beta-\gamma)\cdot R(U) \\
&= R(U)
\end{aligned} \tag{D.12}$$

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

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 $t_V = t_U + \epsilon > t_U$. 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 D.1. We have $s_S(V) \le s_*(U)$ 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:

1. V is not served because

$$\frac{R(V)}{t^{\mathrm{r}}_{\mathrm{comp}}(V) + \epsilon} \le \frac{R(U)}{t^{\mathrm{r}}_{\mathrm{comp}}(U) + \epsilon} \tag{D.13}$$

   Combined with **Rule 1** and Eq. D.13, we have:

$$\sum_{U \in \sigma_S} f(U, V) \ge \alpha \cdot R(U) \tag{D.14}$$

$$\tag{D.15}$$

$$\ge \alpha \cdot \frac{t^{\mathrm{r}}_{\mathrm{comp}}(U) + \epsilon}{t^{\mathrm{r}}_{\mathrm{comp}}(V) + \epsilon} \cdot R(V) \tag{D.16}$$

$$\tag{D.17}$$

$$\ge \alpha \cdot R(V) \tag{D.18}$$

2. V is not served because

$$\frac{R(V)}{R(U)} \le 1 + \delta \tag{D.19}$$

Combined with **Rule 1** and Eq. D.19, we have:

$$\sum_{U \in \sigma_S} f(U, V) \geq \alpha \cdot R(U) \tag{D.20}$$

$$\tag{D.21}$$

$$\geq \alpha \cdot \frac{R(V)}{1+\delta} \tag{D.22}$$

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

1. V is not served because

$$\frac{R(V)}{t^{\text{r}}_{\text{comp}}(V) + \epsilon} \leq \frac{R(U)}{t^{\text{r}}_{\text{comp}}(U) + \epsilon} \tag{D.23}$$

Combined with **Rule 2** and Eq. D.23, we have:

$$\sum_{U \in \sigma_S} f(U, V) \geq \beta \cdot \frac{t^{\text{r}}_{\text{comp}}(V) + \epsilon}{t^{\text{r}}_{\text{comp}}(U) + \epsilon} \cdot R(U) \tag{D.24}$$

$$\geq \beta \cdot \left(\frac{t^{\text{r}}_{\text{comp}}(V) + \epsilon}{t^{\text{r}}_{\text{comp}}(U) + \epsilon} \right) \tag{D.25}$$

$$\cdot \left(\frac{t^{\text{r}}_{\text{comp}}(U) + \epsilon}{t^{\text{r}}_{\text{comp}}(V) + \epsilon} \right) \cdot R(V) \tag{D.26}$$

$$\geq \beta \cdot R(V) \tag{D.27}$$

2. V is not served because

$$\frac{R(V)}{R(U)} \leq 1 + \delta \tag{D.28}$$

Combined with **Rule 2** and Eq. D.28, we have:

$$\sum_{U \in \sigma_S} f(U, V) \geq \beta \cdot \frac{t^{\text{r}}_{\text{comp}}(V) + \epsilon}{t^{\text{r}}_{\text{comp}}(U) + \epsilon} \cdot R(U) \tag{D.29}$$

$$\geq \beta \cdot R(U) \tag{D.30}$$

$$\geq \beta \cdot \frac{R(V)}{1+\delta} \tag{D.31}$$

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:

$$\sum_{U \in \sigma_S} f(U, V) \geq \gamma \cdot (1+\delta)^3 \cdot R(U) \tag{D.32}$$

$$= \gamma \cdot (1+\delta)^3 \cdot R(V) \tag{D.33}$$

Combining all the above cases, i.e., Eqs. D.18, D.22, D.27, and D.31, we have the following inequality:

$$\sum_{U \in \sigma_S} f(U, V) \geq \min\Big(\alpha,\ \frac{\alpha}{1+\delta},\ \beta,\ \frac{\beta}{1+\delta},\ \gamma \cdot (1+\delta)^3\Big) \cdot R(V) \tag{D.34}$$

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

$$U_1 \prec U_2 \prec \cdots \prec U_m, \tag{D.35}$$

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

$$R(U_{k+1}) \ > \ (1+\delta) \cdot R(U_k) \tag{D.36}$$

hence by the property of geometric series,

$$\sum_{k=1}^{n} R(U_k) \leq \frac{1+\delta}{\delta} \cdot R(U_n) \tag{D.37}$$

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:

$$\sum_{U \in \sigma_S} R(U) \leq \frac{1+\delta}{\delta} \sum_{I \in \sigma_S^c} R(I) \tag{D.38}$$

$$= \frac{1+\delta}{\delta} \cdot \mathsf{Goodput}(\sigma_S) \tag{D.39}$$

Amortized charging bound. With Eq. D.38, we have:

$$\mathsf{Goodput}(\sigma_S) \geq \frac{\delta}{1+\delta} \cdot \sum_{U \in \sigma_S} R(U) \tag{D.40}$$

$$= \frac{\delta}{1+\delta} \cdot \sum_{U \in \sigma_S} \sum_{V \in \sigma_*} f(U,V) \tag{D.41}$$

$$= \frac{\delta}{1+\delta} \cdot \sum_{V \in \sigma_*} \sum_{U \in \sigma_S} f(U,V) \tag{D.42}$$

Combining with the per-request bound Eq. D.34, we have:

$$\mathsf{Goodput}(\sigma_S) \geq \frac{\delta}{1+\delta} \cdot \mathfrak{A}(\delta,\alpha,\beta,\gamma) \cdot \sum_{V \in \sigma_*} R(V) \tag{D.43}$$

$$= \frac{\delta}{1+\delta} \cdot \mathfrak{A}(\delta,\alpha,\beta,\gamma) \cdot \mathsf{Goodput}(\sigma_*) \tag{D.44}$$

where

$$\mathfrak{A}(\delta,\alpha,\beta,\gamma) \;=\; \min\Big(\alpha,\, \frac{\alpha}{1+\delta},\, \beta,\, \frac{\beta}{1+\delta},\, \gamma \cdot (1+\delta)^3\Big) \tag{D.45}$$

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

$$\mathfrak{B}(\delta,\alpha,\beta,\gamma) \;=\; \frac{\delta}{1+\delta} \cdot \min\Big(\frac{\alpha}{1+\delta}, \frac{\beta}{1+\delta},\, \gamma \cdot (1+\delta)^3\Big) \tag{D.46}$$

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:

$$\max_{\delta,\alpha,\beta,\gamma} \quad \mathfrak{B}(\delta,\alpha,\beta,\gamma) \tag{D.47}$$

$$\text{s.t.} \quad \alpha + \beta + \gamma \leq 1 \tag{D.48}$$

$$\alpha \geq 0,\, \beta \geq 0, \gamma \geq 0 \tag{D.49}$$

$$\delta > 0 \tag{D.50}$$

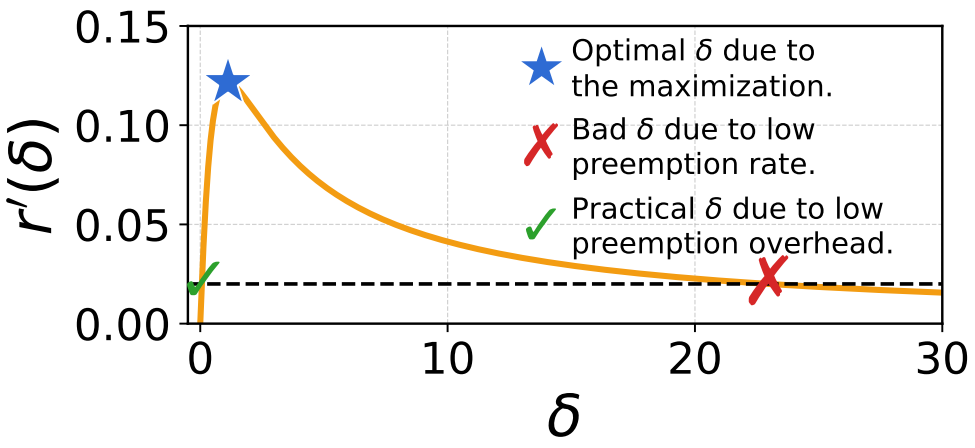


Figure D.1: Competitive ratio $r'(\delta)$ versus preemption threshold δ

Solving these above optimization problem using *numerical analysis* [87] yields an optimal performance guarantee of Concord $r'(\delta) \approx \frac{1}{8.13}$. Note that as we discussed in Section 4, preemption can improve the performance of our online algorithm Concord, but it also introduces practical overhead. Figure D.1 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 Concord. 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.

QED.

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

$$\frac{\mathsf{Goodput}(Concord)}{\mathsf{Goodput}(\mathrm{OPT})} \geq r \tag{D.51}$$

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

$$\mathcal{T} := \{W : \ R(W) \geq p \cdot R(r(b))\}. \tag{D.52}$$

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

$$R(U') \ \geq \ p\, R(r(b)) \ \geq \ p\, R(U). \tag{D.53}$$

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. D.46 from Lemma D.1 and Eq. D.53, we now have

$$\mathsf{Goodput}(\sigma_S) \geq \frac{p \cdot \delta}{1+\delta} \cdot \mathfrak{B}(\delta, \alpha, \beta, \gamma) \cdot \mathsf{Goodput}(\sigma_*) \tag{D.54}$$

Solving these above optimization problem using *numerical analysis* [87] under our experimental setting yields an optimal performance guarantee of Concord $r(\delta) \approx \frac{1}{8.557}$.

QED.