Online Scheduling for LLM Inference with KV Cache Constraints

Patrick Jaillet* Jiashuo Jiang† Konstantina Mellou‡ Marco Molinaro §
Chara Podimata¶ Zijie Zhou‖

January 16, 2026

arXiv:2502.07115v5 [cs.LG] 15 Jan 2026

Abstract

Large Language Model (LLM) inference, where a trained model generates text one word at a time in response to user prompts, is a computationally intensive process requiring efficient scheduling to optimize latency and resource utilization. A key challenge in LLM inference is the management of the Key-Value (KV) cache, which reduces redundant computations but introduces memory constraints. In this work, we model LLM inference with KV cache constraints theoretically and propose a novel batching and scheduling algorithm that minimizes inference latency while effectively managing the KV cache's memory.

More specifically, we make the following contributions. First, to evaluate the performance of online algorithms for scheduling in LLM inference, we introduce a hindsight optimal benchmark, formulated as an integer program that computes the minimum total inference latency under full future information. Second, we prove that no deterministic online algorithm can achieve a constant competitive ratio when the arrival process is arbitrary. Third, motivated by the computational intractability of solving the integer program at scale, we propose a polynomial-time online scheduling algorithm and show that under certain conditions it can achieve a *constant* competitive ratio. We also demonstrate our algorithm's strong empirical performance by comparing it to the hindsight optimal in a synthetic dataset. Finally, we conduct empirical evaluations on a real-world public LLM inference dataset, simulating the Llama2-70B model on A100 GPUs, and show that our algorithm significantly outperforms the benchmark algorithms. Overall, our results offer a path toward more sustainable and cost-effective LLM deployment.

Keywords: online optimization, LLM inference, scheduling, competitive ratio

1 Introduction

Large Language Models (LLMs) [Brown et al., 2020, Chowdhery et al., 2023, OpenAI, 2023, Kaplan et al., 2020, Wei et al., 2022] represent a significant advancement in AI, enabling machines to generate human-like text across various languages and contexts. Trained on vast datasets, these models are becoming critical for applications such as chatbots [Anthropic, 2023, Character, 2021, OpenAI, 2019, 2023], search engines [Microsoft, 2023, Google, 2023, Komo, 2023,

*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Email: jaillet@mit.edu.

†HKUST. Email: jsjiang@ust.hk.

‡Microsoft Research. Email: kmellou@microsoft.com.

§Microsoft Research. Email: mmolinaro@microsoft.com.

¶Sloan School of Management, Massachusetts Institute of Technology. Email: podimata@mit.edu.

‖Operations Research Center, Massachusetts Institute of Technology. Email: zhou98@mit.edu. Part of this work was done during a summer internship at Microsoft Research – Redmond.

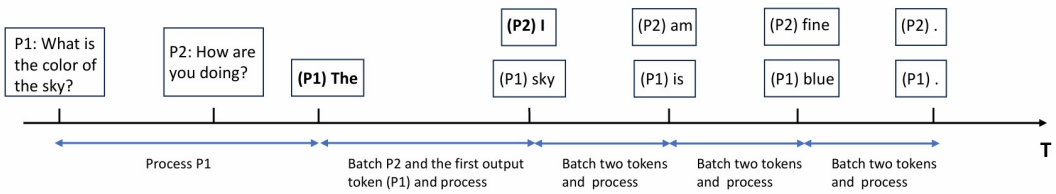

Figure 1: Example of online batching and scheduling.

Perplexity, 2022, You.com, 2020], code assistants [Amazon, 2023, GitHub, 2021, Replit, 2018], and healthcare services [Cascella et al., 2023, Peng et al., 2023, Sallam, 2023].

LLM Inference and the KV Cache. LLMs pose substantial computational challenges, particularly during the inference process where inputs are processed to generate responses. In LLM inference, a *"prompt"* is the input text provided to initiate a model's response generation. These prompts are broken down into smaller units called *"tokens"*, which may consist of words, sub-words, or punctuation marks based on the model's vocabulary. For instance, the prompt "What color is the sky?" can be tokenized into six units: "What," "color," "is," "the," "sky," and "?". Similarly, a response like "The color is blue." would be divided into five tokens: "The," "color," "is," "blue," and ".". When a prompt request is processed, typically it is not answered all-at-once; instead, it requires multiples rounds of processing to generate the tokens of the answer sequentially; in the previous example, the output token "blue" can only be generated after the preceding one "is" is produced by the model.

Each token is associated with two vectors: the **Key (K)**, which represents the token's significance to other tokens based on its relevance, and the **Value (V)**, which stores information that is used in the output if the token is deemed relevant. These KV pairs are computed based *only* on the token content and its absolute position, and once computed, they remain fixed for the entire process.

During inference, the Transformer attention mechanism [Vaswani et al., 2017] uses the stored keys and values to determine how tokens relate to one another. Without optimization, generating each new token would require recalculating the attention scores over all previously seen tokens, leading to a quadratic increase in computation as the sequence grows. To avoid this, modern LLMs use a *KV cache*, which stores all previously computed keys and values. This allows the model to reuse past computations efficiently, reducing the cost of generating each token to linear in the sequence length. However, the KV cache causes memory usage to grow linearly with the number of tokens, and unmanaged growth can result in the GPU running out of memory [Hooper et al., 2024, Kang et al., 2024].

Put together, there are *three* unique challenges posed by LLM inference: **(i)** each token can only be generated after its predecessor; **(ii)** the memory usage grows linearly during inference; and **(iii)** given the scale of real-world LLM inference, decisions must be made within milliseconds, thus making linear or mixed-integer programming solutions unusable.

Batching and Scheduling. When multiple prompt requests are in the queue, batching requests together (rather than handling them one-by-one) and scheduling their processing improves GPU efficiency. For example, Figure 1 illustrates the online batching and scheduling process for two distinct prompts, P1 ("What color is the sky?") and P2 ("How are you doing?"), during LLM inference on a single GPU. Initially, P1 is processed within its own batch. When P2 arrives, it must wait, as simultaneous processing of prompts is limited by worker avail-

ability. Once P1 is processed and generates its first output token, "The," the worker batches this token from P1 and the prompt P2 together and processes them together. After P2 produces its first output token, "I", and P1 produces the next token "sky", both tokens, "sky" and "I", are then batched together for efficient token processing, facilitating the subsequent generation of "is" for P1 and "am" for P2.

Our focus. Our focus in this work is to provide *scheduling* algorithms for LLM inference. Scheduling for LLM inference differs from classical scheduling problems primarily because of the bottlenecks introduced by the KV cache as we articulated above (i.e., the linear memory usage growth, and the KV cache's dynamic behavior). These challenges were also outlined in the recent survey of Mitzenmacher and Shahout [2025]. We also include further explanations of the challenges of scheduling for LLM inference (compared to standard scheduling problems) in Appendix A.2.

Importance. Beyond its theoretical importance, optimizing the scheduling policy in LLM inference is crucial in practice for three reasons. First, it can lead to reduced operational costs; the average daily cost of LLM inference for platforms like ChatGPT is approximately $700,000 [GilPress, 2024, Sun, 2023]. Second, it can enhance user satisfaction by minimizing response times. Last but not least, more efficient scheduling promotes sustainability, as LLM inference currently uses vast amounts of electricity and water daily [Gordon, 2024]. For example, ChatGPT's daily electricity usage exceeds half a million kilowatt-hours (equivalent to the energy consumption of nearly 180,000 U.S. households) and a single conversation uses approximately fifty centiliters of water, akin to the volume of a standard plastic bottle. By optimizing the scheduling policy, we can reduce the number of GPUs required, conserving resources and contributing to sustainability efforts. In fact, designing environmentally-friendlier LLM inference is an active area of research in systems and engineering, see e.g., [Li et al., 2025b].

1.1 Results Roadmap

In this work, we make the following contributions.

Mathematical model for online batching and scheduling in LLM inference. While many studies have focused on improving LLM inference efficiency via engineering [Agrawal et al., 2023, Kwon et al., 2023, Patel et al., 2023, Pope et al., 2023, Sheng et al., 2023, Yu et al., 2022, Zhong et al., 2024], there are very few formal models in this space. To address this gap, we propose a model (Section 2) for optimizing the batching and scheduling policy in LLM inference. *Batching* entails selecting which requests to process concurrently, while *scheduling* determines their timing.

Hindsight optimal benchmark via Integer Programming (IP). In Section 3, we introduce a hindsight optimal benchmark that assumes complete knowledge of future request arrivals and output lengths. We formulate this as an IP that computes the globally optimal scheduling policy minimizing total end-to-end latency under GPU memory constraints. Our IP captures operational constraints, including non-preemptive execution, per-token memory growth, and memory limits. This optimization benchmark serves as a gold standard for evaluating the quality of online scheduling algorithms.

Online batching and scheduling algorithm. In Section 4, we propose a practical online batching and scheduling algorithm: Memory Constrained Shortest First (`MC-SF`). The algorithm prioritizes partially completed requests to reduce latency, and then selects additional waiting requests to fill each batch by maximizing batch size while respecting KV cache memory

constraints throughout token generation. We characterize the feasibility of our algorithm by introducing constraints that anticipate future KV cache memory usage, ensuring that the algorithm only schedules batches that remain within memory limits throughout their execution. While we prove that no deterministic online algorithm can in general have bounded *competitive ratio*, we show that under some assumptions in the structure of the arrivals, our proposed algorithm has *constant* competitive ratio, providing theoretical underpinning for its practical performance. For our theoretical analysis, we assume that the algorithm has access to (relatively reliable) predictions of the output length of the prompt response; see e.g., [Zheng et al., 2024] for practical implementation of obtaining such predictions.

Synthetic-data experiments. In Section 5.1, we conduct numerical experiments on synthetically generated instances to evaluate the performance of our proposed `MC-SF` relative to the hindsight optimal. We design two arrival models to disentangle the sources of performance loss: one where all requests arrive at time zero (eliminating information asymmetry), and one with online stochastic arrivals (reflecting real-world uncertainty). In the first model, `MC-SF` has nearly optimal performance, with an average latency ratio of 1.005 and *exact* optimality in 114 out of 200 instances. In the second model, where future arrivals are unknown, the average ratio remains competitive at 1.047.

Experiments on public conversation dataset. In Section 5.2, we perform numerical simulations using the conversation dataset from Zheng et al. [2023], collected from over 210,000 unique IP addresses via the Vicuna demo and Chatbot Arena website. We evaluate our algorithm against benchmark parametrized algorithms with six parameter configurations in both high- and low-demand settings. In both the high- and low- demand settings, our algorithm significantly outperforms the benchmark algorithms. These gains can be translated to reduced energy consumption and lower costs, supporting more sustainable and efficient LLM deployment. Our experimental results in the synthetic and real-world datasets validate the performance of our algorithm beyond the assumptions that we placed in the theoretical part of the work.

1.2 Related Work

Our work is primarily related to two streams of literature; online scheduling and LLM inference. Both streams include plethora of papers on variations of the main problem and it is virtually impossible to survey them all in this paper. We include the most relevant and/or representative papers below. Section 6 includes some related works pertaining to potential future directions stemming from our model.

Online Scheduling. In online scheduling, a decision-maker needs to decide the optimal timing for job processing as jobs arrive sequentially. There is a large literature on this subject, where different objectives and input models have been studied; see the books/surveys Albers [2003, 2009], Roughgarden [2020], Leung [2004]. A particularly relevant set of studies are those that extend beyond processing individual jobs, where jobs of the same type can be grouped into batches and processed simultaneously [Lucier et al., 2013, Im and Moseley, 2013, Liu and Lu, 2015, Li et al., 2020]. Another relevant line of work considers precedence constraints, either within parts of the requests or between different requests Garg et al. [2019], Azar and Epstein [2002], Robert and Schabanel [2008], Agrawal et al. [2016]. Yet, the unique demands of LLM inference (in particular the growing memory usage in the KV cache and the combination of both batching and the dynamics of sequential token generation) limit the applicability of the existing algorithms.

LLM Inference from an Engineering Perspective. LLM inference is a developing field

with numerous engineering-oriented studies emerging within systems research. For instance, in scheduling, Patel et al. [2023], Zhong et al. [2024] proposed to use separate GPUs for processing only the prompt and the token phases of a request. Regarding batching, works like Yu et al. [2022], Agrawal et al. [2023, 2024b] examine methods for statically or dynamically grouping pending requests for batch execution. Liu et al. [2024] boosts LLM inference efficiency by introducing multi-head latent attention, which reduces the KV cache through low-rank key-value joint compression. Zhu et al. [2023] proposed approximate caching and dynamic choice of model size to accelerate LLM inference. Another problem in modern LLM inference is head-of-line blocking, where long-running jobs delay shorter ones, thus increasing the average and tail latency. Wu et al. [2023] tackle this by assigning priority queues for the prompts based on their input length and (re)assigning prompts to different queues as time goes by. Although all of the aforementioned papers boast significant performance gains practically, they are do not come with formal theory bounds. As such, they may be prone to pathological instances. Our approach in this work has been to center the need for theoretical advancements in scheduling for LLM inference.

LLM Inference from a Mathematical Perspective. Recently, there has been a flurry of works from the operations research and optimization community on the theoretical underpinning of scheduling for LLM inference. Similarly to our work, Bari et al. [2025] model LLM serving as an online batching/scheduling problem that must coordinate prompt inputs and responses all while respecting inference-system constraints. Unlike our KV-centric model, they use a more GPU-execution–driven iteration model. Ao et al. [2025] model optimization for LLM inference as a *multi-stage* optimization problem. Li et al. [2025a] study a model for scheduling in LLM inference centered on throughput/maximal stability, with service times driven by an empirically motivated (piecewise-linear) function of total tokens per batch rather than KV-memory feasibility being the primary constraint as is the case in our paper. Finally, there has also been some recent work on caching for LLM inference [Zhang et al., 2025].

2 Model

We study a batching and scheduling problem within a discrete time horizon for a single computational worker (GPU). The worker has a memory constraint $M > 0$.[1] Let $\mathcal{I}$ denote the instance consisting of unprocessed prompt requests arriving over the discrete time horizon. Each request $i \in \mathcal{I}$ has an associated size s_i, representing the number of tokens in the prompt, and response length o_i, indicating the number of tokens in its response.

Request Processing. Each request is processed online and undergoes two primary phases:

1. *Prompt Phase:* The prompt is processed in order to generate the initial output token. During this phase, the memory required for request i is s_i, accounting for the storage of key and value matrix for all tokens in the prompt within the KV cache.

2. *Token Phase:* Subsequent tokens are produced sequentially. In this phase, the memory needed to process the jth token of request i ($j \in \{1, 2, \ldots, o_i\}$) is $s_i + j$. This increment accounts for each new token's key and value, which adds 1 to the existing KV cache memory. Consequently, the peak memory usage for processing the final token of request i reaches $s_i + o_i$. After the completion of the last token, the KV cache clears all related memory usage $s_i + o_i$ for that request.

[1] M depends on the memory of the GPU and the complexity of the large language model in use.

Batch Processing. A batch may include any unprocessed prompt or output token of different requests; when a prompt request is processed in a batch, its first output token is generated, and, similarly, processing an output token results in the generation of the subsequent token upon batch completion. We assume each batch's processing time is one unit of time, and only one batch can be processed at a time. Moreover, we assume that this process is non-preemptive, meaning that once a prompt request i is added to a batch, it must be processed continuously for o_i periods until its final output token is processed. The memory constraint ensures that for all ongoing requests (those not fully processed or pending final output tokens), the total memory usage at any given moment does not exceed M, i.e., if $S^{(t)}$ is the set of ongoing requests at time t and $o_i^{(t)}$ is the index of the output token of such request i, then it holds that $\sum_{i\in S^{(t)}}(s_i + o_i^{(t)}) \leq M$. The scheduler's task is to decide when (i.e., in which batch) to start processing each incoming request.

Prompt Arrival Process. In this paper, we consider an online arrival model in which unprocessed prompt requests are assigned to the scheduler sequentially over time. An instance $\mathcal{I}$ consists of prompt requests that arrive one by one, where each request i is associated with an arrival time a_i; the arrival time a_i, as well as the request sizes (s_i, o_i) are revealed only at the moment when request i arrives. At any time t, the decision-maker has complete knowledge of (a_i, s_i, o_i) for all requests i such that $a_i \leq t$, and no information about requests with $a_i > t$, which have not yet arrived. In fact, our proposed algorithm works with only a *prediction* $\tilde{o}_i \geq o_i$ of the true output sizes; see Section 4 for more details. We remark that methods for high-accuracy output-size prediction are a very active area of research (e.g., [Jin et al., 2023, Hu et al., 2024, Cheng et al., 2024, Qiu et al., 2024b,a, Fu et al., 2024, Shahout et al., 2025]); see also Mitzenmacher and Shahout [2025] for a survey and open problems from a queuing-theory and optimization perspective. For example, Zheng et al. [2024] present a method with prediction accuracy up to 80%.

Evaluation Metrics. We evaluate an algorithm $\mathcal{A}$'s performance through its *end-to-end latency*. For each request i, its end-to-end latency is computed as $c_i(\mathcal{A}) - a_i$, where $c_i(\mathcal{A})$ is the time the last output token for request i is processed and a_i is the time that request i arrives. We use $\mathrm{TEL}(\mathcal{I}; \mathcal{A}) := \sum_{i\in[n]} c_i(\mathcal{A}) - a_i$ to denote the total end-to-end latency of algorithm $\mathcal{A}$ for a request sequence $\mathcal{I}$.

3 Hindsight Optimal Benchmark and Integer Programming

To evaluate the performance of an online scheduling algorithm, we introduce a natural benchmark known as the *hindsight optimal*. This benchmark represents an idealized scheduling policy that has complete foresight, i.e., it knows all future request arrivals a_i's and their corresponding prompt and output lengths s_i and o_i at the start of the time horizon. Although such information is not available to any real-world system, the hindsight optimal serves as a gold standard: it represents the best possible performance that any algorithm could achieve given perfect future knowledge.

Define $\bar{T}$ as an upper bound on the time when all jobs are completed; for instance, we can take $\bar{T} = \sum_{i\in[n]}(a_i + o_i)$. We formulate an Integer Program that computes the minimum possible

total end-to-end latency achievable by any scheduling policy:

$$\min \sum_{i\in[n]} \left(\sum_{t=\{a_i,\dots,\bar{T}\}} t \cdot x_{i,t} + o_i - a_i \right) \tag{1}$$

$$\text{s.t.} \sum_{t=\{a_i,\dots,\bar{T}\}} x_{i,t} = 1, \qquad \forall i \in [n] \tag{2}$$

$$\sum_{i=1}^{n} \sum_{k=\max\{a_i,t-o_i\}}^{t-1} (s_i + t - k) x_{i,k} \le M, \qquad \forall t \in [\bar{T}] \tag{3}$$

$$x_{i,t} \in \{0,1\}, \qquad \forall i \in [n], \forall t \in [\bar{T}] \tag{4}$$

In this formulation, the only decision variable is $x_{i,t} \in \{0,1\}$, which indicates whether request i begins processing at time t. Since the system operates under a non-preemptive scheduling regime, each request i must be processed without interruption for o_i consecutive rounds once it starts. The objective function (1) minimizes the total end-to-end latency over all requests. For each request i, the latency is defined as the time of its last token completion minus its arrival time, i.e., $(t+o_i)-a_i$, if the request starts processing at time t. This is equivalent to minimizing $\sum_{t=a_i}^{\bar{T}} t \cdot x_{i,t} + o_i - a_i$ across all i, where $x_{i,t}$ determines the start time.

Constraint (2) ensures that each request is scheduled exactly once after its arrival. Constraint (3) enforces the GPU memory limit M at each time step t, by summing over the memory usage of all requests that are active at that time. Specifically, a request i, if scheduled to start at time k, will be active at time t for $k+1 \le t \le k+o_i$, which is equivalent to $k \in [t-o_i, t-1]$. Moreover, since i cannot be scheduled before its arrival time a_i, we have $k \in [\max\{a_i, t-o_i\}, t-1]$. If request i starts at time k, then it contributes to memory usage at time $t \in [k, k+o_i]$ with an amount equal to $s_i + t - k$, reflecting both the prompt memory s_i and the token-wise KV cache growth over time. Lastly, constraint (4) enforces the binary nature of the scheduling decisions.

This integer program provides an exact characterization of the optimal non-preemptive scheduling policy under complete future information. It jointly captures the timing, memory, and latency structure of the system while remaining compact and interpretable. As such, it serves as a hindsight benchmark for evaluating the performance of online algorithms under both stochastic and adversarial arrival models.

4 Efficient Batching and Scheduling Algorithm and Theoretical Results

In this section, we consider the online arrival model described in Section 2, under an adversarial setting where the number of arrivals, the arrival time and size of each prompt, and the output lengths are all chosen by an adversary.

We start by establishing a hardness result for this general online problem. The standard metric for evaluating the performance of an online algorithm is the *competitive ratio*: an algorithm $\mathcal{A}$ has *competitive ratio* α if for every instance $\mathcal{I}$, the algorithm's latency $\text{TEL}(\mathcal{I};\mathcal{A})$ is at most α times that of the hindsight-optimal solution $\text{OPT}(\mathcal{I})$.

We show that unfortunately no deterministic online algorithm can achieve a competitive ratio better than order $\sqrt{n}$, i.e., in the worst-case, the gap between any algorithm and the optimal

solution needs to grow with the number of requests. The proof of Theorem 4.1 can be found in Appendix B.

Theorem 4.1. *Every deterministic algorithm has a competitive ratio at least* $\Omega(\sqrt{n})$.

Despite this impossibility result, we propose the scheduling algorithm `MC-SF` that has demonstrable effectiveness across a wide range of arrival instances, via both theoretical analysis (below) and numerical experiments (Section 5.1).

Algorithm `MC-SF`. As discussed in Section 2, upon the arrival of request i, we observe its input length s_t and also a prediction $\tilde{o}_i$ that overestimates the true output length o_i, i.e., $\tilde{o}_i \geq o_i$. This ensures the algorithm can (over)estimate memory consumption and create feasible batches.

At each round t, let $R^{(t)}$ represent the set of all requests that have not yet been processed, while $S^{(t)}$ denotes the set of requests that are currently in progress but not yet completed (i.e., some output tokens have been generated, but not all of them). Our algorithm prioritizes processing requests in $S^{(t)}$ first. After processing all the requests currently in $S^{(t)}$, there may still be unused memory in the KV cache, so our algorithm chooses a subset of requests, $U^{(t)} \subset R^{(t)}$ to add to the batch in order to maximize memory utilization and minimize total latency. To be more precise, our algorithm tries to process as many requests as possible within each batch; for that, it aims to maximize the number of requests in $U^{(t)}$, provided that they satisfy memory constraints.

Specifically, for a subset $U \subset R^{(t)}$, let $t_{\max}(U) := \max_{i \in U}\{t + \tilde{o}_i\}$ represent the maximum predicted completion time for all requests in U if they are added to the batch at time t. To ensure U is feasible, the KV cache memory limit must not be exceeded at any $t' \in [t+1, t_{\max}(U)]$. This requires that:

$$\sum_{i \in S^{(t)}} (s_i + t' - p_i) \cdot \mathbb{1}_{\{\tilde{o}_i \geq t' - p_i\}} + \sum_{i \in U} (s_i + t' - t) \cdot \mathbb{1}_{\{\tilde{o}_i \geq t' - t\}} \leq M, \quad \forall t' \in [t+1, t_{\max}(U)] \tag{5}$$

where p_i is the starting time to process request i. The first sum accounts for predicted memory usage from ongoing requests in $S^{(t)}$, while the second captures new requests in U. As long as this inequality is satisfied for all $t' \in [t+1, t_{\max}(U)]$, U is feasible to add to the batch. Thus, our selection rule is the following: 1) We sort the set of waiting requests $R^{(t)}$ in non-decreasing order of predicted output lengths $\tilde{o}_i$'s, and 2) select the largest prefix that satisfies the allowed memory usage imposed by inequality (5):

$$U^{(t)} = \text{argmax}_{U \text{ prefix of } R^{(t)} \text{ sorted by } \tilde{o}_i\text{'s}} \left\{ |U| : \text{inequality (5) is satisfied } \forall t' \in [t+1, t_{\max}(U)] \right\} \tag{6}$$

We prioritize adding requests with smaller $\tilde{o}_i$ values, as we predict these requests to complete more quickly. Additionally, the predicted peak memory usage of each request is $s_i + \tilde{o}_i$. In many situations, prompts processed by a single worker tend to have similar input sizes s_i, with relatively low variance, while the output length exhibits greater variability due to the stochastic nature of response generation. As a result, $\tilde{o}_i$ plays a more critical role in determining memory usage, and selecting requests with smaller $\tilde{o}_i$ values typically reduces peak memory consumption, enabling more requests to be packed into each batch.

We continue adding requests in this order, checking the feasibility condition of inequality (5). Importantly, we only need to check this constraint at the predicted *completion times* of ongoing or new requests, specifically $p_j + \tilde{o}_j$ for $j \in S^{(t)} \cup U^{(t)}$. This is because (i) memory usage potentially peaks at these completion times, as a request's memory demand increases until it finishes, and (ii) since memory usage varies linearly between start and end times, satisfying

the constraint at these peak points ensures feasibility throughout the interval. The complete algorithm is detailed in Algorithm 1. Since this algorithm adds requests to each batch in a shortest-first manner while smartly checking memory constraints over the near future, we refer to it as Memory-Constrained Shortest-First (`MC-SF`).

Algorithm 1: Memory Constrained Shortest First (`MC-SF`)

for *each round* $t = 1$ *to* T **do**
  Let $S^{(t)}$ be the set of requests that have already started processing and $R^{(t)}$ be the remaining (waiting) requests at time t. Set $U^{(t)} = \emptyset$
  for *each request* $i \in R^{(t)}$ *in ascending order of predicted output length* $\tilde{o}_i$ **do**
    Set a list with the times $t' = p_j + \tilde{o}_j$ for each $j \in S^{(t)} \cup U^{(t)} \cup \{i\}$
    if *all inequalities in Equation* (5) *hold for all* t' *in this list* **then**
      Add request i to $U^{(t)}$
    else
      Break the **for** loop
  Process the requests in $S^{(t)} \cup U^{(t)}$

The following proposition states that the computational complexity of each round of `MC-SF` is actually independent of the number of requests, and the proof can be found in Appendix B.

Proposition 4.2. *Given that the memory limit of the KV cache is M, `MC-SF` has a computational complexity of $O(M^2)$ at each round $t \in [T]$.*

Next, we formally analyze the algorithm `MC-SF` for the special case in which all prompts have identical size $s_i = s$[2] and arrive simultaneously at time $t = 0$. This case zooms in on the performance of the algorithm for the important scenario where prompt size variation is small and demand is high, meaning many requests are typically waiting to be processed at any given time. Under this setting, we show that `MC-SF` achieves a constant competitive ratio, long as the predictions $\tilde{o}_i$ are also within a constant factor of the true output sizes o_i.

Theorem 4.3. *Consider instances where all requests arrive at time 0 and have the same prompt size. Assume the predicted output lengths satisfy $o_i \leq \tilde{o}_i \leq \alpha o_i$ for some constant $\alpha \geq 1$. Then the algorithm `MC-SF` is $O(1)$-competitive for such instances, as long as the memory available M is at least twice the maximum predicted memory occupation of a single request in the instance, i.e., $M \geq 2 \max_i(s_i + \tilde{o}_i)$.*

Proof of Theorem 4.3. In order to illustrate the key ideas of the analysis, we present here the proof for the case of exact output length predictions, i.e., when $\tilde{o}_i = o_i$ for all requests i. The proof in the presence of prediction errors is identical and just tracks how these errors percolate to the final bound, and details are presented in Appendix B.3. We have also not optimized the constants in the $O(1)$, and opted to prioritize simplicity of exposition.

Fix throughout an instance satisfying the assumption of the theorem. Without chance of confusion, we also use `MC-SF` to denote the total latency of this algorithm. Also recall that OPT is the latency of the optimal solution in hindsight.

For an output length size o, let n_o be the number of requests in the instance with this output length. Also, let $\text{vol}_o := s \cdot o + \frac{o \cdot (o+1)}{2}$ denote the volume of memory that a request of this output length o occupies; the first summand $(s \cdot o)$ is because the input tokens s have to stay in the memory until the end of the processing, and the second term is the memory occupied

[2] A follow-up paper [Wang et al., 2025] builds on our model and presents methods for dealing with variable input and output prompt lengths.

every time that one more token is process until the total o tokens (i.e., $\sum_{j\in[o]} j$ tokens in the memory). We will bound the total latency of MC-SF and OPT based on these quantities.

Upper bound on the total latency of MC-SF.

Lemma 4.4 (UB on MC-SF)**.** *The total latency incurred by the algorithm MC-SF is at most*

$$\frac{1536}{M}\sum_{o} n_o \cdot \sum_{o'\le o} n_{o'}\cdot vol_{o'} + 24\sum_{o} n_o \cdot o$$

To prove this lemma, we will group the possible output lengths $1,\dots,o_{\max}$ in powers of 2. More precisely, let U_ℓ for $\ell = 0,\dots,\lfloor \log o_{\max}\rfloor$ denote the set of requests that have output length in the interval $[2^\ell, 2^{\ell+1})$. Slightly abusing notation, let $\mathrm{vol}_\ell(I)$ be the total amount of memory that the requests U_ℓ occupy in MC-SF's schedule added up over all times in the interval I.

Lemma 4.5. *Consider one of the sets of requests U_ℓ, and let $\underline{o}_\ell := 2^\ell$ and $\bar{o}_\ell := 2^{\ell+1}-1$ denote the smallest and largest possible output length for requests in this set. Let $\underline{t}$ and $\bar{t}$ be the first and last time the algorithm MC-SF processes a request in U_ℓ. Then the distance between these times can be upper bounded as*

$$\bar{t} - \underline{t} \;\le\; \frac{192}{M}\cdot \sum_{o=\underline{o}_\ell}^{\bar{o}_\ell} n_o \cdot vol_o + 5\bar{o}_\ell.$$

Proof. (For the remainder of the proof we omit the subscript ℓ in $\underline{o}_\ell$ and $\bar{o}_\ell$.) To prove this, let us partition the interval $\{\underline{t},\dots,\bar{t}\}$ into disjoint subintervals $I_1, I_2, \dots, I_w$ of length $\bar{o}$ (where I_w is the only exceptional interval that can be smaller than $\bar{o}$). For an interval $I \subseteq [T]$, let $\mathrm{peak}(I)$ be the peak memory use by MC-SF during this interval. The next lemma essentially says that if the peak memory utilization of an interval I_{j+1} is large, then the total volume of requests in U_ℓ scheduled "around" that point has to also be large.

Claim 4.6 (Peak to volume)*.* Consider any 3 consecutive intervals I_j, I_{j+1}, I_{j+2} of length $\bar{o}$. Then $\mathrm{vol}_\ell(I_j \cup I_{j+1} \cup I_{j+2}) \ge \frac{1}{4}\,\mathrm{peak}(I_{j+1}) \cdot \frac{\mathrm{vol}_{\bar{o}}}{s+\bar{o}}$.

Proof of Claim 4.6. Let $\tilde{t} \in I_{j+1}$ be the time when the peak memory occupation $\mathrm{peak}(I_{j+1})$ happens. Since the interval I_{j+1} is strictly between times $\underline{t}$ and $\bar{t}$, by definition of the algorithm it only processes requests of U_ℓ (i.e., with output lengths between $2^\ell = \underline{o}$ and $2^{\ell+1}-1 = \bar{o}$ in I_{j+1}), and so only those contribute to the memory occupation at time $\tilde{t}$. If k of these requests contribute to this peak occupation, then each contributes at most $s+\bar{o}$ to it; thus, we have $k \ge \frac{\mathrm{peak}(I_{j+1})}{s+\bar{o}}$. Each such request is completely contained in the bigger interval $I_j \cup I_{j+1} \cup I_{j+2}$, each such request contributes with at least $\mathrm{vol}_{\underline{o}}$ to the memory volume $\mathrm{vol}_\ell(I_j \cup I_{j+1} \cup I_{j+2})$, which then gives $\mathrm{vol}_\ell(I_j \cup I_{j+1} \cup I_{j+2}) \ge k \cdot \mathrm{vol}_{\underline{o}}$. Finally, since $\underline{o} \ge \frac{1}{2}\bar{o}$, a quick calculation shows that $\mathrm{vol}_{\underline{o}} \ge \frac{1}{4}\mathrm{vol}_{\bar{o}}$. Combining these threes inequalities gives the claim. □

We now conclude the proof of Lemma 4.5. Suppose for contradiction that $\bar{t} - \underline{t} > \frac{192}{M}\sum_{o=\underline{o}}^{\bar{o}} n_o \cdot \mathrm{vol}_o + 5\bar{o}$. First, since $\bar{o} \le 2\underline{o}$, a quick calculation shows that $\mathrm{vol}_o \ge \frac{1}{4}\mathrm{vol}_{\bar{o}}$ for all o between $\underline{o}$ and $\bar{o}$. Then since $|U_\ell| = \sum_{o=\underline{o}}^{\bar{o}} n_o$, our assumption implies that $\bar{t} - \underline{t} > \frac{48}{M}\cdot |U_\ell| \cdot \mathrm{vol}_{\bar{o}} + 5\bar{o}$.

This further implies that w (the number of the intervals I_j of length $\bar{o}$ in this period) is at least

$$w \geq \left\lfloor \frac{\frac{48}{M} \cdot |U_\ell| \cdot \mathrm{vol}_{\bar{o}} + 5\bar{o}}{\bar{o}} \right\rfloor \geq \frac{48}{M} \cdot |U_\ell| \cdot \frac{\mathrm{vol}_{\bar{o}}}{\bar{o}} + 4.$$

Every such interval I_j other than I_w has peak memory utilization more than $M - (s + \bar{o})$, otherwise the algorithm would have scheduled one more request U_ℓ in it. Then applying the previous claim to the first $\lfloor \frac{w-1}{3} \rfloor$ groups of 3 consecutive intervals I_j's, we obtain that

$$\begin{aligned} \mathrm{vol}_\ell(\{\underline{t}, \ldots, \bar{t}\}) &\geq \left\lfloor \frac{w-1}{3} \right\rfloor \cdot \frac{1}{4} \cdot (M - (s + \bar{o})) \cdot \frac{\mathrm{vol}_{\bar{o}}}{s + \bar{o}} \\ &\geq \frac{4}{M} \cdot |U_\ell| \cdot \frac{\mathrm{vol}_{\bar{o}}}{\bar{o}} \cdot \frac{M}{2} \cdot \frac{\mathrm{vol}_{\bar{o}}}{s + \bar{o}} > |U_\ell| \cdot \mathrm{vol}_{\bar{o}}, \end{aligned} \tag{7}$$

where the second inequality uses the assumption M is twice as big as the maximum single request occupation, i.e., $s + \bar{o} \leq \frac{M}{2}$, and the last inequality uses $\mathrm{vol}_{\bar{o}} = s \cdot \bar{o} + \frac{\bar{o} \cdot (\bar{o}+1)}{2} > \frac{1}{2}\bar{o} \cdot (s + \bar{o})$.

However, each request U_ℓ contributes at most $\mathrm{vol}_{\bar{o}}$ to $\mathrm{vol}_\ell(\{\underline{t}, \ldots, \bar{t}\})$, and thus $\mathrm{vol}_\ell(\{\underline{t}, \ldots, \bar{t}\}) \leq |U_\ell| \cdot \mathrm{vol}_{\bar{o}}$; this contradicts Equation (7), and concludes the proof of the lemma. □

Proof of Lemma 4.4. Let t_ℓ be the first time a request in U_ℓ is (starting to be) processed by `MC-SF` (let $\ell_{\max} := \lfloor \log o_{\max} \rfloor$ and let $t_{\ell_{\max}+1}$ be the last time the algorithm is processing something). The latency for each request in U_ℓ is at most $t_{\ell+1} + \bar{o}_\ell$ (i.e., if the algorithm starts processing requests from the next group then is has already started to process all requests in U_ℓ, which take at most $+\bar{o}_\ell$ time to complete); thus, the total latency is at most $\sum_\ell |U_\ell| \cdot (t_{\ell+1} + \bar{o}_\ell)$. However, from the Lemma 4.5 we know that

$$t_{\ell+1} - t_\ell \;\leq\; \frac{192}{M} \cdot |U_\ell| \cdot \mathrm{vol}_{\bar{o}_\ell} + 5\bar{o}_\ell + 1 \;\leq\; \frac{192}{M} \cdot |U_\ell| \cdot \mathrm{vol}_{\bar{o}_\ell} + 6\bar{o}_\ell,$$

and so $t_{\ell+1} \leq \frac{192}{M} \sum_{\ell' \leq \ell} |U_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}} + 6 \sum_{\ell' \leq \ell} \bar{o}_{\ell'}$.

This gives that the total latency of `MC-SF` can be upper bounded as

$$\texttt{MC-SF} \;\leq\; \frac{192}{M} \underbrace{\sum_\ell |U_\ell| \sum_{\ell' \leq \ell} |U_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}}}_{A} + 6 \underbrace{\sum_\ell |U_\ell| \sum_{\ell' \leq \ell} \bar{o}_{\ell'}}_{B}. \tag{8}$$

Concluding the proof of the lemma requires just a bit of algebra to clean up the bound.

To upper bound the term A, let $O_\ell := \{\underline{o}_\ell, \ldots, \bar{o}_\ell\}$ be the possible output lengths of the requests in U_ℓ. We first observe that since $\bar{o}_\ell \leq 2\underline{o}_\ell$, we have $\mathrm{vol}_{\bar{o}} \leq 4\mathrm{vol}_o$ for every $o \in O_\ell$. Moreover, since $|U_\ell| = \sum_{o \in O_\ell} n_o$, we have

$$|U_\ell|^2 \cdot \mathrm{vol}_{\bar{o}_\ell} \leq 2 \Bigg(\sum_{o \in O_\ell} n_o \sum_{o' \in O_\ell, o' \leq o} n_{o'} \Bigg) \cdot \mathrm{vol}_{\bar{o}_\ell} \leq 8 \sum_{o \in O_\ell} n_o \sum_{o' \in O_\ell, o' \leq o} n_{o'} \cdot \mathrm{vol}_{o'}$$

and for $\ell' < \ell$

$$|U_\ell| \cdot |U_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}} \leq \Bigg(\sum_{o \in O_\ell} n_o \sum_{o' \in O_{\ell'}} n_{o'} \Bigg) \cdot \mathrm{vol}_{\bar{o}_{\ell'}} \leq 4 \sum_{o \in O_\ell} n_o \sum_{o' \in O_{\ell'}} n_{o'} \cdot \mathrm{vol}_{o'},$$

which combined give

$$|U_\ell| \sum_{\ell' \le \ell} |U_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}} \;\le\; 8 \sum_{o \in O_\ell} n_o \sum_{o' \le o} n_{o'} \cdot \mathrm{vol}_{o'},$$

and so adding up over all ℓ gives the upper bound $A \le 8 \sum_o n_o \sum_{o' \le o} n_{o'} \cdot \mathrm{vol}_{o'}$.

To upper bound the term B in (8), we observe that since the $\bar{o}_\ell$'s grow exponentially, $\sum_{\ell' \le \ell} \bar{o}_{\ell'} \le 2\bar{o}_\ell$. Then since $\bar{o}_\ell \le 2o$ for every $o \in O_\ell$, we have

$$B \le 2 \sum_\ell |U_\ell| \cdot \bar{o}_\ell = 2 \sum_\ell \sum_{o \in O_\ell} n_o \cdot \bar{o}_\ell \le 4 \sum_\ell \sum_{o \in O_\ell} n_o \cdot o = 4 \sum_o n_o \cdot o.$$

Plugging these upper bounds on A and B on (8), we get

$$\texttt{MC-SF} \le \frac{1536}{M} \sum_o n_o \sum_{o' \le o} n_{o'} \cdot \mathrm{vol}_{o'} + 24 \sum_o n_o \cdot o.$$

This concludes the proof of Lemma 4.4. □

Lower bound on the total latency of OPT.

Lemma 4.7. *We have the following lower bound on the total optimal latency OPT:*

$$OPT \ge \frac{1}{6M} \sum_o n_o \cdot \sum_{o' \le o} n_{o'} \cdot vol_{o'} + \frac{1}{6} \sum_o n_o \cdot o.$$

To lower bound the total latency of OPT we consider the following LP relaxation. Let U_o denote the set of requests with output length o. Let $\bar{a}_o^t$ be the number of requests in U_o that finish at time t in the optimal solution. The memory volume of all requests that finish up to time t need to fit in the total memory $t \cdot M$ available up to that time, and hence $\sum_{t' \le t} \sum_o \bar{a}_o^t \cdot \mathrm{vol}_o \le t \cdot M$. Moreover, $\sum_t \bar{a}_o^t = n_o$ (all requests in U_o finish at some time). Finally, the $\sum_o \bar{a}_o^t$ requests that finish at time t have latency (recall all requests are released at time 0) equal to t, and the optimal latency OPT is given by $\sum_t t \cdot \sum_o \bar{a}_o^t$. Together these observations show that OPT can be lower bounded by the following Linear Program with variables a_o^t, where in particular we relax the requirement that $\bar{a}_o^t$'s are integers:

$$\begin{aligned} \mathrm{OPT}_{LP} := \min & \sum_t t \cdot \sum_o a_o^t \\ \text{s.t.} & \sum_{t' \le t} \sum_o a_o^t \cdot \mathrm{vol}_o \le t \cdot M, \qquad \forall t \\ & \sum_t a_o^t = n_o, \qquad \forall o \\ & a_o^t \ge 0, \qquad \forall t, o. \end{aligned} \tag{9}$$

We then lower bound OPT_{LP}. Consider the optimal solution $\{a_o^{*t}\}_{t,o}$ for this LP. For a given output size o, let t_o^* be the first time t where $a_o^{*t} > 0$, i.e., where a request U_o is assigned to time t.

Indeed we can see the above LP as the problem of, for each o, (fractionally) assigning n_o units over the timesteps $0, 1, \ldots$ subject to the constraint that the timesteps have a limited receiving

capacity, and assigning a unit to time t incurs a cost of t. Since vol_o is increasing over o, we observe that the optimal solution to this LP is to first try to assign all requests with the smallest o (call it $o_{\min}$) to time 1; if the constraint $\sum_o a_o^1 \cdot \text{vol}_o \le M$ does not allow all $n_{o_{\min}}$ requests to be assigned to time 1, the remaining ones are assigned to time 2, and so on; otherwise we move to the $2nd$ smallest o and assign the requests U_o to time 1. This optimizes the above LP because it maximizes the number of items assigned to time 1, which has the smallest cost (the analogous argument holds for the other times). In summary, we have that the "first time" values t_o^* are non-decreasing, namely $t_o^* \le t_{o'}^*$ when $o < o'$.

Let $t^*_{o_{\max}+1}$ be the last time such that $\sum_o a_o^{*t} > 0$. Using this observation we will prove the following lower bound on the "first times" t_o^*.

Claim 4.8. For all o we have $t^*_{o+1} \ge \frac{1}{M} \sum_{o' \le o} n_{o'} \cdot \text{vol}_{o'}$.

Proof. By the observation above, in the optimal solution $\{a_t^{*o'}\}_{o',t}$ all items with output length at most o are assigned to times $\le t^*_{o+1}$, i.e., no later than when the next output length is assigned; thus, $\sum_{t' \le t^*_{o+1}} a_{o'}^{*t'} = n_{o'}$ for all $o' \le o$. Then considering (9) to time t^*_{o+1} we get

$$\sum_{o' \le o} n_{o'} \cdot \text{vol}_{o'} = \sum_{t' \le t^*_{o+1}} \sum_{o' \le o} a_{o'}^{*t'} \cdot \text{vol}_{o'} \le t^*_{o+1} \cdot M,$$

and rearranging we get the claim. □

We are now able to prove the lower bound on OPT from Lemma 4.7.

Proof of Lemma 4.7. By definition of t_o^*, we know that $a_o^{*t} = 0$ for all $t < t_o^*$, and so $\sum_t t \cdot a_o^{*t} \ge t_o^* \cdot \sum_{t \ge t_o^*} a_o^{*t} = t_o^* \cdot n_o$. Plugging the bound on t_o^* from the previous claim and adding over all o we can lower bound OPT_{LP} (and thus OPT) as

$$\begin{aligned}
\text{OPT} \ge \text{OPT}_{LP} &= \sum_t t \cdot \sum_o a_o^{*t} \\
&\ge \sum_o n_o \cdot t_o^* \\
&\ge \sum_o n_o \cdot \left(\frac{1}{M} \sum_{o' < o} n_{o'} \cdot \text{vol}_{o'} \right) \\
&= \frac{1}{M} \sum_o n_o \sum_{o' \le o} n_{o'} \cdot \text{vol}_{o'} - \frac{1}{M} \sum_o n_o^2 \cdot \text{vol}_o. \qquad (10)
\end{aligned}$$

To remove the negative term in the right-hand side (and add a new one, to match that of Lemma 4.4), we provide two other lower bounds on the original OPT (not the LP).

The first is that for any output length o, due to memory constraints, at most $\frac{n_o}{2}$ of them can be finished in the optimal schedule by time $\frac{n_o}{2} \frac{\text{vol}_o}{M}$, since the total memory available up to this time is $\frac{n_o}{2} \cdot \text{vol}_o$ and each such request consumes vol_o memory; thus, the total latency on the optimal solution for the request of output length o is at least $\frac{n_o}{2} \cdot \frac{n_o}{2} \frac{\text{vol}_o}{M}$. Adding this over all o, the total latency OPT has the lower bound

$$\text{OPT} \ge \frac{1}{4M} \sum_o n_o^2 \cdot \text{vol}_o. \qquad (11)$$

Finally, since each request of output length o takes $(o+1)$ units of processing/time to finish (and thus has latency at least o), we also have $\text{OPT} \geq \sum_o n_o \cdot o$ (which is at least $\sum_o n_o$).

Adding this bound plus 4 times (11) plus (10) we get

$$6\,\text{OPT} \geq \frac{1}{M} \sum_o n_o \sum_{o' \leq o} n_{o'} \cdot \text{vol}_{o'} + \sum_o n_o \cdot o. \tag{12}$$

This concludes the proof of Lemma 4.7. □

Combining the bound on `MC-SF` from Lemma 4.4 and the lower bound on OPT from Lemma 4.7, we obtain $\texttt{MC-SF} \leq O(1) \cdot \text{OPT}$, thus proving Theorem 4.3. □

5 Numerical Simulation

In this section, we conduct numerical simulations using both synthetic data and real-world traces to evaluate the performance of our proposed scheduling algorithm `MC-SF`.

In the first of these experiments (Section 5.1), the primary goal of is to assess the performance of `MC-SF` within the framework of the mathematical model introduced in Section 2, comparing it against the hindsight-optimal policy from Section 3. The second set of experiments (Section 5.2) evaluates the practical effectiveness of `MC-SF` using a large-scale, open-source dataset derived from real LLM inference traces. Due to the size and complexity of this dataset, it is computationally infeasible to compute the hindsight optimal-solution, and we instead compare `MC-SF` against several baseline scheduling policies that are representative of those used in real-world LLM inference systems. Additional experimental results are provided in Appendix C.

5.1 Synthetic Data Numerical Simulations

We first evaluate the performance of `MC-SF` on synthetic datasets. These experiments aim to quantify the two primary sources of performance gap between the algorithm and the hindsight optimal: (i) the information asymmetry between online and offline algorithms, and (ii) the inherent suboptimality of the algorithm itself. To partially disentangle these effects, we consider two different arrival models, which we describe below. To obtain the hindsight-optimal solution, we solve the integer programming model from Section 3 using the Gurobi solver. Since solving the Integer Program is computationally expensive, we limit this comparison to small-scale synthetic instances to meet reasonable memory and time limits.

Experimental Setup. For each experiment, the memory capacity M is drawn uniformly from the integers between 30 and 50. For each request, the prompt size s_i is drawn uniformly from the integers between 1 and 5, and the output length o_i from the integers between 1 and $M - s_i$. We run 200 independent trials under each of the following arrival models:

1. *Arrival Model 1 (All-at-once arrivals).* All n requests arrive at time $t = 0$, where n is a random integer between 40 and 60. This model allows both `MC-SF` and the hindsight optimal to access the full set of requests at the beginning of execution, eliminating any information gap. Therefore, any observed performance difference is solely due to the algorithm's structural approximation.

2. *Arrival Model 2 (Online stochastic arrivals).* Requests arrive over a discrete time horizon $[1, T]$, where T is drawn uniformly from the integers between 40 and 60. The arrival process

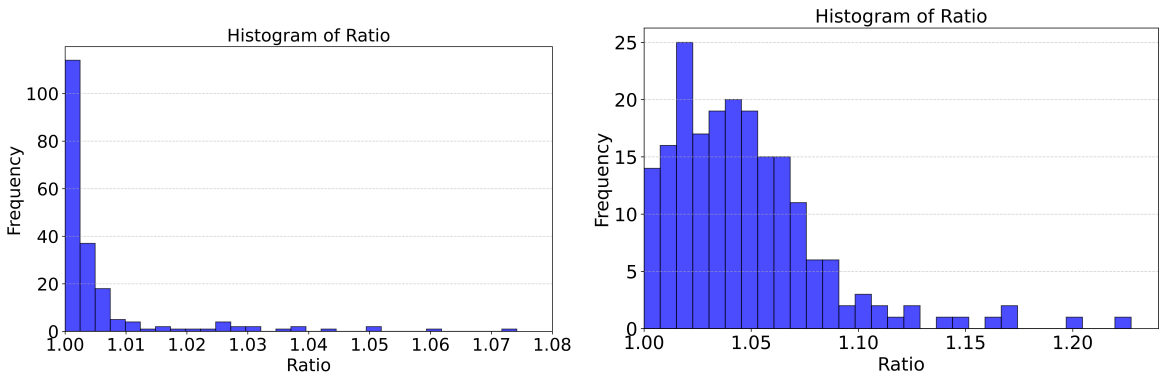

Figure 2: Histogram of latency ratio: `MC-SF` vs Hindsight Optimal. **Left:** Arrival Model 1. **Right:** Arrival Model 2.

follows a stationary Poisson distribution with rate $\lambda \in [0.5, 1.5]$, sampled uniformly. This now introduces *online uncertainty*, which the algorithm must respond to without knowing future arrivals.

Performance Metric. We evaluate performance using the ratio between the total end-to-end latency incurred by `MC-SF` and that of the hindsight-optimal policy. A ratio of 1 indicates that the algorithm achieves the optimal schedule, while larger values reflect a performance gap.

Results for Arrival Model 1. Across 200 trials, the best-case ratio is 1.000 (meaning the algorithm produces the optimal schedule), the worst-case ratio is 1.074, while the average ratio is 1.005. The left figure of Figure 2 displays the distribution of latency ratio of `MC-SF` vs. Hindsight Optimal under Arrival Model 1. These results show that in this setting, `MC-SF` performs almost optimally. In 114 of the 200 trials, the algorithm exactly matches the optimal latency, demonstrating that its design closely approximates the structure of the hindsight-optimal policy. Since in this first arrival model there is no information gap between `MC-SF` and hindsight-optimum, both having full knowledge of the input, these ratios close to 1 indicate that our proposed algorithm has a very small structural suboptimality.

Results for Arrival Model 2. Across 200 trials, here the best-case ratio is 1.000, the worst-case ratio is 1.227, and the average ratio is 1.047. The right figure of Figure 2 displays the distribution of latency ratio of `MC-SF` vs. Hindsight Optimal under Arrival Model 2. Since the requests are no longer known a priori by the algorithm, the performance ratio captures both the effect of information asymmetry and the algorithm's structural approximation error. The resulting ratios are still quite close to 1, indicating strong performance of `MC-SF` and establishing it as an effective and practically viable scheduling policy. The difference of the ratios between the two models can be partially attributed to the cost of online decision-making.

5.2 Real Data Numerical Experiments

Dataset Overview. We use a conversational dataset[3] by Zheng et al. [2023] from over 210,000 distinct IP addresses via the Vicuna demo and Chatbot Arena platforms. To manage its size, we selected a random subset of 10,000 conversations for analysis. Each conversation includes a user-generated question and a response from an LLM. We define the question as the prompt input and each word in the response as an output token. Figure 7 in Appendix C shows the

[3]Publicly available at `https://huggingface.co/datasets/lmsys/lmsys-chat-1m`.

distribution of the sizes (word count) of prompts (mean: 40.62 and median: 11) and output tokens (mean: 85.32 and median: 45).

Simulation Setup. In our simulation, we operate over a continuous time horizon, with 10,000 prompts arriving according to a stationary Poisson process. Let λ denote the arrival rate per second, where we consider two cases: **Case 1: High Demand** $\lambda = 50$ and **Case 2: Low Demand** $\lambda = 10$. In both scenarios, we simulate the performance of the Llama2-70B model on two linked A100 GPUs (operating collectively as a single worker) and consider the memory limit $M = 16492$; additional details can be found in Appendix C.

At a high level, the simulation works as follows: prompts chosen from the conversational dataset arrive continuously via the chosen Poisson arrival process. To calculate the processing time of a batch created by an algorithm, we use an LLM inference simulator from Agrawal et al. [2024a].[4] We then report the average end-to-end latency, i.e., the total end-to-end latency divided by the number of requests.

Benchmark Algorithms. Currently, most LLM inference research is built on vLLM, a widely recognized high-performance inference engine designed to enhance the efficiency of Large Language Model (LLM) inference [Kwon et al., 2023]. Since LLM inference is a relatively new field, scheduling algorithms in both academia and industry primarily rely on those implemented in vLLM. The scheduling algorithm in vLLM follows a first-come, first-served (FCFS) strategy: when the machine is idle, it prioritizes requests based on their arrival time. However, instead of maximizing the number of requests in a batch, vLLM uses a predefined threshold for memory occupation. Once the KV cache occupancy exceeds this threshold, no additional requests are added to the batch. This motivates us to define the following benchmarks:

α-protection greedy algorithms. We begin with a class of parameterized algorithms called α-protection greedy scheduling algorithms, where $\alpha \in (0, 1)$. These algorithms maintain a protection memory threshold of αM, serving as a safeguard against memory overflow. When forming each batch, the algorithm gives priority to existing token jobs. For new prompt requests, it checks if adding a new prompt i with initial memory $s_i + 1$ would cause the memory usage to exceed $(1 - \alpha)M$. If the memory limit is exceeded, no further prompts are added to the batch. In the event that the KV cache memory overflows during the processing of the requests, the α-protection greedy scheduling algorithms will clear all active requests sending them back to the waiting queue as unprocessed.

α-protection, β-clearing algorithms. α-protection greedy scheduling algorithms can cause unnecessary evictions; to address this, we define a new class of benchmark algorithms: α-protection β-clearing algorithms. These algorithms follow the same principles as α-protection greedy scheduling but, when the KV cache memory limit is exceeded, each active request is cleared and sent back to the scheduler with an independent probability β.

Memory Constrained Benchmark (`MC-Benchmark`). Beyond the benchmarks discussed above, we introduce an additional benchmark algorithm, denoted as `MC-Benchmark`, which is partially inspired by the vLLM scheduling policy and partially by the memory feasibility checks used in `MC-SF`. Following the vLLM structure, `MC-Benchmark` forms batches by processing requests in ascending order of their arrival times. While constructing a batch, it decides whether to include each request based on a prospective memory check used in `MC-SF`: it only adds a request if the future memory usage, including KV cache growth, remains within the memory limit M. The detailed pseudocode for `MC-Benchmark` is provided in Appendix C (Algorithm 2).

[4] Publicly available at `https://github.com/microsoft/vidur/`.

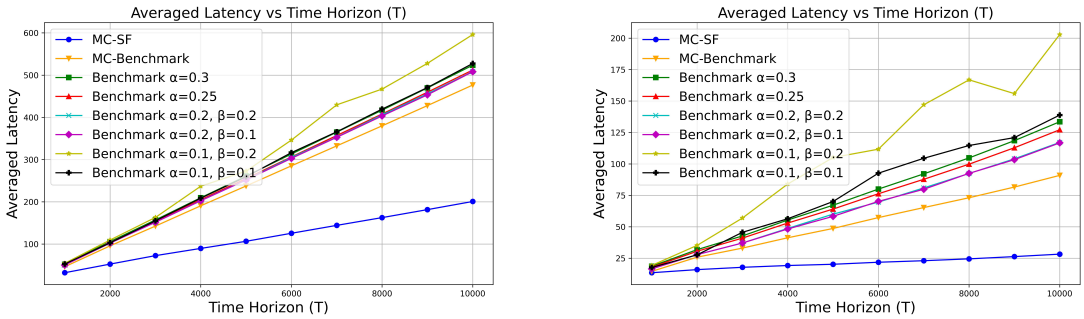

Figure 3: Average End-to-End Latency Across Scheduling Algorithms. **Left:** High Demand. **Right:** Low Demand.

5.2.1 Results: High vs. Low Demand

We evaluate the performance of `MC-SF` and baseline algorithms under two demand regimes: high demand ($\lambda = 50$) and low demand ($\lambda = 10$).

First, we examine memory usage over time. Figures 8 and 11 show that `MC-SF` consistently stays within the memory limit M across both settings. Despite batch processing durations varying in practice, the memory check in Equation (6) reliably prevents overflow by ensuring feasibility over future memory usage. Under low demand, memory usage remains close to full utilization, suggesting a stable system state.

We also investigate the performance of the α-protection and α-protection–β-clearing heuristics. We observe that for very small protection levels α, the α-protection heuristic may lead to repeated evictions and infinite processing loops, therefore we perform a grid search with step size 0.01 to identify the smallest possible α: $\alpha = 0.21$ in the high-demand setting and $\alpha = 0.24$ in the low-demand setting. We then evaluate latency across six configurations, including $\alpha = 0.3$, $\alpha = 0.25$, and four combinations of $\alpha \in \{0.2, 0.1\}$, $\beta \in \{0.2, 0.1\}$.

Figure 3 presents the average end-to-end latency across all algorithms for request volumes $\{1000, 2000, \ldots, 10000\}$. In the high-demand case (left), average latency increases linearly for all algorithms, indicating overload. Notably, `MC-SF` has a slope of approximately $1/6$, compared to $1/2$ for the best-performing benchmark, highlighting its superior scalability. In the low-demand case (right), `MC-SF` maintains a much lower latency growth rate—about $1/800$, which is over eight times smaller than the best benchmark slope ($\sim 1/100$).

Although our paper primarily targets latency minimization, we also report throughput as a secondary performance indicator. Figure 4 plots the instantaneous per-second throughput achieved by `MC-SF` and `MC-Benchmark` for the first 1000 arriving requests. The light green bars show the per-second arrival workload, measured as the total number of tokens introduced in each second (input+output per request). Under this overloaded regime, `MC-SF` has higher processing throughput than `MC-Benchmark` for most time intervals, indicating that its latency improvements do not come at the expense of reduced service rate.

These results confirm that `MC-SF` remains memory-safe and highly efficient across both regimes. Additional results are also provided in Appendix C.

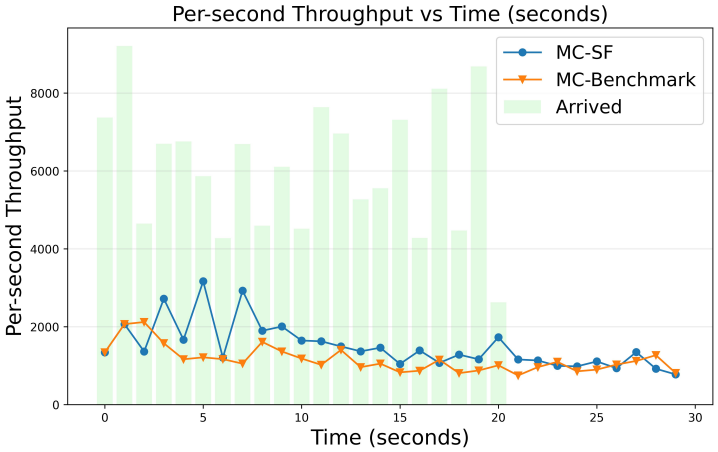

Figure 4: Per-second Throughput Across Scheduling Algorithms.

5.2.2 Performance under Prediction Errors

In the previous experiments, we assumed that the scheduler has access to the true output length o_i for each request i (taken from the dataset). In practice, output length can be predicted and will inevitably contain error. This subsection evaluates the robustness of `MC-SF` when only noisy length predictions are available.

Prediction Model. We replace the true output length o_i with a random prediction $\hat{o}_i$ generated as

$$\hat{o}_i \sim \text{Uniform}\big((1-\epsilon)o_i,\ (1+\epsilon)o_i\big),$$

where $\epsilon \in \{0.2, 0.5, 0.8\}$ controls the prediction error. The simulation setup (dataset, arrival process, inference-time estimation via Vidur, and latency metric) is identical to Section 5.2, except that `MC-SF` now uses $\hat{o}_i$ instead of o_i when performing memory-feasibility checks and building batches.

Risk of Overflow and Memory Protection. Because `MC-SF` is designed to aggressively utilize available KV-cache memory, an underestimate $\hat{o}_i < o_i$ can cause the realized KV-cache growth to exceed the physical limit M. In our simulator, such an overflow triggers a clearing event, where all active requests are evicted and re-queued, increasing latency and potentially causing repeated retries. To mitigate this underestimation risk, we introduce a simple protection margin mentioned above: we reserve a fixed fraction αM of memory and run `MC-SF` as if the effective budget were $(1-\alpha)M$. Concretely, we set $\alpha = 0.1$ and apply the same selection logic as before, but with M replaced by $(1-\alpha)M$ in the feasibility check.

Results. Figure 5 reports the average end-to-end latency under different prediction error levels. As expected, larger prediction error leads to higher latency, since the scheduler must operate with noisier length estimates and a more conservative effective memory budget. Importantly, the $\alpha = 0.1$ protection margin prevents the instability caused by systematic underestimation and substantially reduces the frequency and impact of clearing events. Even at $\epsilon = 0.8$, `MC-SF` with protection achieves significantly lower latency than the benchmark FCFS policy, highlighting that `MC-SF` remains effective under substantial prediction noise.

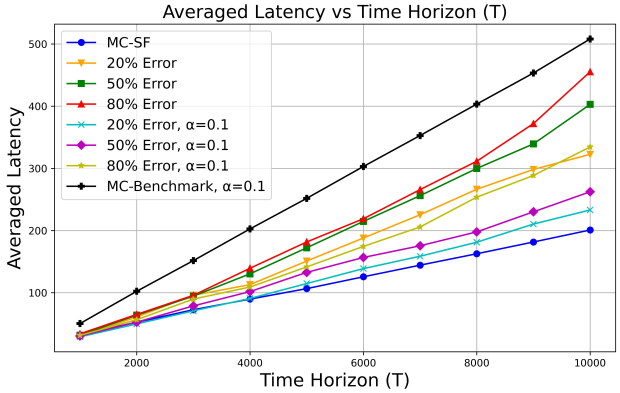

Figure 5: Average End-to-End Latency Across Scheduling Algorithms Under Prediction Error.

6 Discussion

In this paper, we developed a formal model for online batching and scheduling in LLM inference that explicitly captures the dynamic memory growth induced by the KV cache. We introduced a hindsight-optimal benchmark via an integer programming formulation, established fundamental limits by showing that no deterministic online algorithm admits a constant competitive ratio under adversarial arrivals, and proposed a practical polynomial-time algorithm, `MC-SF`, that achieves *constant* competitive ratio under some structured conditions on the prompts that arrive. Through both synthetic experiments benchmarked against the hindsight optimal and large-scale simulations using real LLM inference traces, we demonstrated that `MC-SF` achieves near-optimal latency and substantial improvements over existing scheduling heuristics while remaining memory-safe. Together, these results provide an end-to-end theoretical and empirical framework for understanding and designing scheduling policies for KV-cache-constrained LLM inference.

There are several research directions that stem from our work.

Our model in Section 2 assumes that upon arrival, each request is accompanied by a prediction $\tilde{o}_i$ that upper bounds its true output length o_i. An interesting future direction is to *jointly* design prediction mechanisms for output lengths and batching–scheduling policies that operate simultaneously during inference. Ideally, such predictors would provide not only upper bounds on o_i, but also informative lower bounds, enabling tighter control of memory usage and more aggressive batching decisions. Our proposed algorithm and analysis can serve as a benchmark for this setting, characterizing the best achievable competitive performance when output lengths are either known exactly or upper bounded by a prediction algorithm. In fact, a recent follow-up work to our paper [Chen et al., 2025] has taken one step closer to this direction by extending the model that we present in this work to a model where output lengths are predicted to be in a known interval. Finally, note that the literature on traditional scheduling has also provided online algorithms that only have "predictions" about the remaining time of jobs when they are being scheduled [Azar et al., 2021, 2022, Gupta et al., 2026].

Another important direction is to extend our framework to settings with *multiple* computational workers operating in parallel. While the availability of multiple GPUs substantially enlarges the design space of batching and scheduling algorithms, it also introduces new challenges that are absent in the single-worker setting. For example, workers may be heterogeneous in terms

of memory capacity or compute throughput, requiring "matching" decisions between requests and workers to maximize the overall efficiency of the entire system. Moreover, decisions must now balance load across workers while accounting for the evolving KV-cache footprint of each active request; see e.g., the recent work of Balseiro et al. [2025] on load balancing for LLM inference and the work of Jaiswal et al. [2025] for a more applied point-of-view on the subject. For settings where a prompt needs to be processed by multiple GPUs, it will be interesting to see how the literature on multi-server job scheduling (e.g., [Grosof et al., 2022]) can provide insights for scheduling for LLM inference.

Finally, an important direction is to study other arrival models, even in the single-worker setting, where most requests are generated by an unknown stochastic process but a small fraction of requests are extreme outliers, being either unusually large or unusually small. Such heavy-tailed or mixed workloads are commonly observed in real LLM inference systems and pose challenges that are not well captured by purely adversarial or fully stochastic models. Our empirical results already suggest that algorithms such as `MC-SF` remain effective under this type of workload heterogeneity. Moreover, our current theoretical analysis can serve as a benchmark for evaluating the performance of future algorithms designed for these hybrid stochastic–adversarial settings.

References

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient LLM inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for LLM inference. *Proceedings of Machine Learning and Systems*, 6:351–366, 2024a.

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. *arXiv preprint arXiv:2403.02310*, 2024b.

Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallel dag jobs online to minimize average flow time. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, page 176–189, USA, 2016. Society for Industrial and Applied Mathematics. ISBN 9781611974331.

Susanne Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2):3–26, 2003. doi: 10.1007/S10107-003-0436-0. URL `https://doi.org/10.1007/s10107-003-0436-0`.

Susanne Albers. Online scheduling. In *Introduction to scheduling*, pages 71–98. CRC Press, 2009.

Amazon. Amazon CodeWhisperer. `https://aws.amazon.com/codewhisperer/`, 2023.

Anthropic. Claude. `https://claude.ai`, 2023.

Ruicheng Ao, Gan Luo, David Simchi-Levi, and Xinshang Wang. Optimizing llm inference: Fluid-guided online scheduling with memory constraints. *arXiv preprint arXiv:2504.11320*, 2025.

Yossi Azar and Leah Epstein. On-line scheduling with precedence constraints. *Discrete Applied Mathematics*, 119(1):169–180, 2002. ISSN 0166-218X. doi: https://doi.org/10.1016/S0166-218X(01)00272-4. URL `https://www.sciencedirect.com/science/article/pii/`

S0166218X01002724. Special Issue devoted to Foundation of Heuristics in Combinatoria l Optimization.

Yossi Azar, Stefano Leonardi, and Noam Touitou. Flow time scheduling with uncertain processing time. In *proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1070–1080, 2021.

Yossi Azar, Stefano Leonardi, and Noam Touitou. Distortion-oblivious algorithms for minimizing flow time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 252–274. SIAM, 2022.

Santiago R Balseiro, Vahab S Mirrokni, and Bartek Wydrowski. Load balancing with network latencies via distributed gradient descent. *arXiv preprint arXiv:2504.10693*, 2025.

Agrim Bari, Parikshit Hegde, and Gustavo de Veciana. Optimal scheduling algorithms for llm inference: Theory and practice. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(3):1–43, 2025.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Marco Cascella, Jonathan Montomoli, Valentina Bellini, and Elena Bignami. Evaluating the feasibility of ChatGPT in healthcare: an analysis of multiple clinical and research scenarios. *Journal of medical systems*, 47(1):33, 2023.

Character. Character ai. `https://character.ai`, 2021.

Zixi Chen, Yinyu Ye, and Zijie Zhou. Adaptively robust llm inference optimization under prediction uncertainty. *arXiv preprint arXiv:2508.14544*, 2025.

Ke Cheng, Wen Hu, Zhi Wang, Peng Du, Jianguo Li, and Sheng Zhang. Enabling efficient batch serving for lmaas via generation length prediction. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 853–864. IEEE, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24 (240):1–113, 2023.

Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank. *Advances in Neural Information Processing Systems*, 37: 59006–59029, 2024.

Naveen Garg, Anupam Gupta, Amit Kumar, and Sahil Singla. Non-clairvoyant precedence constrained scheduling. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 63:1–63:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/LIPICS.ICALP.2019.63. URL `https://doi.org/10.4230/LIPIcs.ICALP.2019.63`.

GilPress. ChatGPT users statistics. WhatsTheBigData, 2024. URL `https://whatsthebigdata.com/chatgpt-users/#google_vignette`.

GitHub. GitHub copilot. `https://github.com/features/copilot`, 2021.

Google. Bard. https://bard.google.com, 2023.

Cindy Gordon. ChatGPT and Generative AI innovations are creating sustainability havoc. Forbes, 2024. URL https://www.forbes.com/sites/cindygordon/2024/03/12/chatgpt-and-generative-ai-innovations-are-creating-sustainability-havoc/.

Isaac Grosof, Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. Optimal scheduling in the multiserver-job model under heavy traffic. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(3):1–32, 2022.

Anupam Gupta, Amit Kumar, Debmalya Panigrahi, and Zhaozi Wang. An optimal online algorithm for robust flow time scheduling. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1214–1238. SIAM, 2026.

Öncü Hazır and Safia Kedad-Sidhoum. Batch sizing and just-in-time scheduling with common due date. *Annals of Operations Research*, 213:187–202, 2014.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length LLM inference with KV cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

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

Sungjin Im and Benjamin Moseley. Online batch scheduling for flow objectives. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 102–104, 2013.

Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St Amant, Chetan Bansal, Victor Ruhle, Anoop Kulkarni, et al. Sageserve: Optimizing llm serving on cloud data centers with forecast aware auto-scaling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(3):1–24, 2025.

Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. S3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. Gear: An efficient KV cache compression recipe for near-lossless generative inference of LLM. *arXiv preprint arXiv:2403.05527*, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Ali Husseinzadeh Kashan and Behrooz Karimi. Scheduling a single batch-processing machine with arbitrary job sizes and incompatible job families: an ant colony framework. *Journal of the Operational Research Society*, 59(9):1269–1280, 2008.

Komo. Komo AI. https://komo.ai/, 2023.

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

Joseph Y.-T. Leung, editor. *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004. ISBN 978-1-58488-397-5. URL `http://www.crcnetbase.com/isbn/978-1-58488-397-5`.

Wenhua Li, Libo Wang, Xing Chai, and Hang Yuan. Online batch scheduling of simple linear deteriorating jobs with incompatible families. *Mathematics*, 8(2):170, 2020.

Yueying Li, Jim Dai, and Tianyi Peng. Throughput-optimal scheduling algorithms for llm inference and ai agents. *arXiv preprint arXiv:2504.07347*, 2025a.

Yueying Li, Zhanqiu Hu, Esha Choukse, Rodrigo Fonseca, G Edward Suh, and Udit Gupta. Ecoserve: Designing carbon-aware ai inference systems. *arXiv preprint arXiv:2502.05043*, 2025b.

Pierre Lienhart. LLM inference series: 2. The two-phase process behind LLMs' responses. `https://medium.com/@plienhar/`, 2023.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

Peihai Liu and Xiwen Lu. Online unbounded batch scheduling on parallel machines with delivery times. *Journal of Combinatorial Optimization*, 29:228–236, 2015.

Brendan Lucier, Ishai Menache, Joseph Naor, and Jonathan Yaniv. Efficient online scheduling for deadline-sensitive jobs. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 305–314, 2013.

Microsoft. Bing AI. `https://www.bing.com/chat`, 2023.

Michael Mitzenmacher and Rana Shahout. Queueing, predictions, and large language models: Challenges and open problems. *Stochastic Systems*, 15(3):195–219, 2025.

OpenAI. ChatGPT. `https://chat.openai.com`, 2019.

OpenAI. GPT-4 technical report. arxiv 2303.08774. *View in Article*, 2(5), 2023.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. *Power*, 400(700W):1–75, 2023.

Cheng Peng, Xi Yang, Aokun Chen, Kaleb E Smith, Nima PourNejatian, Anthony B Costa, Cheryl Martin, Mona G Flores, Ying Zhang, Tanja Magoc, et al. A study of generative large language model for medical research and healthcare. *NPJ digital medicine*, 6(1):210, 2023.

Perplexity. Perplexity AI. `https://www.perplexity.ai/`, 2022.

Michael L Pinedo. *Scheduling*, volume 29. Springer, 2012.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Chris N Potts and Mikhail Y Kovalyov. Scheduling with batching: A review. *European journal of operational research*, 120(2):228–249, 2000.

Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Power-aware deep learn-

ing model serving with {μ-Serve}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 75–93, 2024a.

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

Replit. Replit ghostwriter. `https://replit.com/site/ghostwriter`, 2018.

Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 491–500. SIAM, 2008. URL `http://dl.acm.org/citation.cfm?id=1347082.1347136`.

Tim Roughgarden, editor. *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020. ISBN 9781108637435. doi: 10.1017/9781108637435. URL `https://doi.org/10.1017/9781108637435`.

Malik Sallam. The utility of ChatGPT as an example of large language models in healthcare education, research and practice: Systematic review on the future perspectives and potential limitations. *MedRxiv*, pages 2023–02, 2023.

Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. Don't stop me now: Embedding based scheduling for LLMS. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL `https://openreview.net/forum?id=7JhGdZvW4T`.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

Yan Sun. The economics of large language models. Substack, 2023. URL `https://sunyan.substack.com/p/the-economics-of-large-language-models`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Meixuan Wang, Yinyu Ye, and Zijie Zhou. Llm serving optimization with variable prefill and decode lengths. *arXiv preprint arXiv:2508.06133*, 2025.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

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

Fan Yang, Morteza Davari, Wenchao Wei, Ben Hermans, and Roel Leus. Scheduling a single parallel-batching machine with non-identical job sizes and incompatible job families. *European Journal of Operational Research*, 303(2):602–615, 2022.

You.com. You.com. https://you.com/, 2020.

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

Wenxin Zhang, Yueying Li, Ciamac C Moallemi, and Tianyi Peng. Tail-optimized caching for llm inference. *arXiv preprint arXiv:2510.15152*, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, et al. LMSYS-Chat-1M: A large-scale real-world LLM conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An LLM-empowered LLM inference pipeline. *Advances in Neural Information Processing Systems*, 36, 2024.

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

Banghua Zhu, Ying Sheng, Lianmin Zheng, Clark Barrett, Michael I Jordan, and Jiantao Jiao. On optimal caching and model multiplexing for large model inference. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 59062–59094, 2023.

Supplementary Material for paper *"Online Scheduling for LLM Inference with KV Cache Constraints"*

A Background: Online Optimization in LLM Inference

This section provides background in LLM inference using a single computational worker (i.e., a single GPU).

A.1 LLM Inference Process on a Single Request

We first demonstrate how a single GPU worker processes an incoming prompt, using the example prompt "What color is the sky?" from Lienhart [2023]. The workflow is illustrated in Figure 6.

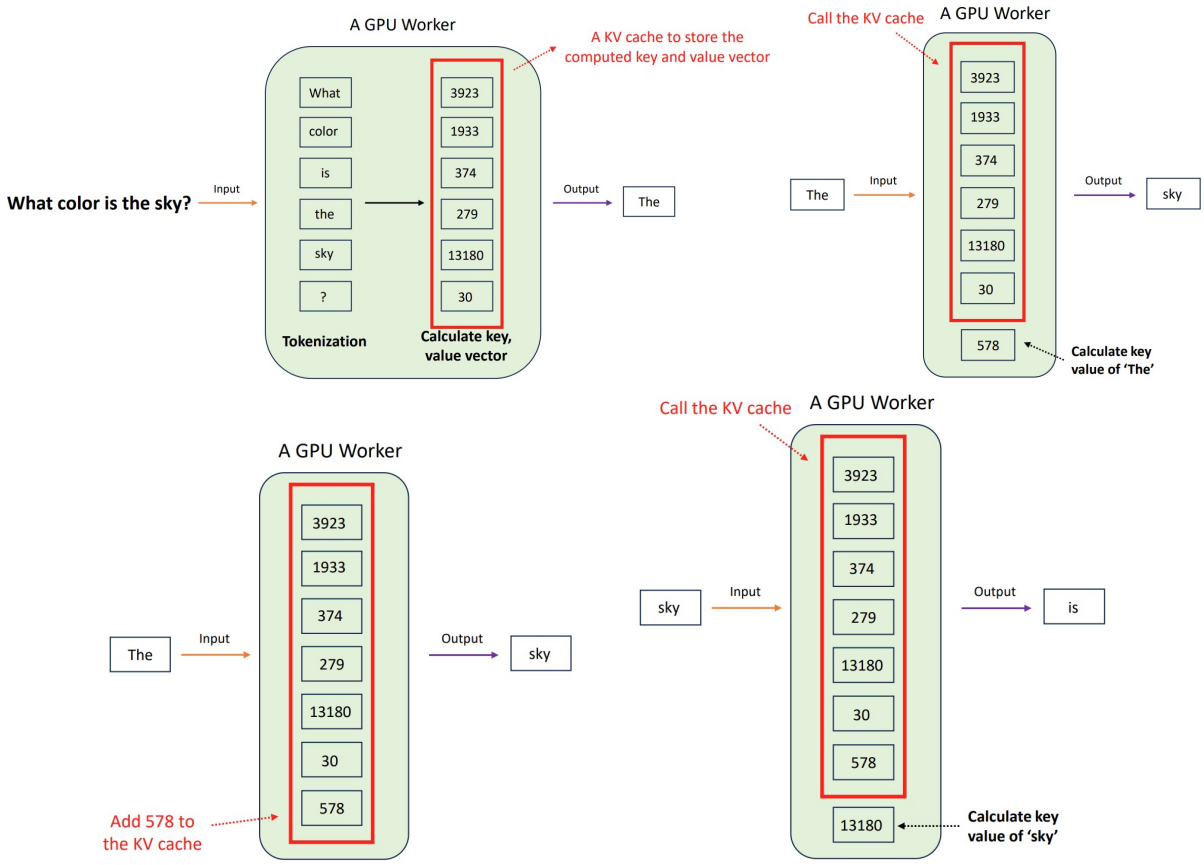

Figure 6: An example of LLM inference [Lienhart, 2023].

Upon receiving the prompt, as shown in the top left figure, the worker begins by tokenizing the prompt and generates key and value vectors. Then, it uses these vectors to calculate the attention scores, which indicate the relevance of each token, enabling the generation of the first output token, "The." To avoid recalculating these vectors with each new token, the worker saves them in a key-value (KV) cache.

Next, as depicted in the top right figure, after generating "The," the worker processes this token

by calculating its key and value and retrieving the key-value vectors of the initial prompt tokens from the KV cache. Using this combined data, the model generates the next token, "sky."

The bottom left figure shows that during "sky" generation, the worker calculates the key and value for "The" and adds them to the KV cache to avoid recalculations. Finally, as illustrated in the bottom right figure, the worker processes "sky," continuing this cycle until all output tokens are generated.

Finally, we note that when a prompt request arrives, the computational worker does not know the length of its response (i.e., the number of tokens it will generate). However, certain techniques can predict the output length of each prompt before processing begins. For instance, Zheng et al. [2024] proposes a methodology that achieves an output length prediction accuracy of up to 81%.

A.2 Illustration of Novel Difficult Aspects of LLM Inference Scheduling

Though scheduling problems have been studied extensively in the previous literature on computing resource allocation (e.g., Pinedo [2012]), we illustrate some of the new dynamics of LLM inference that separates it from standard scheduling problems. To be specific, due to the autoregressive LLM inference process described in Section A.1, the memory size required by a request keeps growing during the inference procedure, which is fundamentally different from the previous scheduling models where the size (or the resource occupation) of a job is fixed.

To see this point, suppose that we are following the principle of processing the shortest job first and there are two prompts P1 and P2, whose sizes eventually grow to t_1 and t_2. If $t_1 + t_2 > M$, where M is the memory limit of the KV cache, then the two prompts cannot be processed at the same time in the classical scheduling model. However, this may not be the case for our LLM inference model since the sizes are varying over time. Suppose that the initial size for both prompts is the same, denoted by s, and it satisfies that $s \leq t_1 \leq M/2$. The size of both jobs increases by 1 at each round until reaching t_1 and t_2. Then, we can process P1 and P2 at the same time until the time $t_1 - s$ when the size of both jobs increases to t_1; note that we can process both P1 and P2 concurrently, since the memory requirement for the two jobs together is upper bounded $2t_1 \leq M$. After time $t_1 - s$, though the size of P2 may be larger than t_1, prompt P1 has finished processing (since we have processed $s + t_1 - s = t_1$ tokens), and the required memory can be released to further process P2. Therefore, P1 and P2 can be processed at the same time in the LLM inference model.

Moreover, the batching problem in our setting significantly differs from traditional problems (e.g., Potts and Kovalyov [2000]). When assuming that the KV cache memory limit is the sole constraint for the batch size, the number of jobs in a batch during each round varies depending on the specific jobs included in the batch. This is because the memory usage for each output token increases linearly over time. While some studies (Kashan and Karimi [2008], Hazır and Kedad-Sidhoum [2014], Yang et al. [2022]) consider batch size constraints as the total weight of jobs—where weight corresponds to the memory usage of each prompt or token—in our problem, the situation is more complex. Processing a prompt or token increases the memory usage. Consequently, the effective weight of jobs at any given time depends on the starting time to process this token, and the index of this token in the request. This weight dependency also makes the problem more challenging.

B Appendix for Section 4

B.1 Proof of Theorem 4.1

Proof of Theorem 4.1. Fix a deterministic algorithm $\mathcal{A}$. For any $M \geq 1$, consider the following instance $\mathcal{I}$ with available memory M and with $s_i = 1$ for all request i. First, a request is released at time 0 with output length $o_1 = M - 1$ ("long request"). Let b be the time $\mathcal{A}$ starts processing this first request. Then $\frac{M}{2}$ requests are released at time $r := b + M - \frac{\sqrt{M}}{2}$ with output size $o_i = 1$ ("short requests"). Let $n := \frac{M}{2} + 1$ denote the number of requests in this instance. We claim that the total latency of $\mathcal{A}$ on this instance is at least $\Omega(\sqrt{n}) \cdot \mathrm{OPT}(\mathcal{I})$.

For that, we first claim that

$$\mathrm{OPT}(\mathcal{I}) \leq 3.5M. \tag{13}$$

If $r \geq M$, then it is possible to start processing the long request at time 0 and finish by time $M-1$ (incurring latency $M-1$) and then at time $r \geq M$ start processing all the short requests (since the maximum memory occupation of each short request is 2, they can all be executed together), finishing them at time $r+1$ (incurring latency 1 for each of these requests); in this case we have $\mathrm{OPT}(\mathcal{I}) \leq M - 1 + \frac{M}{2} \leq 3.5M$ as desired.

If $r < M$, then a possible offline solution is to first do all the short requests and then the long request, namely start executing all the short requests at time r, finishing them at time $r+1$ (incurring latency 1 for each), and then start to process the long request at time $r+2$, finishing it at time $r + 2 + (M-1)$ (incurring latency $r + 2 + (M-1)$). Therefore, $\mathrm{OPT}(\mathcal{I}) \leq \frac{M}{2} + r + 1 + M \leq 2.5M + 1 \leq 3.5M$, where the second inequality uses the fact we are in the case $r < M$ and the last inequality uses $M \geq 1$. Thus, in both cases we have that (13) holds.

We now lower bound the latency that $\mathcal{A}$ experiences over $\mathcal{I}$. In all times between time $r = b + (M-1) - \frac{\sqrt{M}}{2}$ and $b + (M-1)$, the long request occupies memory at least $M - \frac{\sqrt{M}}{2}$; thus, the total memory available for other requests across all those times is at most $\frac{\sqrt{M}}{2} \cdot \frac{\sqrt{M}}{2} = \frac{M}{4}$. Thus, at least $\frac{M}{2} - \frac{M}{4} = \frac{M}{4}$ of the short requests can only be started by the algorithm $\mathcal{A}$ on or after time $b + M$, and thus each of them incurs latency at least $b + M - r = \frac{\sqrt{M}}{2}$. Thus, the total latency of $\mathcal{A}$ is at least $\frac{M}{4} \cdot \frac{\sqrt{M}}{2}$.

Combining this with the upper bound on OPT from (13), we get

$$\frac{\mathrm{TEL}(\mathcal{I}; \mathcal{A})}{\mathrm{OPT}(\mathcal{I})} \geq \frac{\sqrt{M}}{28}.$$

Since the number of items in the instance is $n = 1 + \frac{M}{2}$, this shows that the competitive ratio of $\mathcal{A}$ cannot be better than $\Omega(\sqrt{n})$, thus proving the theorem. □

B.2 Proof of Proposition 4.2

Proof of Proposition 4.2. At each time step $t \in [T]$, to solve Eq. (6), we add requests in ascending order of their indices, stopping the process once any inequality is violated. The memory limit M is a constant, which implies that we will add at most $M/(\max_{i \in [n]} s_i + 1) = O(M)$ requests in $U^{(t)}$. For each request i, to decide whether to add it or not in $U^{(t)}$, one has to check all inequalities at the completion times $p_j + o_j$ for $j \in S^{(t)} \cup U^{(t)}$ in Eq. (5). The number of inequalities we need to check is $O(M)$ since $|S^{(t)} \cup U^{(t)}| = O(M)$. Therefore, the feasibility check for each request i has a complexity of $O(M)$. Since we can add at most $O(M)$ number of requests to a batch, the complexity at time t is at most $O(M^2)$. □

B.3 Proof of Theorem 4.3 with Inaccurate Output Length Predictions

Recall that in Section 4 we proved that the algorithm `MC-SF` is $O(1)$-competitive when the output length predictions $\tilde{o}_i$ are exact, i.e., $\tilde{o}_i o_i$ for all i. Now we show the required modifications in the proof to accommodate approximate predictions, namely when $o_i \le \tilde{o}_i \le \alpha o_i$ for all requests i for some constant $\alpha \ge 1$; this will provide a full proof of Theorem 4.3.

Upper bound on the total latency of `MC-SF`. First, we define the modified quantity $\tilde{n}_o$ as the number of requests in the instance whose *predicted* output $\tilde{o}_i$ equals o. We analogously define the modified requests groups $\tilde{U}_\ell$, namely for $\ell = 0, \dots, \lfloor \log \tilde{o}_{\max} \rfloor$ ($\tilde{o}_{\max}$ is the largest $\tilde{o}_i$), let $\tilde{U}_\ell$ denote the set of requests that have predicted output length in the interval $[2^\ell, 2^{\ell+1})$.

In the presence of imperfect predictions, Lemma B.1 becomes the following.

Lemma B.1. *Consider one of the sets of requests $\tilde{U}_\ell$, and let $\underline{o}_\ell := 2^\ell$ and $\bar{o}_\ell := 2^{\ell+1} - 1$ denote the smallest and largest predicted possible output length for requests in this set. Let $\underline{t}$ and $\bar{t}$ be the first and last time the algorithm `MC-SF` processes a request in $\tilde{U}_\ell$. Then the distance between these times can be upper bounded as*

$$\bar{t} - \underline{t} \le \frac{192\alpha^2}{M} \cdot \sum_{\tilde{o}=\underline{\tilde{o}}_\ell}^{\bar{o}_\ell} \tilde{n}_o \cdot vol_o + 5\bar{o}_\ell.$$

Proof. (For the remainder of the proof we omit the subscript ℓ in $\underline{o}_\ell$ and $\bar{o}_\ell$.) The proof follows the same steps as that of the old Lemma B.1. We start by partitioning the interval $\{\underline{t}, \dots, \bar{t}\}$ into disjoint subintervals $I_1, I_2, \dots, I_w$ of length $\bar{o}$ (where I_w is the only exceptional interval that can be smaller than $\bar{o}$). Again, for an interval $I \subseteq [T]$, let $\text{peak}(I)$ be the *actual* (not predicted) peak memory use by `MC-SF` during this interval. Slightly abusing notation, also let $\text{vol}_\ell(I)$ be the total actual amount of memory that the requests $\tilde{U}_\ell$ occupy in `MC-SF`'s schedule added up over all times in the interval I. The next claim is the "peak-to-volume" Claim 4.6 in the presence of uncertain predictions.

Claim B.2 (Peak-to-volume). Consider any 3 consecutive intervals I_j, I_{j+1}, I_{j+2} of length $\bar{o}$. Then $\text{vol}_\ell(I_j \cup I_{j+1} \cup I_{j+2}) \ge \frac{1}{4\alpha^2} \text{peak}(I_{j+1}) \cdot \frac{\text{vol}_{\bar{o}}}{s+\bar{o}}$.

Proof of Claim B.2. Let $\tilde{t} \in I_{j+1}$ be the time when the peak memory occupation $\text{peak}(I_{j+1})$ happens. Since the interval I_{j+1} is strictly between times $\underline{t}$ and $\bar{t}$, by definition of the algorithm it only processes requests of $\tilde{U}_\ell$, and so only those contribute to the memory occupation at time $\tilde{t}$. If k of these requests contribute to this peak occupation, then each contributes at most $s + \bar{o}$ to it (recall that the true output length is always at most the predicted output length, which is at most $\bar{o}$ for requests in $\tilde{U}_\ell$); thus, we have $k \ge \frac{\text{peak}(I_{j+1})}{s+\bar{o}}$. Since such request is completely processed in the bigger interval $I_j \cup I_{j+1} \cup I_{j+2}$, each such request contributes with at least $\text{vol}_{\underline{o}/\alpha}$ to the memory volume $\text{vol}_\ell(I_j \cup I_{j+1} \cup I_{j+2})$ (recall that the true output length is always at least $\frac{1}{\alpha}$ of the predicted output length); hence, $\text{vol}_\ell(I_j \cup I_{j+1} \cup I_{j+2}) \ge k \cdot \text{vol}_{\underline{o}/\alpha}$. Finally, since $\underline{o}/\alpha \ge \frac{1}{2\alpha}\bar{o}$, a quick calculation shows that $\text{vol}_{\underline{o}/\alpha} \ge \frac{1}{4\alpha^2}\text{vol}_{\bar{o}}$. Combining these threes inequalities gives the claim. □

We now conclude the proof of Lemma B.1. Suppose for contradiction that $\bar{t} - \underline{t} > \frac{192\alpha^3}{M} \sum_{o=\underline{o}}^{\bar{o}} \tilde{n}_o \cdot \text{vol}_o + 5\bar{o}$. First, since $\bar{o} \le 2\underline{o}$, again we have $\text{vol}_o \ge \frac{1}{4}\text{vol}_{\bar{o}}$ for all o between $\underline{o}$ and $\bar{o}$. Then since $|\tilde{U}_\ell| = \sum_{o=\underline{o}}^{\bar{o}} \tilde{n}_o$, our assumption implies that $\bar{t} - \underline{t} > \frac{48\alpha^3}{M} \cdot |\tilde{U}_\ell| \cdot \text{vol}_{\bar{o}} + 5\bar{o}$.

This further implies that w (the number of the intervals I_j of length $\bar{o}$ in this period) is at least

$$w \geq \left\lfloor \frac{\frac{48\alpha^3}{M} \cdot |\tilde{U}_\ell| \cdot \mathrm{vol}_{\bar{o}} + 5\bar{o}}{\bar{o}} \right\rfloor \geq \frac{48\alpha^3}{M} \cdot |\tilde{U}_\ell| \cdot \frac{\mathrm{vol}_{\bar{o}}}{\bar{o}} + 4.$$

Every such interval I_j other than I_w has real peak memory utilization more than $\frac{1}{\alpha}(M-(s+\bar{o}))$: otherwise even using the predicted output lengths $\tilde{o}_i \leq \alpha o_i$, which is what effectively the memory constraint check in `MC-SF` uses, there would be a time in this interval where the algorithm would have scheduled an additional request from $\tilde{U}_\ell$. Then applying Claim 4.6 to the first $\lfloor \frac{w-1}{3} \rfloor$ groups of 3 consecutive intervals I_j's, we obtain that

$$\begin{aligned} \mathrm{vol}_\ell(\{\underline{t}, \ldots, \bar{t}\}) &\geq \left\lfloor \frac{w-1}{3} \right\rfloor \cdot \frac{1}{4\alpha^2} \cdot \frac{(M-(s+\bar{o}))}{\alpha} \cdot \frac{\mathrm{vol}_{\bar{o}}}{s+\bar{o}} \\ &\geq \frac{4}{M} \cdot |\tilde{U}_\ell| \cdot \frac{\mathrm{vol}_{\bar{o}}}{\bar{o}} \cdot \frac{M}{2} \cdot \frac{\mathrm{vol}_{\bar{o}}}{s+\bar{o}} > |\tilde{U}_\ell| \cdot \mathrm{vol}_{\bar{o}}, \end{aligned} \tag{14}$$

where the second inequality uses the assumption M is twice as big as the maximum single request predicted occupation, i.e., $s+\bar{o} \leq \frac{M}{2}$, and the last inequality uses $\mathrm{vol}_{\bar{o}} = s\cdot\bar{o} + \frac{\bar{o}\cdot(\bar{o}+1)}{2} > \frac{1}{2}\bar{o}\cdot(s+\bar{o})$.

However, each request $\tilde{U}_\ell$ contributes at most $\mathrm{vol}_{\bar{o}}$ to $\mathrm{vol}_\ell(\{\underline{t}, \ldots, \bar{t}\})$, and thus $\mathrm{vol}_\ell(\{\underline{t}, \ldots, \bar{t}\}) \leq |\tilde{U}_\ell| \cdot \mathrm{vol}_{\bar{o}}$; this contradicts Equation (14), and concludes the proof of the lemma. □

We then get the following upper bound on `MC-SF` that generalizes Lemma 4.4.

Lemma B.3 (UB on `MC-SF`)**.** *The total latency incurred by the algorithm `MC-SF` is at most*

$$\frac{1536\alpha^3}{M} \sum_o \tilde{n}_o \cdot \sum_{o' \leq o} \tilde{n}_{o'} \cdot vol_{o'} + 24 \sum_o \tilde{n}_o \cdot o$$

Proof. Let t_ℓ be the first time a request in $\tilde{U}_\ell$ is (starting to be) processed by `MC-SF` (let $\ell_{\max} := \lfloor \log \tilde{o}_{\max} \rfloor$ and let $t_{\ell_{\max}+1}$ be the last time the algorithm is processing something). The latency for each request in $\tilde{U}_\ell$ is at most $t_{\ell+1} + \bar{o}_\ell$ (i.e., if the algorithm starts processing requests from the next group then is has already started to process all requests in $\tilde{U}_\ell$, which take at most $+\bar{o}_\ell$ time to complete); thus, the total latency is at most $\sum_\ell |\tilde{U}_\ell| \cdot (t_{\ell+1} + \bar{o}_\ell)$. However, from Lemma B.1 we know that

$$t_{\ell+1} - t_\ell \;\leq\; \frac{192\alpha^3}{M} \cdot |\tilde{U}_\ell| \cdot \mathrm{vol}_{\bar{o}_\ell} + 5\bar{o}_\ell + 1 \;\leq\; \frac{192\alpha^3}{M} \cdot |\tilde{U}_\ell| \cdot \mathrm{vol}_{\bar{o}_\ell} + 6\bar{o}_\ell,$$

and so $t_{\ell+1} \leq \frac{192\alpha^3}{M} \sum_{\ell' \leq \ell} |\tilde{U}_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}} + 6\sum_{\ell' \leq \ell} \bar{o}_{\ell'}$. This gives that the total latency of `MC-SF` can be upper bounded as

$$\texttt{MC-SF} \;\leq\; \frac{192\alpha^3}{M} \underbrace{\sum_\ell |\tilde{U}_\ell| \sum_{\ell' \leq \ell} |\tilde{U}_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}}}_{A} + 6 \underbrace{\sum_\ell |\tilde{U}_\ell| \sum_{\ell' \leq \ell} \bar{o}_{\ell'}}_{B}. \tag{15}$$

Concluding the proof of the lemma requires just a bit of algebra to clean up the bound.

To upper bound the term A, let $\tilde{O}_\ell := \{\underline{o}_\ell, \ldots, \bar{o}_\ell\}$ be the possible predicted output lengths of the requests in $\tilde{U}_\ell$. Again we observe that since $\bar{o}_\ell \leq 2\underline{o}_\ell$, we have $\mathrm{vol}_{\bar{o}} \leq 4\mathrm{vol}_o$ for every $o \in \tilde{O}_\ell$. Moreover, since $|\tilde{U}_\ell| = \sum_{o \in \tilde{O}_\ell} \tilde{n}_o$, we have

$$|\tilde{U}_\ell|^2 \cdot \mathrm{vol}_{\bar{o}_\ell} \leq 2\Bigg(\sum_{o \in \tilde{O}_\ell} \tilde{n}_o \sum_{o' \in \tilde{O}_\ell, o' \leq o} \tilde{n}_{o'} \Bigg) \cdot \mathrm{vol}_{\bar{o}_\ell} \leq 8 \sum_{o \in \tilde{O}_\ell} \tilde{n}_o \sum_{o' \in \tilde{O}_\ell, o' \leq o} \tilde{n}_{o'} \cdot \mathrm{vol}_{o'}$$

and for $\ell' < \ell$

$$|\tilde{U}_\ell| \cdot |\tilde{U}_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}} \leq \left(\sum_{o \in \tilde{O}_\ell} \tilde{n}_o \sum_{o' \in \tilde{O}_{\ell'}} n_{o'} \right) \cdot \mathrm{vol}_{\bar{o}_{\ell'}} \leq 4 \sum_{o \in \tilde{O}_\ell} \tilde{n}_o \sum_{o' \in \tilde{O}_{\ell'}} \tilde{n}_{o'} \cdot \mathrm{vol}_{o'},$$

which combined give

$$|\tilde{U}_\ell| \sum_{\ell' \leq \ell} |\tilde{U}_{\ell'}| \cdot \mathrm{vol}_{\bar{o}_{\ell'}} \leq 8 \sum_{o \in \tilde{O}_\ell} \tilde{n}_o \sum_{o' \leq o} \tilde{n}_{o'} \cdot \mathrm{vol}_{o'},$$

and so adding up over all ℓ gives the upper bound $A \leq 8 \sum_o \tilde{n}_o \sum_{o' \leq o} \tilde{n}_{o'} \cdot \mathrm{vol}_{o'}$.

To upper bound the term B in (15), we observe that since the $\bar{o}_\ell$'s grow exponentially, $\sum_{\ell' \leq \ell} \bar{o}_{\ell'} \leq 2\bar{o}_\ell$. Then since $\bar{o}_\ell \leq 2o$ for every $o \in \tilde{O}_\ell$, we have

$$B \leq 2 \sum_\ell |\tilde{U}_\ell| \cdot \bar{o}_\ell = 2 \sum_\ell \sum_{o \in \tilde{O}_\ell} \tilde{n}_o \cdot \bar{o}_\ell \leq 4 \sum_\ell \sum_{o \in \tilde{O}_\ell} \tilde{n}_o \cdot o = 4 \sum_o \tilde{n}_o \cdot o.$$

Plugging these upper bounds on A and B on (15), we get

$$\texttt{MC-SF} \leq \frac{1536\alpha^3}{M} \sum_o \tilde{n}_o \sum_{o' \leq o} \tilde{n}_{o'} \cdot \mathrm{vol}_{o'} + 24 \sum_o \tilde{n}_o \cdot o.$$

This concludes the proof of Lemma B.3. □

Lower bound on the total latency of OPT. In the presence of incorrect predictions, we have the following lower bound on the optimal latency, which mirrors Lemma 4.7.

Lemma B.4. *The total optimal latency OPT satisfies:*

$$OPT \geq \frac{1}{6M\alpha^2} \sum_o \tilde{n}_o \cdot \sum_{o' \leq o} \tilde{n}_{o'} \cdot vol_{o'} + \frac{1}{6\alpha} \sum_o \tilde{n}_o \cdot o.$$

For this, we consider the same LP relaxation as before, with the required adjustments. Namely, let $\tilde{U}_o$ denote the set of requests with predicted output length o (so $|\tilde{U}_o| = \tilde{n}_o$); thus the true volume of each such request is at least $\mathrm{vol}_{o/\alpha} \geq \frac{\mathrm{vol}_o}{\alpha^2}$. Let $\bar{a}_o^t$ be the number of requests in $\tilde{U}_o$ that finish at time t in the optimal solution. The memory volume of all requests that finish up to time t need to fit in the total memory $t \cdot M$ available up to that time, and hence $\sum_{t' \leq t} \sum_o \bar{a}_o^t \cdot \frac{\mathrm{vol}_o}{\alpha^2} \leq t \cdot M$. Moreover, $\sum_t \bar{a}_o^t = \tilde{n}_o$ (all requests in $\tilde{U}_o$ finish at some time). Finally, the $\sum_o \bar{a}_o^t$ requests that finish at time t have latency (recall all requests are released at time 0) equal to t, and the optimal latency OPT is given by $\sum_t t \cdot \sum_o \bar{a}_o^t$. Together these observations show that OPT can be lower bounded by the following Linear Program with variables a_o^t, where in particular we relax the requirement that $\bar{a}_o^t$'s are integers:

$$\begin{aligned} \mathrm{OPT}_{LP} := \min & \sum_t t \cdot \sum_o a_o^t \\ \text{s.t.} & \sum_{t' \leq t} \sum_o a_o^t \cdot \mathrm{vol}_o \leq t \cdot M\alpha^2, \qquad \forall t \\ & \sum_t a_o^t = \tilde{n}_o, \qquad \forall o \\ & a_o^t \geq 0, \qquad \forall t, o. \end{aligned} \tag{16}$$

We then lower bound OPT_{LP}. Consider the optimal solution $\{a_o^{*t}\}_{t,o}$ for this LP. For a given output size o, let t_o^* be the first time t where $a_o^{*t} > 0$, i.e., where a request U_o is assigned to time t. Using exactly the same argument as in the proof of Lemma 4.7 we have that the "first time" values t_o^* are non-decreasing, namely $t_o^* \le t_{o'}^*$ when $o < o'$.

Let $t^*_{o_{\max}+1}$ be the last time such that $\sum_o a_o^{*t} > 0$. Using this observation we will prove the following lower bound on the "first times" t_o^*.

Claim B.5. For all o we have $t^*_{o+1} \ge \frac{1}{M\alpha^2} \sum_{o' \le o} \tilde{n}_{o'} \cdot \text{vol}_{o'}$.

Proof. By the observation above, in the optimal solution $\{a_t^{*o'}\}_{o',t}$ all items with output length at most o are assigned to times $\le t^*_{o+1}$, i.e., no later than when the next output length is assigned; thus, $\sum_{t' \le t^*_{o+1}} a_{o'}^{*t'} = \tilde{n}_{o'}$ for all $o' \le o$. Then considering (16) to time t^*_{o+1} we get

$$\sum_{o' \le o} \tilde{n}_{o'} \cdot \text{vol}_{o'} = \sum_{t' \le t^*_{o+1}} \sum_{o' \le o} a_{o'}^{*t'} \cdot \text{vol}_{o'} \le t^*_{o+1} \cdot M\alpha^2,$$

and rearranging we get the claim. □

We are now able to prove the lower bound on OPT from Lemma B.4.

Proof of Lemma B.4. By definition of t_o^*, we know that $a_o^{*t} = 0$ for all $t < t_o^*$, and so $\sum_t t \cdot a_o^{*t} \ge t_o^* \cdot \sum_{t \ge t_o^*} a_o^{*t} = t_o^* \cdot \tilde{n}_o$. Plugging the bound on t_o^* from the previous claim and adding over all o we can lower bound OPT_{LP} (and thus OPT) as

$$\begin{aligned}
\text{OPT} \ge \text{OPT}_{LP} &= \sum_t t \cdot \sum_o a_o^{*t} \\
&\ge \sum_o \tilde{n}_o \cdot t_o^* \\
&\ge \sum_o \tilde{n}_o \cdot \left(\frac{1}{M\alpha^2} \sum_{o' < o} \tilde{n}_{o'} \cdot \text{vol}_{o'} \right) \\
&= \frac{1}{M\alpha^2} \sum_o \tilde{n}_o \sum_{o' \le o} \tilde{n}_{o'} \cdot \text{vol}_{o'} - \frac{1}{M\alpha^2} \sum_o \tilde{n}_o^2 \cdot \text{vol}_o. \qquad (17)
\end{aligned}$$

To remove the negative term in the right-hand side (and add a new one, to match that of Lemma B.3), we provide two other lower bounds on the original OPT (not the LP).

The first is that for any predicted output length o, due to memory constraints, at most $\frac{\tilde{n}_o}{2}$ of them can be finished in the optimal schedule by time $\frac{\tilde{n}_o}{2} \frac{\text{vol}_o}{M\alpha^2}$, since the total memory available up to this time is $\frac{\tilde{n}_o}{2} \cdot \frac{\text{vol}_o}{\alpha^2}$ and each such request consumes at least $\frac{\text{vol}_o}{\alpha^2}$ memory; thus, the total latency on the optimal solution for the request of predicted output length o is at least $\frac{\tilde{n}_o}{2} \cdot \frac{\tilde{n}_o}{2} \frac{\text{vol}_o}{M\alpha^2}$. Adding this over all o, the total latency OPT has the lower bound

$$\text{OPT} \ge \frac{1}{4M\alpha^2} \sum_o \tilde{n}_o^2 \cdot \text{vol}_o. \qquad (18)$$

For the second additional lower bound, since each request of predicted output length o takes at least $\frac{o}{\alpha} + 1$ units of processing/time to finish (and thus has latency at least $\frac{o}{\alpha}$), we also have $\text{OPT} \ge \sum_o \tilde{n}_o \cdot \frac{o}{\alpha}$.

Adding this bound plus 4 times (18) plus (17) we get

$$6\,\mathrm{OPT} \geq \frac{1}{M\alpha^2}\sum_o \tilde{n}_o \sum_{o' \leq o} \tilde{n}_{o'} \cdot \mathrm{vol}_{o'} + \sum_o \tilde{n}_o \cdot \frac{o}{\alpha}. \tag{19}$$

This concludes the proof of Lemma B.4. □

Combining the bound on `MC-SF` from Lemma B.3 and the lower bound on OPT from Lemma B.4, we obtain $\texttt{MC-SF} \leq \alpha^5 \cdot O(1) \cdot \mathrm{OPT}$, which is $O(1) \cdot \mathrm{OPT}$ as α is taken to be constant. Thus, this concludes the proof of Theorem 4.3.

C Appendix for Section 5

All experiments were conducted on a Microsoft Surface Laptop with Snapdragon® X Elite (12 Core) processor.

Algorithm 2: Memory Constrained Benchmark (`MC-Benchmark`)

Input: Memory capacity M, time horizon T
Output: Schedule for processing requests
for *each round* $t = 1$ *to* T **do**
  Let $S^{(t)}$ be the set of requests that have already stated processing, and let $R^{(t)}$ be the set of waiting requests at time t. Also set $U^{(t)} = \emptyset$
  for *each request* $i \in R^{(t)}$ *in ascending order of arrival time* a_i **do**
    Set a time list $t' = p_j + o_j$ for $j \in S^{(t)} \cup U^{(t)} \cup \{i\}$
    if *all inequalities in Equation* (5) *hold for all* t' **then**
      Add request i to $U^{(t)}$
    else
      Break the for loop
  Process the requests in $S^{(t)} \cup U^{(t)}$

In this section, we provide additional details for the numerical experiments with real data. We take the memory capacity as 16,492 as observed in real experiments reported to us by systems engineers through private communication.

We next summarize the figures that support our key findings in the numerical experiments. Figure 7 displays the empirical distributions of input prompt lengths and output response lengths from the dataset used in our study. Figures 8 and 11 show that our algorithm consistently utilizes most of the available memory per batch, under both high-demand and low-demand conditions respectively. Figures 9 and 12 report the total latency for various values of the protection level parameter α, and identify recommended ranges for its selection in high- and low-demand scenarios, respectively. Finally, Figures 10 and 13 evaluate the impact of the KV cache clearing probability β on total latency, highlighting appropriate ranges of β for high- and low-demand settings respectively.

In Figure 9, we varied α while fixing $\beta = 0.1$ and $\beta = 0.2$. For both settings, $\alpha \in [0.15, 0.25]$ minimizes average latency. When $\alpha < 0.1$, performance degrades significantly as the protected memory is insufficient, necessitating frequent clearing and rescheduling of requests, which leads to redundant computation.

Figure 10 illustrates average latency for various values of β with fixed α values of 0.1 and 0.2. In both cases, the algorithm performs well with $\beta \in [0.05, 0.25]$. For extremely low β, the

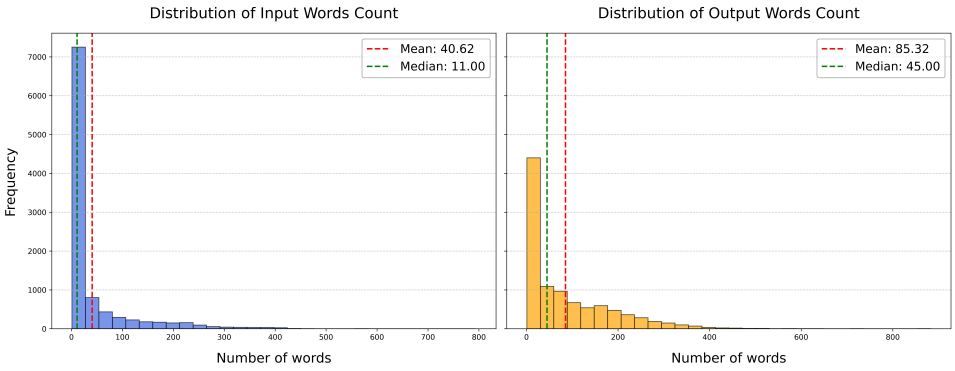

Figure 7: Distribution of the number of words of input prompt and output response respectively

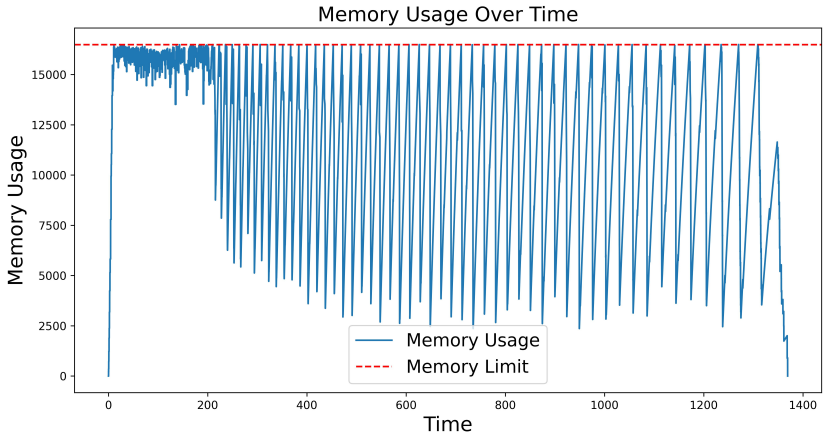

Figure 8: Memory Usage over Time for Algorithm 1 in the High Demand Case

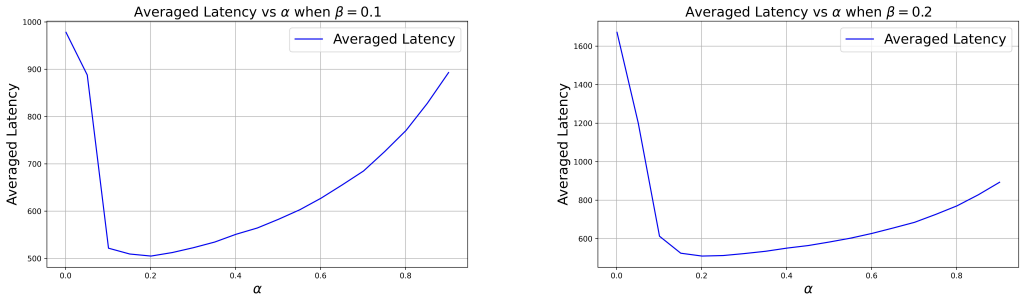

Figure 9: Average End-to-End Latency for Different α Values under the High Demand Case

algorithm underperforms as insufficient clearing of requests limits memory availability. This may keep the memory usage be above the limit after clearing the processing requests. To clear enough memory with extremely small β, this requires a significant time. Conversely, higher β values are inefficient due to excessive request clearing, resulting in increased recomputation.

In Figure 12, we vary α while fixing β at 0.1 and 0.2. For both settings, values of α in the range $[0.10, 0.25]$ minimize average latency, while $\alpha < 0.1$ leads to significant performance degradation.

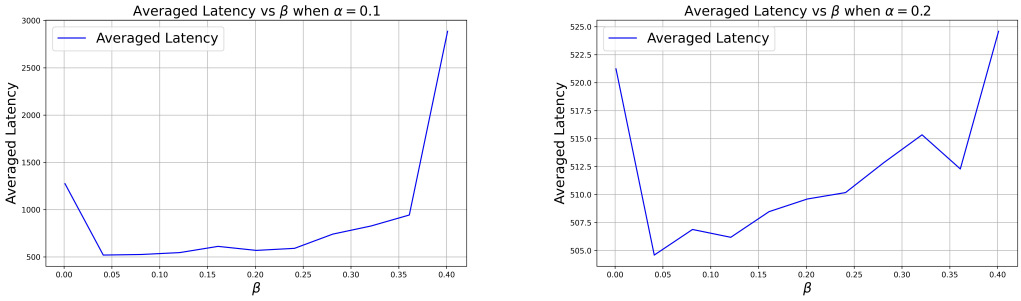

Figure 10: Average End-to-End Latency for Different β Values under the High Demand Case

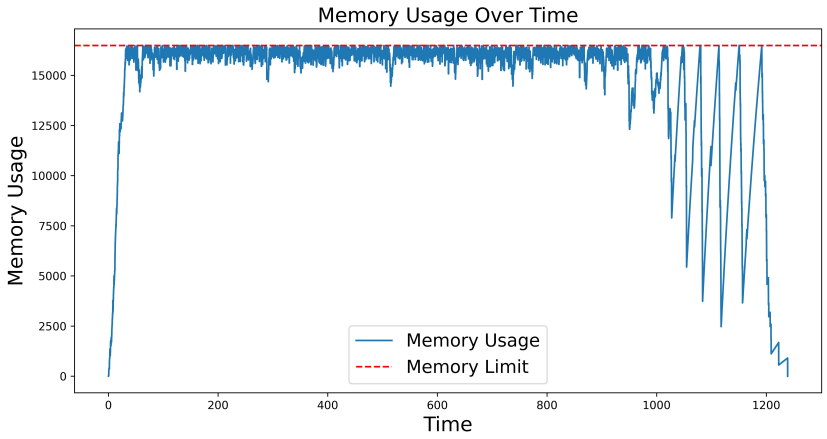

Figure 11: Memory Usage over Time for Algorithm 1 in the Low Demand Case

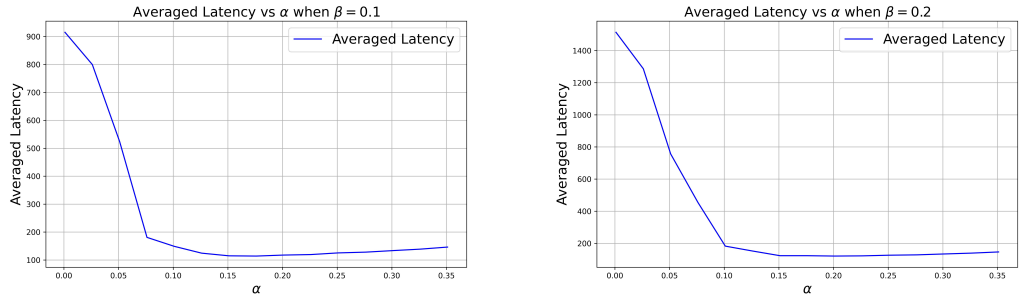

Figure 12: Average End-to-End Latency for Different α Values under the Low Demand Case

Figure 13 shows the average latency across various β values with α fixed at 0.1 and 0.2. In both cases, the algorithm achieves stable performance for β values between 0.05 and 0.20. These trends are consistent with those observed in the high-demand scenario, indicating that similar parameter tuning benefits both models.

Finally, we provide a table with relevant statistics for the case of 1000 requests with arrival rate $\lambda = 50$ for 50 independent runs.

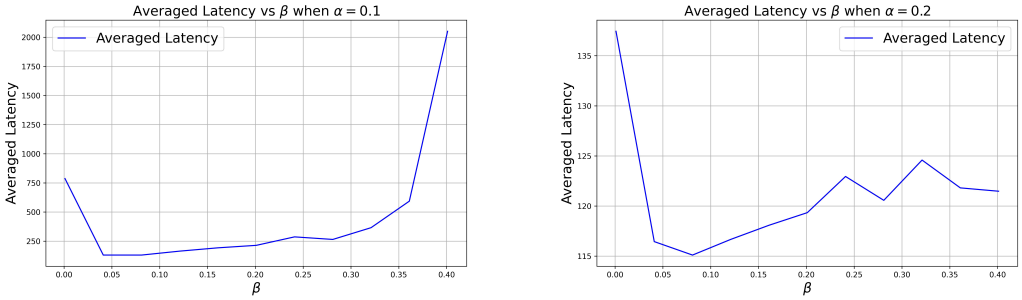

Figure 13: Average End-to-End Latency for Different β Values under the Low Demand Case

Table 1: Relevant statistics among 50 independent experiments of different algorithms where there are 1000 requests, and the arrival rate $\lambda = 50$

Algorithm	Average	Std. Dev.	Max	Min
`MC-SF`	32.112	0.354	33.097	31.505
`MC-Benchmark`	46.472	0.310	47.135	45.838
Benchmark $\alpha = 0.3$	51.933	0.324	52.532	51.204
Benchmark $\alpha = 0.25$	51.046	0.351	51.757	50.279
Benchmark $\alpha = 0.2$, $\beta = 0.2$	50.401	0.343	51.035	49.700
Benchmark $\alpha = 0.2$, $\beta = 0.1$	50.395	0.360	51.083	49.586
Benchmark $\alpha = 0.1$, $\beta = 0.2$	53.393	1.457	56.357	49.488
Benchmark $\alpha = 0.1$, $\beta = 0.1$	50.862	0.946	53.978	49.086