Fast Inference for Augmented Large Language Models

Rana Shahout[†], Cong Liang[§], Shiji Xin[†], Qianru Lao[†],
Yong Cui [§], Minlan Yu[†], Michael Mitzenmacher[†]

Harvard University [†], Tsinghua University [§]

在一个 req 进行 API call 时决定如何管理 memory，例如前面部分算的 KV cache

ABSTRACT

Augmented Large Language Models (LLMs) enhance the capabilities of standalone LLMs by integrating external data sources through API calls. In interactive LLM applications, efficient scheduling is crucial for maintaining low request completion times, directly impacting user engagement. However, these augmentations introduce scheduling challenges due to the need to manage limited memory for cached information (KV caches). As a result, traditional size-based scheduling algorithms, such as Shortest Job First (SJF), become less effective at minimizing completion times. Existing work focuses only on handling requests during API calls by preserving, discarding, or swapping memory without considering how to schedule requests with API calls. In this paper, we propose *LAMPS*, a novel LLM inference framework for augmented LLMs. *LAMPS* minimizes request completion time through a unified scheduling approach that considers the total length of requests and their handling strategies during API calls. Recognizing that LLM inference is memory-bound, our approach ranks requests based on their consumption of memory over time, which depends on both the output sizes and how a request is managed during its API calls. To implement our scheduling, *LAMPS* predicts the strategy that minimizes memory waste of a request during its API calls, aligning with but improving upon existing approaches. We also propose starvation prevention techniques and optimizations to mitigate the overhead of our scheduling. We implement *LAMPS* on top of vLLM and evaluate its performance against baseline LLM inference systems, demonstrating improvements in end-to-end latency by 27%-85% and reductions in TTFT by 4%-96% compared to the existing augmented-LLM system, with even greater gains over vLLM.

1 INTRODUCTION

Recent progress in large language models (LLMs) has initiated a new wave of interactive AI applications. A prominent example is OpenAI's ChatGPT [14], which facilitates conversational interactions across various tasks. One way to extend LLM capabilities is to augment them with external tools [12], resulting in what we refer to as API-augmented requests. These augmentations include arithmetic calculation [8], ChatGPT plugins [15], image generation [4], and virtual environments [22]. Consequently, AI development is increasingly moving towards compound AI systems [28] that integrate multiple interacting components, such as model calls, retrievers, and external tools, rather than relying solely on monolithic models.

API-augmented requests present several challenges, particularly regarding memory consumption during the LLM decoding phase, which is memory-bound and requires careful management. Each request has associated key and value matrices that grow in size during the request. LLMs cache these matrices in a key-value (KV) cache throughout the sequence generation process to enhance efficiency, eliminating the need to recompute them at every iteration. This caching significantly reduces computation time but requires substantial memory, roughly proportional to the input and output lengths, the number of layers, and hidden dimensions. High memory consumption during decoding can translate to higher latency and lower throughput, as it limits the system's ability to process multiple requests concurrently.

With API augmentation, memory demands increase further, depending on how the system handles requests during API calls (Figure 2). There are three main strategies for handling a request's KV cache during API calls:

- *Preserve*: The system retains the KV cache in memory while waiting for the API response.
- *Discard and Recompute*: The system discards and recomputes the KV matrices from the start once the API returns.
- *Swap*: The system offloads the KV cache to the CPU to free up memory, and reloads it when the API returns.

We refer to these as *memory handling strategies*, or *handling strategies* for brevity. All three strategies have downsides. With *Preserve*, because the KV cache remains in memory throughout the API call, memory is wastefully consumed during the call. *Discard and Recompute* incurs additional memory and computational costs when recomputing the KV cache. *Swap* introduces overhead from pausing running requests and managing data transfer between CPU and GPU memory. Additionally, because the duration of API calls can vary significantly across different augmentation types and requests, one strategy does not fit all requests. For instance, a simple calculation might be completed in milliseconds, whereas image generation could take several seconds.

Existing LLM inference systems generally operate for standalone LLMs and fail to guarantee low request completion time for augmented LLMs. This issue becomes more significant when handling requests involving external API calls, leading to delays such as head-of-line (HoL) blocking. HoL blocking occurs when long-running requests, including those waiting for API responses, prevent shorter ones from being processed efficiently. Previous works attempt to mitigate the issue of HoL blocking by managing memory during API calls. For example, vLLM [11] discards and recomputes API-augmented requests, treating API calls as termination signals and the request returning from the API as a new job. INFERCEPT [1] classifies requests and applies strategies dynamically, aiming to reduce memory wastage for API-augmented requests. However, both systems still rely on a first-come first-served (FCFS) scheduling policy, which increases HoL blocking by allowing long-running requests to block shorter ones, especially under high load.

Scheduling strategies can reduce head-of-line blocking by prioritizing requests. Traditional size-based scheduling prioritizes jobs[1] with shorter execution time. Without API calls, this approach is effective, as execution time and memory usage for LLM requests without API calls correlate (with the execution time corresponding roughly to the output length) [21]. However, with API-augmented requests, the output length may not reflect the total request time, including API calls. A request with a short output might involve a lengthy API call, while a request with a longer output may require minimal API interaction.

Rather than treating LLM execution and API calls as separate processes, we argue that integrating the scheduling and memory handling of requests can significantly improve request completion time, especially under high load. Achieving this requires information about both request output length and API call time, which are often unavailable, so we rely on predicting these values. This challenge leads to a key question: *Given predictions of request output length and API call time, how should we schedule and handle API-augmented requests to prevent head-of-line blocking and minimize latency?*

This paper presents *LAMPS* (*LLM API- and Memory-based Predictive Scheduling*), a novel inference framework for augmented LLMs. *LAMPS* minimizes request completion time through a unified scheduling approach that jointly considers the total length of requests and their API call-handling strategies. Recognizing that LLM inference is memory-bound, our approach ranks requests based on their *memory over time*, which reflects how memory resources are allocated throughout a request's processing. By considering both the output size and the memory handling strategies during API calls, *LAMPS* enables effective management of varying output sizes and API interactions.

To achieve this, *LAMPS* predicts memory consumption using request properties, such as pre-API output length and API characteristics. We develop a prediction model leveraging the opt-125m language model [29] to estimate pre-API output lengths from input prompts while predicting API duration based on the type of API being called. Based on this profile, we determine which handling strategy to use for a request: Preserve, Discard and Recompute, or Swap. If a request involves multiple API calls, each call is treated independently, and the request is reinserted into the scheduling process after each API call is handled, continuing until the last API call is completed. Our approach aligns with INFERCEPT equations [1], except we determine requests' memory strategy *before* scheduling them. We then schedule the requests according to our policy, which aims to minimize response time by prioritizing requests by their memory footprint over time, accounting for API interactions. While it is theoretically possible to collectively optimize handling and scheduling decisions, such an approach is impractical in an online setting. Consequently, we employ a greedy algorithm that first minimizes memory usage for each individual request and then schedules requests based on their memory requirements over time.

Our contributions are as follows:

- We develop a prediction model to estimate pre-API output lengths from input prompts and predict API duration based on the type of API being called, allowing us to predict memory consumption over time.
- Using the predicted memory consumption over time, we assign a handling strategy to each request before processing.
- We propose a scheduling policy that considers the request length and the API handling strategy to minimize request completion time. We integrate optimizations into our scheduling policy to mitigate starvation and reduce scheduling overhead.
- We implement *LAMPS* on top of vLLM [11], a state-of-the-art LLM inference system. We evaluate *LAMPS* on two datasets, comparing *LAMPS* against baseline systems. Our results show that *LAMPS* consistently outperforms INFERCEPT across various datasets and request rates, achieving improvements in end-to-end latency ranging from 27% to 85% and reductions in TTFT from 4% to 96%, with even greater improvements over vLLM. We also analyze the components of *LAMPS* and the effect of prediction accuracy on its performance.

[1]The terms job and request are used interchangeably in this paper.

2 BACKGROUND AND MOTIVATION

LLMs expand their capabilities by integrating external tools, allowing them to handle more complex tasks. However, this augmentation introduces challenges for request handling and scheduling with the goals of minimizing response times and managing memory efficiently during inference. This section presents the background for LLM execution and API interactions.

2.1 Augmented LLMs

Augmented Language Models [12, 24] refer to language models that enhance their capabilities by incorporating external tools, retrieval mechanisms, or reasoning strategies to overcome the limitations of traditional LLMs. Unlike pure LLMs, which rely solely on pre-trained parameters to generate responses, augmented LLMs can query external data sources to expand their capabilities. Figure 1 shows an example of an augmented LLM request. These augmentations, which we refer to as *API* (Application Programming Interfaces), fall into three main categories as described in [12]: incorporating non-LLM tools during decoding (such as calculators [25], information retrieval systems [3]), iterative self-calling of an LLM (like chatbots maintaining conversation history), and complex compositions involving multiple LLMs, models, and tools (exemplified by frameworks like LangChain [5], DSpy [10], Gorilla [17], SGLang [30], and AgentGraph [6]).

LLM API time varies significantly based on augmentation types, with a clear distinction between short-running and long-running augmentations. Despite this variation, today's systems still rely on FCFS scheduling. This suggests that API handling strategies should be tailored to specific augmentation types rather than using a one-size-fits-all approach.

2.2 Transformer-Based Generative Models

At each step, a Transformer model generates the most probable next token based on the sequence of previously generated tokens. A model generating a sequence of length n needs to perform n iterations, with each token passing through several layers of self-attention and feed-forward networks.

During the i-th iteration, the model operates on all prior tokens $(t_0, t_1, \ldots, t_{i-1})$ using self-attention mechanisms. The resulting output can be represented as:

$$h_{\text{out}} = \text{softmax}\left(\frac{q_i \cdot K^\top}{\sqrt{d_h}}\right) \cdot V$$

Here, q_i is the query vector for the current token t_i, while K and V are matrices containing the key and value vectors for all preceding tokens, where $K, V \in \mathbb{R}^{i \times d_h}$.

2.2.1 Key-Value (KV) Cache. To reduce computational overhead, LLMs cache the key and value matrices (KV cache) during sequence generation. This approach avoids recomputing these matrices at each step, improving efficiency but leading to high memory usage, which scales with the sequence length, number of layers, and hidden dimensions. As more tokens are generated, memory demands grow, particularly for long sequences. For instance, the GPT-3 175B model requires around 2.3 GB of memory to store key-value pairs for a sequence length of 512 tokens. This high memory consumption poses challenges for efficient preemptive scheduling, especially when working with limited GPU memory.

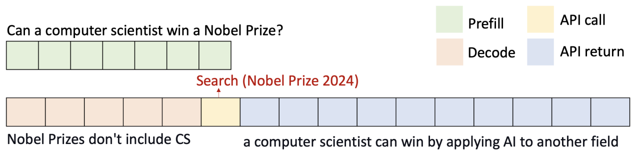
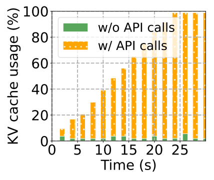

Can a computer scientist win a Nobel Prize?
Prefill
Decode
API call
API return
Search (Nobel Prize 2024)
Nobel Prizes don't include CS
a computer scientist can win by applying AI to another field

Figure 1: Illustration of an augmented-LLM request. The API fetches detailed information about the 2024 Nobel Prize.

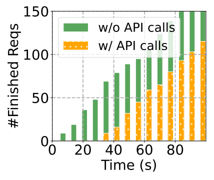
2.2.2 Scheduling in LLMs. In LLM inference systems, iteration-level scheduling, as implemented in systems like Orca [27] and vLLM [11], is commonly used. It differs from traditional request-level scheduling, where the system processes a batch of requests until completion, forcing earlier requests to wait until the entire batch is completed, and new requests must wait in a queue until the next batch is processed. Iteration-level scheduling processes one token at a time for each request in the batch, allowing the system to dynamically adjust the batch after every iteration. Requests that complete an iteration can exit the batch, and new ones can be introduced, optimizing resource usage within the constraints of GPU memory. The default policy in these systems is FCFS. Most recent scheduling works [21, 26], however, focus on LLM requests without API augmentations. API-augmented requests introduce challenges, such as handling external interactions and variable API call times, which require new scheduling strategies beyond those used for standalone LLMs. Accordingly, *LAMPS* focuses on developing such scheduling for API-augmented requests.

2.3 Handling Requests During API

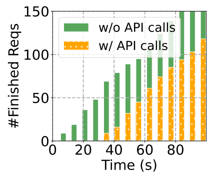
Optimizing memory management during API calls involves selecting a strategy that minimizes GPU memory waste. In an augmented LLM inference system, this choice depends on two factors: the duration of the API call and the length of the pre-API output. For brief API calls, the Preserve strategy may be advantageous in avoiding recomputation overhead. For longer API calls, either Discard or Swap is preferable. If the pre-API portion of the request is computationally light



(a) Preserve



(b) Preserve



(c) Discard

Figure 2: Impact of including API calls: using a subset of INFERCEPT [1], we compare two variations of the dataset—one with API calls and one without. (a) KV cache usage (%) over time when all API calls are handled using Preserve. (b) Number of completed requests over time using Preserve. (c) Number of completed requests over time using Discard.

	R_1	R_2	R_3
Total length	6	2	3
API start after	5	1	2
API duration	2	7	1
Memory action	Preserve	Discard	Swap

Table 1: Total length and API call duration in iterations unit.

(short), Discard is beneficial. Otherwise, Swap may be more efficient despite the potential delays it introduces.

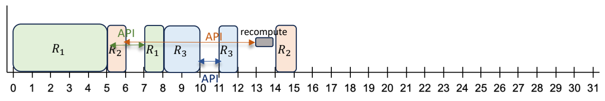
INFERCEPT addresses this optimization challenge by developing the following equations (equations 1-3 in [1]) that model the memory wastage associated with each handling strategy.

$$\text{WastePreserve}_i = T_{\text{INT}} \times C_i \times M \tag{1}$$

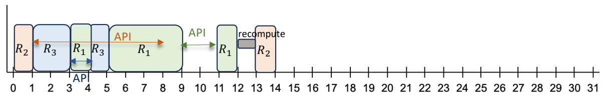
$$\text{WasteDiscard}_i = T_{\text{fwd}}(C_i) \times C_i \times M + T_{\text{fwd}}(C_i) \times C_{\text{other}} \times M \tag{2}$$

$$\text{WasteSwap}_i = 2 \times T_{\text{swap}}(C_i) \times C_{\text{batch}} \times M \tag{3}$$

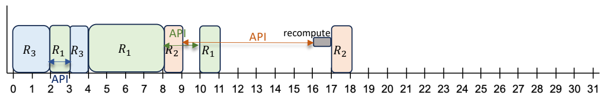
Here T^j_{INT} is the duration of the API call for request i, and C_i, C_{other}, and C_{batch} represent the context size (in tokens) of request i before the API call, the context size of other requests in the batch with request i, and the total context size of all requests in the batch, respectively. M denotes the memory consumed per token for the KV cache. $T_{\text{fwd}}(C_i)$ and $T_{\text{swap}}(C_i)$ represent the time required for model forwarding with context C_i and the time to swap context C_i, respectively. INFERCEPT dynamically selects a strategy that minimizes memory waste. However, its scheduling policy remains FCFS.



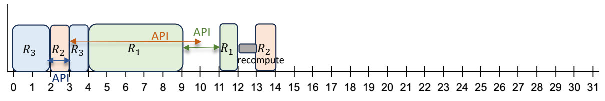
(a) FCFS



(b) SJF (request length)



(c) SJF (request+API length)



(d) Preferred*

Figure 3: Comparison of scheduling policies for API-augmented requests with a memory budget of 6 unit, requests lengths and API duration are summarized in Table 1. (a) FCFS: yielding an average request completion time of 11.66 units. (b) SJF: achieving an average request completion time of 10.33 units. (c) SJF by Total Length: Orders requests by total length, achieving an average request completion time of 11 units (d) Optimized: Integrates length and API handling, achieving an average request completion time of 10 units.

3 CHALLENGES AND DESIGN PRINCIPLES

3.1 Challenge: Scheduling API-augmented Requests

Our goal is to reduce average and typical request times through scheduling. Size-based scheduling methods, such as Shortest Job First (SJF), can reduce request completion time by utilizing known or predicted request sizes. Traditional scheduling methods encounter challenges with API-augmented requests. Indeed, it is not clear what the size should be with API calls: should the API delay be included are not? Even assuming known output lengths, SJF can fail to perform optimally when requests involve API calls. The following example in Figure 3 illustrates this issue.

Example. Consider three requests R1, R2, and R3 that all arrive at time 0. Each request includes one API call, triggered at different times during decode generation. Their output lengths are 6, 2, and 3 tokens, respectively, with API durations (in token generation units) of 2, 7, and 1 units, respectively. The strategies to handle requests during the API for each request (determined by the INFERCEPT equations) are Preserve for R1, Discard for R2, and Swap for R3. Table 1 summarizes this information.

In this example, we assume that only one request can run at a time and the memory budget is limited to 6 units. We first consider the setting of INFERCEPT. The strategy for handling each request during its API call is determined dynamically at runtime using the INFERCEPT equations ((1), (2) and (3)). For simplicity, we assume complete information is known about the total length and API duration of each request. A request is scheduled only if there is enough memory available.

Although all requests arrive at the same time, the FCFS scheduling policy used by INFERCEPT determines their order based on request ID, processing them as R_1, R_2, R_3. With a memory budget of 6 units, the scheduler processes the pre-API part of R_2 during R_1's API call, as R_2 will be discarded after one unit, freeing memory to continue with R_1. In contrast, R_3's pre-API part cannot run during R_1's API call because it will not release memory before the API response completes, preventing R_1 from resuming. This scheduling yields an average request completion time of 11.66 units (Figure 3a). The SJF policy schedules requests based only on length, processing them from shortest to longest: R_2, R_3, R_1. At time unit 8, the API of R_2 completes, leaving a post-API part of length 2 (including recomputation). However, the running request, R_1, also has two units remaining, so R_2 must wait. At time unit 9, R_1 enters its API call, consuming five units of memory, leaving only 1 unit available, which is insufficient to start the post-API part of R_2. As a result, R_2 must wait until R_1 finishes. This policy results in an average request completion time of 10.33 units (Figure 3b).

These approaches fall short of optimal scheduling because they ignore the interaction between scheduling and request handling during API calls. A naive strategy, referred to as *SJF by total length* (Figure 3c), orders requests by their total length (output length plus API duration). Again, in this example, the pre-API part of R_2 can run during R_1's API call. This policy achieves an average request completion time of 11 units, worse than SJF. A more effective scheduling policy (Figure 3d) integrates total length and the API handling strategy. This approach yields an average request completion time of 10 units, outperforming previous methods. Notably, the post-API part of R_2 becomes ready at time unit 10, but due to memory constraints, it waits until R_1 finishes.

Our intuition is that, under memory constraints, R_3, the least memory-intensive request, should run first to release memory quickly. It should be followed by R_2, with R_1 (the most memory-consuming request due to its Preserve handling) scheduled last. This insight informs our proposal to incorporate API handling strategies into the scheduler, ranking requests based on memory consumption over time.

Dataset	Type	Duration (sec)	Num
INFERCEPT	Math	(9e-5, 6e-5)	(3.75, 1.3)
	QA	(0.69, 0.17)	(2.52, 1.73)
	VE	(0.09, 0.014)	(28.18, 15.2)
	Chatbot	(28.6, 15.6)	(4.45, 1.96)
	Image	(20.03, 7.8)	(6.91, 3.93)
	TTS	(17.24, 7.6)	(6.91, 3.93)
ToolBench	-	(1.72, 3.33)	(2.45, 1.81)

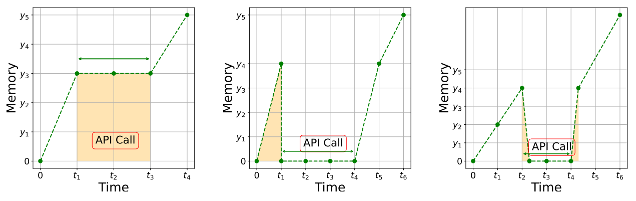
Table 2: API durations and number for two different datasets: INFERCEPT [1] and ToolBench [18]. First part of this table is taken from INFERCEPT [1] (Table 1).

3.2 Key Design Principles

3.2.1 Predicted API handling strategy. Our approach integrates the selection of API handling strategies directly into the scheduling policy, determining the appropriate handling strategy before the request is processed. To do this, we first estimate the context size for batched requests, considering the memory usage of other requests that might be affected during the API handling phase. This estimation involves profiling the number of requests in a batch. API durations are predictable based on API types, as each corresponds to specific operations with known computational complexities and resource demands. For example, math APIs, which involve simple calculations, have short execution times, while image generation APIs, requiring intensive computation, have longer durations. Analysis of historical data (Table 2) shows that execution times within the same API type have low variance, enabling reliable predictions. Lastly, we use a lightweight predictor (see Section 6 for details) to estimate the pre-API length from input prompts.

In contrast, INFERCEPT incorporates the pre-API output length and API duration for each request, also considering the impact a request has on all other requests in memory. When a request reaches the API, INFERCEPT dynamically selects a strategy to minimize memory waste based on these factors. However, its scheduling policy follows a simple FCFS, whereas our method integrates these considerations directly into a more adaptive scheduling policy.

3.2.2 Integrating API handling strategies with scheduling. Minimizing request completion time requires a unified scheduling method that considers both the total length of requests and their specific handling strategies during API calls. By knowing whether a request will Preserve, Discard, or Swap the model during an API call, the scheduler can predict the impact on system resources and the request completion time and rank the requests accordingly. For example, it may order



(a) Preserve **(b) Discard** **(c) Swap**

Figure 4: Memory consumption over time for a request with one API call using three memory management strategies: (1) Preserve, (2) Discard and Recompute, and (3) Swap. The highlighted area represents memory waste for one request.

two requests with the same total length differently because they have different handling strategies during the API call. Or it may prioritize a request with a longer total length but a more memory-friendly handling strategy during an API call over a shorter request with a handling strategy that may more negatively impact system performance, as our example showed (e.g., Figures 3d and 3b).

4 DESIGN

4.1 System Overview

LAMPS combines API handling strategies with scheduling policies to minimize the completion time for API-augmented requests. Using design principles in Section 3.2, *LAMPS* employs three main steps to reduce LLM inference response time: predicting the pre-API output length and the duration of the API, determining the handling of requests during API calls, which aims to minimize memory waste, and, finally, proposing a scheduling policy that considers both the request length and the API handling method.

Figure 5 illustrates the *LAMPS* architecture. Users submit requests to the request pool. *LAMPS* predicts pre-API output length and estimates API properties (duration and response length, Table 2) based on input prompts. Using these predictions, *LAMPS* estimates memory consumption over time, considering this in API handling decisions and scheduling policy ranking. *LAMPS* determines how to handle requests during API calls to minimize memory waste, aligning with INFERCEPT (equations (1), (2) and (3)). Each request is labeled with a handling strategy (preserve, discard, or swap). Based on this handling method and request output length, *LAMPS* implements a scheduling policy tailored for API-augmented requests. This policy prioritizes requests based on their memory consumption over time. The pseudocode of the *LAMPS* scheduler is provided in Algorithm 1.

Algorithm 1 *LAMPS* Scheduler

```
Input: Request pool $P$, predictor model $Predictor$, waiting queue $WaitingQueue$, running batch $runningBatch$, starvation threshold $StarvationT$.
while True do
    for all $r \in P$ do
        $predictions_r \leftarrow Predictor(r.prompt)$
        $r.handling \leftarrow HandlingStrategy(predictions_r)$
        $WaitingQueue$.put($r$)
    end for
    for all $r \in PQueue \cup DQueue \cup SQueue$ do
        if $r$.APIcallFinished() then
            $WaitingQueue$.put($r$)
        end if
    end for
    for all $r \in WaitingQueue$ do
        $r.score \leftarrow$ HandlingRanking($r$)
    end for
    $WaitingQueue \leftarrow$ Sort($WaitingQueue$) by $r.score$
    $runningBatch \leftarrow \emptyset$
    for all $r \in WaitingQueue$ do
        if $runningBatch$ is not full then
            $runningBatch \leftarrow runningBatch + r$
            $r.StarvationCnt \leftarrow 0$
        else
            $r.StarvationCnt \leftarrow r.StarvationCnt + 1$
        end if
    end for
    for all $r \in WaitingQueue$ do
        if $r.StarvationCnt \geq StarvationT$ then
            Place $r$ in $WaitingQueue$ head
            $r.StarvationCnt \leftarrow 0$
        end if
    end for
    Remove finished requests from $WaitingQueue$
    Execute $runningBatch$
    for all $r \in runningBatch$ do
        if $r$.encounterAPIcall() then
            if $r.handling == Preserve$ then
                $PQueue$.put($r$)
            else if $r.handling == Discard$ then
                $DQueue$.put($r$)
            else if $r.handling == Swap$ then
                $SQueue$.put($r$)
            end if
        end if
    end for
end while
```

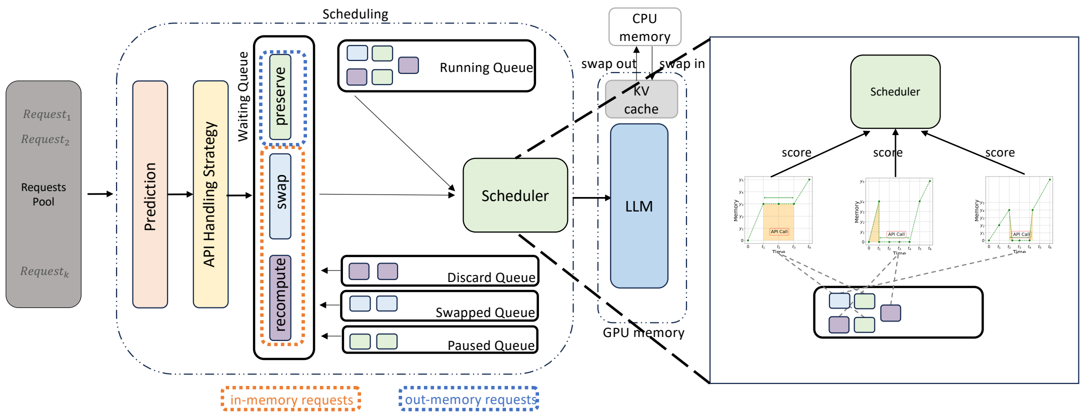


Figure 5: *LAMPS* architecture. *LAMPS* minimizes completion time for API-augmented requests through three steps: predicting pre-API output length and API properties, determining request handling strategies (preserve, discard, or swap) to minimize memory waste, and implementing a scheduling policy based on the handling method and request output length.

4.2 Predicting API Handling Strategy

Our goal is to predict the best API handling strategy that minimizes memory waste before a request is processed, enabling the scheduler to rank each request. In the following discussion, we explain how to predict the best handling strategy for requests during API calls, assuming a single API for simplicity. We first predict the output length and API duration. Our approach generalizes beyond the dataset by using a predictor to estimate pre-API output length based on the prompt. Output length prediction is studied in the context of LLMs. Several works have modeled output length prediction as a classification problem, using models such as DistilBERT and OPT [7, 9, 23]. These approaches categorize outputs into discrete bins to estimate lengths. Other methods use regression-based techniques to predict length [19, 20]. Moreover, recent work [21] demonstrates using LLM layer embeddings to predict output length. Our approach builds on these insights.

For the API duration, we leverage the fact that APIs belong to fixed classes (e.g., Math, Image), each with consistent functionality and a similar duration (latency). By extracting the API type from the prompt, we estimate the API response length using the average length from the training set for that API class. This method relies on the standardized outputs and consistent behaviors of APIs within each class.

We classify and determine how to handle requests during API calls by predicting memory waste based on pre-API and API duration estimates. INFERCEPT uses Equations (1), (2), and (3) to compute the memory waste. Here, we present an equivalent method to quantify memory usage (specifically calculating waste based on predictions) by considering the memory-over-time function. For simplicity, we focus on a single API, but this approach extends to multiple APIs.

Figure 4 represents the memory-over-time function and the highlighted areas in Figures 4a, 4b and 4c represent the memory waste of a single request due to an API call. We combine this waste with our estimation of the context size for batched requests according to the setup, aligning with Equations (1), (2), and (3), respectively. After estimating the memory waste for each option, we select the strategy that minimizes the waste. Memory consumption increases linearly with the output length until the request reaches the API call, as seen in all three cases. In Figure 4a, the Preserve strategy keeps memory allocated throughout the API call. Thus, memory use remains constant during the call, with the shaded area indicating this idle memory. In Figures 4b and 4c, the Discard and Swap strategies release memory during the API call, resulting in zero memory usage while waiting for the API response. In Figure 4b, the Discard approach recomputes the first part after the API, while the Swap approach (Figure 4c) shows a delay before the memory is fully released (swapped out) and then restored (swapped in) after the API call completes. Swapping in causes a spike in memory consumption.

Our insight is that evaluating memory usage by integrating the memory-over-time function offers a more accurate measure of resource consumption than relying on instantaneous memory values. The integral of memory versus time provides the total memory consumption for a request, accounting for the API's impact on memory usage. This approach incorporates output length, API duration, and how the request is handled during the API call. Instantaneous memory measurements fail to reflect how long memory resources are occupied, which is particularly important in the decoding phase of LLMs where the system is memory-bound. It is not just the amount of memory a request consumes at a particular moment but also how long that memory remains in use. A strategy that uses more memory for a shorter period can be more efficient than one that uses less memory but occupies it longer. Integrating memory over time captures memory waste across different strategies during API calls.

Multi-API. To generalize to requests involving multiple APIs, we break down each request into segments, each ending with a single API call. After an API call completes, the request re-enters the system for further processing and is treated as a new request focused on the subsequent API call. At this point, we classify the request based on the characteristics of the current API. For example, suppose a job involves initial processing, followed by two API calls interleaved with additional processing phases. We divide this job into segments, each consisting of a processing phase and a single API call at the end. We estimate the returned token length in each segment based on the specific API call. While this approach does not account for the cumulative memory consumption of the entire job, predicting the total number of API calls and their combined resource usage beforehand is challenging. This segmentation aligns with INTERCEPT, which handles multi-API requests by processing jobs incrementally as they reach each API.

Effect of Mispredictions. Mispredictions are to be expected. Small mispredictions in API duration or output size will typically have a small effect; indeed, they may not change the overall ranking of jobs. Mispredicting a short API or output as long may have a large effect on that particular job, but does not typically harm other jobs in the system. (See related results in [13].) A long-running API call incorrectly predicted as short may lead the system to select a memory-wasteful strategy, such as preserving the request in memory during the API call. This unnecessary memory consumption may limit the system's ability to process additional requests and reduce overall throughput. A request with a long output misclassified as short may cause head-of-line blocking and delay other requests. The main overhead in this scenario is increased latency.

4.3 Scheduling Policy

Traditional job scheduling typically assumes that job completion times are either completely unknown—making First-Come, First-Served (FCFS) a natural strategy—or known in advance or predictable, enabling size-based policies like Shortest Job First (SJF) and Shortest Remaining Process Time (SRPT) to minimize average response time.

Integrating API calls into the output response increases memory consumption, which may degrade performance due to memory constraints. In LLM systems, when memory is full, jobs are either discarded and recomputed when memory becomes available, or KV cache entries are swapped from the GPU to the CPU. Both approaches impact response time: discarding requires recomputation, and swapping interrupts the model's forward pass, causing delays for the entire batch. Intuitively, requests should be ranked based on their memory consumption. Without API calls, ranking based on memory consumption aligns with ranking based on service time (or request length), as memory consumption has a linear relationship with request length. With API calls, this relationship breaks, as requests are handled differently according to the strategy that minimizes memory consumption during the API. Consider Figure 4, which shows memory over time; we consider the area (integral) as a rank function of a request and select the function based on the predicted handling strategy during the API call. For example, the memory over time function matches Figure 4a for the predicted preserve strategy for a request. This approach incorporates the strategy of handling requests during API within the scheduling policy and provides a way to compare and rank requests among different handling strategies. Referencing Figure 3, consider memory consumption over time. Among the three requests, R_3 consumes the least memory and should be prioritized, followed by R_2. R_1 consumes the most memory due to its length, API duration, and the preserve handling strategy, so it should be scheduled last.

Our scheduling strategy uses iteration-level scheduling [27], where the scheduler ranks requests at the end of each iteration. We use a selective score update mechanism to reduce the overhead of frequent ranking. For example, for datasets with long-running requests, frequent score updates are unnecessary; instead, we cache their scores and refresh them at predefined intervals. This balances ranking accuracy with the computational costs of maintaining updated scores.

4.4 Starvation Prevention

Scheduling policies can cause certain requests to experience long wait times, leading to high tail latency, a form of starvation that degrades system performance and user experience. This issue arises when longer or resource-intensive requests are continually deferred in favor of shorter ones, exacerbating tail latency. Our memory-focused scheduling policy alone does not detect and mitigate starvation, which can result in extended wait times and reduced fairness. To solve this, we have implemented a starvation prevention mechanism to improve the scheduler's tail latency using a per-request counter. The counter increments when a request remains in the waiting queue for a new iteration. Upon reaching a predefined threshold, *LAMPS* tags the request as starving and prioritizes it by placing it at the head of the scheduled requests for the current iteration. The relative order of prioritized and non-prioritized requests is maintained according to *LAMPS*'s ranking decisions. Prioritization continues until request completion, avoiding memory waste from preempted (half-finished) requests. If the request has not been prioritized and encounters API calls or is scheduled, the counter resets to 0. Parameter experiments led us to set the predefined threshold at 100 (testing with the datasets in Section 6). This value effectively filters out requests that complete within a reasonable time while identifying starving ones. The starvation prevention mechanism activates only when *LAMPS* initially schedules the request to the running queue, thus leveraging *LAMPS*'s sorting decisions.

5 IMPLEMENTATION

We implement *LAMPS* on top of vLLM [11], a state-of-the-art LLM inference system. We enable the score update mechanism only on the ToolBench dataset with an interval of ten, while disabling it for other datasets where scheduling overhead is not a bottleneck. To implement the prediction mechanism, we use the OPT-125M language model [29], a transformer-based model developed by Meta. With 125 million parameters and support for a context length of 2048 tokens, OPT-125M can effectively handle datasets with long contexts, such as the multi-API dataset. Although smaller than many larger language models, OPT-125M delivers strong language generation capabilities. Our approach utilizes the embeddings generated by OPT-125M during the initial processing of input prompts. After tokenizing the input and processing it through the model's layers, we extract the final token's embedding, which is then fed into a linear classifier. This classifier assigns the input to one of 50 bins, each representing a range of 10 tokens, and is trained using cross-entropy loss.

The model estimates the completion length for each prompt based on learned representations from the ToolBench dataset [18], which involves complex conversations with API interactions. We train the model using an 80-20 split for training and validation, classifying output lengths into bins. We apply this model specifically to the ToolBench dataset because the other dataset already includes detailed output length information, making prediction unnecessary in that case. *LAMPS* is evaluated using the test portion of the ToolBench data to ensure accuracy.

6 EVALUATION

In this section, we first present end-to-end experiments demonstrating the overall performance improvements of *LAMPS* compared to INFERCEPT and vanilla vLLM on two different-sized LLM models. Next, we evaluate *LAMPS*'s design choices, highlighting the effectiveness of each component. Finally, we analyze the prediction component and the impact of mispredictions on *LAMPS*'s performance.

6.1 Methodology

LLM models. We use the 6B-parameter GPT-J model (which we denote by GPT-J 6B), and the 13B-parameter Vicuna model (Vicuna 13B). Both were also used by INFERCEPT.

Testbed. For the experiments, we used a machine with dual AMD EPYC 7313 CPUs (16 cores per CPU, totaling 64 threads), 503 GB of RAM, and two NVIDIA A100 GPUs with 80 GB memory each connected via NVLink. We manually limited the maximum memory usage of each GPU to 40 GB to emulate the setup used in INFERCEPT's experiments, which we inferred to involve A100 GPUs with 40 GB memory based on their use of AWS machines.

Datasets. We evaluate our system using two distinct datasets. The first, similar to the one used in INFERCEPT, includes arithmetic tasks, knowledge-based question answering, multi-step chatbot dialogues, and interactions in an embodied virtual environment. This dataset includes API execution times, frequencies, and output length. The second data set is ToolBench [18] is an instruction-tuning dataset tailored for tool-use tasks, comprising over 16,000 real-world APIs across 49 categories. It encompasses both single-API and multi-API scenarios, containing only prompts and API call types. We use this dataset to predict output length, API duration, and API response length.

Metrics. To evaluate *LAMPS*, we measure end-to-end latency, defined as the time from when a request is submitted to the system until its completion, time-to-first-token (TTFT) (reflecting system responsiveness), and throughput (indicating request generation speed). For each metric, we report the mean and 99th percentile (P99).

Baselines. As baselines, we compare *LAMPS* with both vanilla vLLM and INFERCEPT.

6.2 End-to-end Performance

End-to-end latency and TTFT vs. request rate. Figure 6 shows how varying the request arrival rate affects the mean and P99 of end-to-end latency and TTFT across three datasets: (1) a single-API dataset (a subset of the INFERCEPT dataset containing only a single API), (2) the full INFERCEPT dataset, and (3) the ToolBench dataset. We evaluated these metrics using the LLMs GPT-J 6B and Vicuna 13B.

GPT-J 6B results. *LAMPS* shows clear performance gains over vLLM and INFERCEPT in mean TTFT and end-to-end latency across all tested datasets. On the single-API dataset at a request rate of 3, *LAMPS* reduces mean TTFT by 4.61%

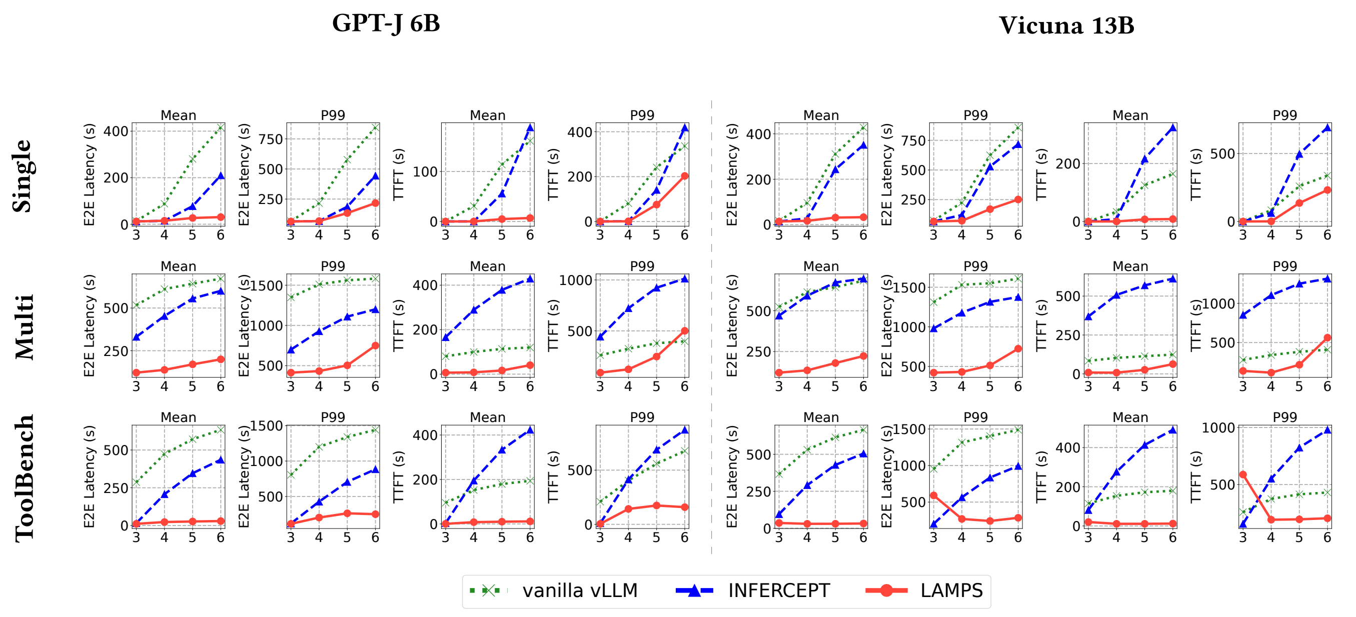


Figure 6: End-to-end performance (mean and P99 of latency and TTFT) as a function of request arrival rate when serving GPT-J 6B and Vicuna 13B using different datasets (single-API, multi-API, ToolBench).

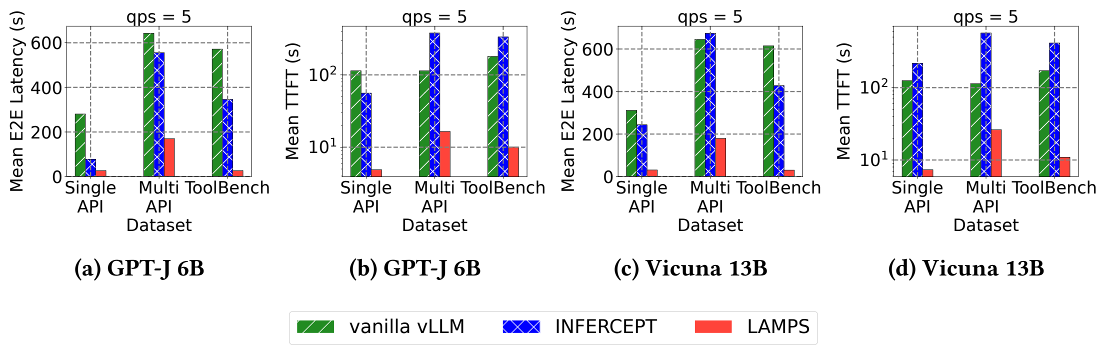


Figure 7: End-to-end performance (mean and P99 of latency and TTFT) of single-API, multi-API, ToolBench datasets when the request arrival rate is fixed to five when serving GPT-J 6B and Vicuna 13B.

compared to INFERCEPT and 22.86% compared to vLLM. Mean end-to-end latency increases slightly by 0.78% against INFERCEPT but drops by 14.48% compared to vLLM. At higher rates, such as 5, *LAMPS* further reduces mean TTFT by 91.27% over INFERCEPT and 95.71% over vLLM, with latency reductions of up to 65.51% and 90.44%, respectively. For the multi-API dataset, *LAMPS* achieves 95.93% TTFT reduction and 63.32% latency improvement over INFERCEPT at a rate of 3. On ToolBench, *LAMPS* reduces mean TTFT by 87.04% compared to INFERCEPT and 99.51% compared to vLLM, with a 27.24% latency reduction compared to INFERCEPT and 96.07% compared to vLLM at a request rate of 3.

Vicuna 13B results. *LAMPS* consistently outperforms vLLM and INFERCEPT in TTFT and end-to-end latency across all datasets. On the single-API INFERCEPT dataset, at a request rate of 3, *LAMPS* reduces mean TTFT by approximately 4.78% compared to INFERCEPT and 18.65% compared to vLLM, with mean end-to-end latency reductions of around 0.24% and 15.73%, respectively. For higher request rates (e.g., 4), *LAMPS* achieves significantly greater improvements, reducing mean TTFT by over 98% compared to baselines and end-to-end latency by up to 82%. On the multi-API dataset, *LAMPS* achieves similar gains, with TTFT reductions of over 89% and mean end-to-end latency improvements of up to 78%

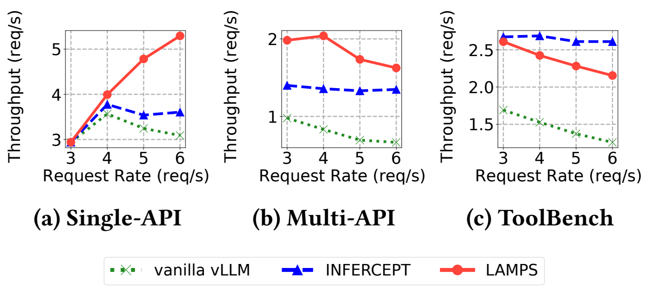


Figure 8: Throughput as a function of request arrival rate with Vicuna 13B using the different datasets.

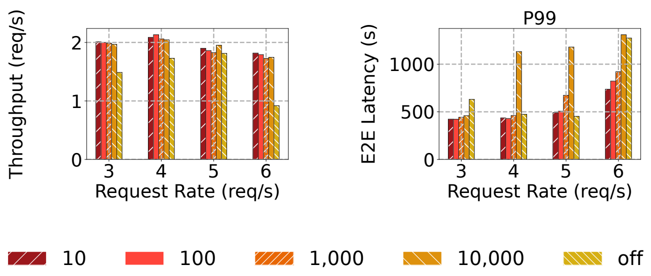


Figure 9: Starvation threshold, Multi-API dataset with GPT-J 6B.

at 4. In the ToolBench dataset, at a request rate of 3, *LAMPS* reduces mean TTFT by 75.46% and end-to-end latency by 62.61% compared to INFERCEPT while also achieving improvements over vLLM by up to 90.25% in mean latency.

At low request rates in a single-API dataset, the performance gap between *LAMPS* and the baseline methods is small, as FCFS and size-based policies perform similarly under less system pressure. However, as the arrival rate increases or with a multi-API dataset, the head-of-line blocking problem worsens with FCFS. For P99 latency and TTFT, while *LAMPS* shows a higher growth rate in these metrics at higher request rates, its absolute TTFT values remain lower than the baselines. This is because *LAMPS* prioritizes low-memory requests, leading to increased tail latency and TTFT at higher rates due to more pronounced starvation effects.

Figure 7 shows the mean end-to-end latency and mean time-to-first-token (TTFT) for different datasets with a fixed arrival rate of five. All systems exhibit a similar pattern across the datasets, with higher latency and TTFT for the Multi API and ToolBench datasets than the Single API dataset. However, *LAMPS* consistently achieves lower latency and TTFT across all datasets.

Throughput vs. request rate. Figure 8 shows throughput vs. request arrival rate across three datasets using Vicuna 13B. We measured throughput by limiting each benchmark to 30 minutes and counting completed requests. *LAMPS* achieves throughput gains over INFERCEPT and vLLM in single-API and multi-API scenarios. In single-API cases, *LAMPS* outperforms INFERCEPT by up to 46.81% and vLLM by 71.34% at higher request rates. The multi-API dataset highlights even greater gains, with *LAMPS* achieving up to 50.23% better throughput than INFERCEPT and over 144% compared to vLLM. However, the improvements over INFERCEPT are less pronounced on the ToolBench dataset due to many requests exceeding 2048 tokens. Both vLLM and INFERCEPT prioritize new requests over ongoing ones, leading to higher throughput for long requests, as seen with ToolBench. In contrast, our system focuses on optimizing end-to-end latency rather than throughput. While this strategy results in gains in latency and throughput on some datasets (such as single-API and multi-API), on the ToolBench dataset, we observe significant improvements in latency accompanied by a slight degradation in throughput. This trade-off occurs because optimizing latency can sometimes reduce throughput, especially with longer requests.

Figure 9 compares the throughput and tail latency of *LAMPS* under various starvation prevention thresholds. We observe that introducing a starvation prevention threshold reduces tail latency and enhances throughput, with a threshold of 100 providing a good balance between both metrics.

6.3 Breakdown of *LAMPS* Components

To further understand the benefits of *LAMPS*, we incrementally added its components to vLLM and compared the results with INFERCEPT. We used the Multi-API dataset because it has the highest latency among the datasets (Figure 7), Figure 10 shows throughput, as well as the mean and P99 of end-to-end latency and TTFT. First, we added the predicted API handling component to vLLM while keeping the scheduling policy as FCFS (referred to as *LAMPS* w/o scheduling). With this addition, the performance was close to INFERCEPT but slightly worse. The main difference between INFERCEPT and *LAMPS* w/o scheduling is that we use predicted information to estimate how to handle API calls in advance. In contrast, INFERCEPT dynamically decides how to handle requests during the API call when the request reaches the API. Next, we integrated our scheduling policy. The main improvements across all metrics came from the scheduling policy. Our scheduling policy effectively reduces head-of-line blocking and optimizes resource utilization. However, predicting the API handling policy is a necessary preliminary step in implementing our scheduling.

6.4 Prediction Component

In this subsection, we evaluate the impact of prediction errors on the performance of *LAMPS*.

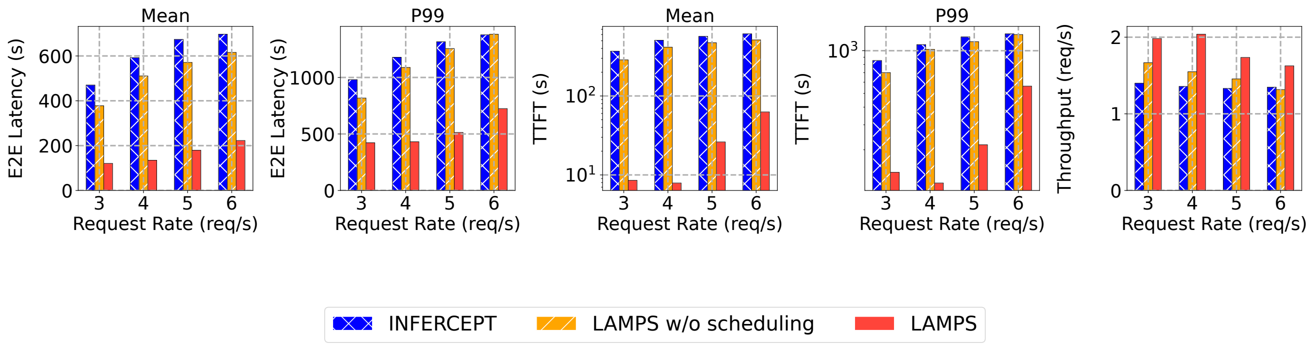


Figure 10: Breakdown of *LAMPS* components, using Multi-API dataset with Vicuna 13B.

Effect of Mispredictions. Using the INFERCEPT dataset, we inject controlled Gaussian errors into the predictions for API duration and output length: error $\sim \mathcal{N}(0, p \times m)$, where p is the error parameter and m is the measured value. The predicted values are then calculated as: predicted_value = measured_value + error. By varying the *error_parameter* parameter, we evaluate how different prediction inaccuracies affect the overall performance of *LAMPS*. Figure 11 demonstrates the impact of prediction errors on system performance, focusing on end-to-end latency and throughput. As the prediction error parameter increases (e.g., 5%, 10%, 30%, 50%), median latency increases, particularly under higher request rates (8-10 req/s). This indicates that inaccurate predictions lead to longer waiting times. Similarly, throughput decreases as error rates increase, especially at higher request rates. However, we observe that performance degradation in *LAMPS* occurs only when large prediction errors occur. This suggests that as long as reasonably accurate predictions are maintained, *LAMPS* can deliver improved performance.

Prediction Accuracy and Overhead. We evaluated the precision of our response length predictions using the ToolBench dataset by measuring the absolute difference between the predicted and actual word lengths (which is part of the dataset).We used two accuracy metrics, Acc-5 and Acc-15 that represent the percentage of predictions that differ from the actual length by no more than 5 words and 15 words, respectively. The results show 68.5% accuracy for Acc-5 and 78.3% accuracy for Acc-15, with a Mean Absolute Error (MAE) of 3.06. When focusing on the first 20 bins (responses up to 200 words), the MAE improves to 1.366, indicating higher accuracy for shorter responses. We used an NVIDIA A100 GPU for inference, achieving an average prediction time of 13.7 ms per input on the ToolBench dataset. Table 3 shows Acc-5 and Acc-15 per bin for the first ten bins.

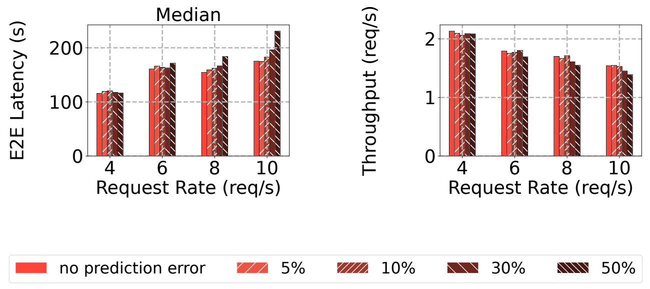


Figure 11: Error injection, Multi-API dataset with GPT-J 6B.

Bin num	0	1	2	3	4	5	6	7	8	9	10
Acc-5	0.0	0.75	0.834	0.813	0.713	0.769	0.568	0.5	0.321	0.333	0.3
Acc-15	0.0	0.75	0.921	0.933	0.867	0.851	0.691	0.545	0.321	0.462	0.3

Table 3: Bin accuracy for top 10 bins.

7 RELATED WORKS

Several studies focus on improving inference throughput through optimized scheduling strategies. Orca [27] introduces iteration-level scheduling, where a new batch is created at the end of each model forward pass. This approach increases GPU utilization and enhances inference throughput. Other research targets efficient GPU memory management. vLLM [11] introduces paged attention, treating the KV cache as virtual memory mapped to non-contiguous physical GPU memory, improving GPU memory utilization. Another line of work addresses the imbalance between the prefill and decoding stages. Sarathi [2] employs chunked prefill, which divides prompt tokens into smaller chunks merged with decoding requests to form a batch for each iteration. Splitwise [16] separates the prefill and decoding stages across different machines to match their distinct computational demands. These techniques for memory optimization are complementary and can be integrated with *LAMPS*.

Different approaches have been explored to design effective scheduling policies for LLMs. FastServe [26] builds upon Orca by scheduling each output token individually using a Multi-Level Feedback Queue (MLFQ) to avoid head-of-line blocking. However, this approach leads to frequent preemptions, increasing the cost of managing the key-value (KV) cache memory and offloading to the CPU. Prediction-based scheduling methods have been introduced to address these challenges. Zhen et al. [31] enhanced LLM inference by predicting response lengths with an additional LLM and scheduling requests based on these predictions. While this optimizes latency, it introduces overhead due to the extra LLM used for length prediction. Other works predict output lengths using lightweight models like DistilBERT and OPT [9]. Recently, Trail [21] obtained output length predictions directly from the target LLM by feeding the embedding of its internal structures into a lightweight classifier. Using these predictions, they scheduled requests with a prediction-based Shortest Remaining Processing Time variant with limited preemption to manage memory overhead. Importantly, however, all of these previous works focus on requests without API augmentations.

8 CONCLUSION AND FUTURE WORK

We have introduced *LAMPS* (*LLM API- and Memory-based Predictive Scheduling*), an LLM inference framework designed explicitly for API-augmented requests. *LAMPS* optimizes request completion time through a unified scheduling strategy that ranks requests based on their memory consumption over time. By predicting pre-API outputs, *LAMPS* can estimate the optimal handling strategy during API calls, choosing between preserving, discarding, or swapping memory to minimize memory waste. This predictive approach allows *LAMPS* to schedule requests with varying output sizes and API interactions effectively. Additionally, our framework incorporates a starvation prevention mechanism for better tail latency. Experimental results demonstrate that *LAMPS* improves end-to-end latency by 27%-85% and reduces TTFT by 4%-96% compared to INFERCEPT, with even greater gains over vLLM.

We believe that this work serves as a starting point for API-augmented requests. For further improvements, we will focus on enhancing prediction accuracy, particularly for memory-intensive requests, and aim to better handle multi-API requests by accounting for cumulative memory consumption throughout the entire process. Another direction for building upon *LAMPS* is to manage multiple LLMs, directing requests to the most suitable LLM based on the specific API type and the current load of the LLMs. This would be a load-balancing scheduling variation. Similarly, one might have requests that have to go through a sequence of LLMs and/or servers for API calls, according to some ordering (that differs among requests). This is similar to a jobshop scheduling variation.

More generally, we suggest that scheduling with API calls appears to open the door to many interesting algorithmic problems. We are not aware of API calls of the form considered here being studied in the (theoretical) scheduling algorithms literature. More consideration of algorithmic bounds for these types of problems may yield more additional practical strategies for API-augmented requests.

REFERENCES

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. In *Forty-first International Conference on Machine Learning*.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310* (2024).

[3] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. 1999. *Modern information retrieval.* Vol. 463. ACM press New York.

[4] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, et al. 2023. Improving image generation with better captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf* 2, 3 (2023), 8.

[5] H. LangChain Chase. [n.d.]. LangChain. https://github.com/langchain-ai/langchain.

[6] Lu Chen, Zhi Chen, Bowen Tan, Sishan Long, Milica Gašić, and Kai Yu. 2019. AgentGraph: Toward universal dialogue management with structured deep reinforcement learning. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 27, 9 (2019), 1378–1391.

[7] Ke Cheng, Wen Hu, Zhi Wang, Peng Du, Jianguo Li, and Sheng Zhang. 2024. Enabling Efficient Batch Serving for LMaaS via Generation Length Prediction. *arXiv preprint arXiv:2406.04785* (2024).

[8] Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2024. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *Advances in neural information processing systems* 36 (2024).

[9] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=zUYfbdNl1m

[10] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. 2024. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *The Twelfth International Conference on Learning Representations*.

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

[12] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. 2023. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842* (2023).

[13] Michael Mitzenmacher. 2021. Queues with Small Advice. In *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA 21)*. 1–12.

[14] OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt.

[15] OpenAI. March 2023. ChatGPT plugins. https://openai.com/blog/chatgpt-plugins.

[16] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[17] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334* (2023).

[18] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:2307.16789 [cs.AI]

[19] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. 2024. Power-aware Deep Learning Model Serving with {μ-Serve}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 75–93.

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

[21] Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. 2024. Don't Stop Me Now: Embedding Based Scheduling for LLMs. *arXiv preprint arXiv:2410.01035* (2024).

[22] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768* (2020).

[23] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. Dynamollm: Designing llm inference clusters for performance and energy efficiency. *arXiv preprint arXiv:2408.00741* (2024).

[24] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 186345.

[25] Wolfram. [n.d.]. Chatgpt gets its 'wolfram superpowers. https://writings.stephenwolfram.com/2023/03/chatgpt-gets-its-wolfram-superpowers/.

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

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

[28] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/.

[29] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[30] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104* (2023).

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