ELIS: Efficient LLM Iterative Scheduling System with Response Length Predictor

Seungbeom Choi*, Jeonghoe Goo*, Eunjoo Jeon, Mingyu Yang, Minsung Jang

Cloud Research Team, Samsung SDS

South Korea

Abstract

We propose ELIS, a serving system for Large Language Models (LLMs) featuring an *Iterative Shortest Remaining Time First (ISRTF)* scheduler designed to efficiently manage inference tasks with the shortest remaining tokens. Current LLM serving systems often employ a first-come-first-served scheduling strategy, which can lead to the "head-of-line blocking" problem. To overcome this limitation, it is necessary to predict LLM inference times and apply a shortest job first scheduling strategy. However, due to the auto-regressive nature of LLMs, predicting the inference latency is challenging. ELIS addresses this challenge by training a response length predictor for LLMs using the BGE model, an encoder-based state-of-the-art model. Additionally, we have devised the *ISRTF* scheduling strategy, an optimization of shortest remaining time first tailored to existing LLM iteration batching. To evaluate our work in an industrial setting, we simulate streams of requests based on our study of real-world user LLM serving trace records. Furthermore, we implemented ELIS as a cloud-native scheduler system on Kubernetes to evaluate its performance in production environments. Our experimental results demonstrate that *ISRTF* reduces the average job completion time by up to 19.6%.

1 Introduction

Since the launch of OpenAI ChatGPT in 2022, numerous Large Language Models (LLMs) and services have been introduced, achieving high–performance [9, 10]. LLMs based on Transformer decoders differ from traditional machine learning and encoder–based models in several ways. First, LLMs can perform multiple tasks such as summarization, text generation, and question–answering with a single model, meaning that one LLM must handle various user requests. Second, LLMs generate tokens auto–regressively, often causing longer inference latency compared to Transformer encoder-based models like BERT, which can process tokens in parallel. Third, LLMs are GPU memory-bound during inference, leading to speed degradation due to their billions of parameters.

Despite the challenges mentioned above, it is essential to serve as many user requests as possible at minimal cost. To achieve this goal, user requests for LLM services arriving at different times are batched together and passed to the LLMs. This traditional batching process creates inefficiency, as short-context jobs must wait for longer ones to finish. ORCA addresses this problem by implementing iteration-level batch scheduling, where jobs are processed per iteration without waiting for the entire batch to complete [35].

However, iteration-level batching uses a First–Come–First–Served (FCFS) scheduling strategy, which is non-preemptive and can lead to head-of-line blocking. Designing optimal latency-based priority scheduling for LLMs is challenging for several reasons. First, decoder-based LLMs have difficulty predicting the length of the generated output tokens, which is a major factor in latency. Second, the latency of the predictor should not become a bottleneck; thus, models that require a relatively large amount of computing resources, such as decoder-based generative models, cannot be used despite their high accuracy. Third, applying latency-based priority scheduling involves changing the priority of ongoing jobs and managing high context change costs, including Key-Value (KV) cache changes, which complicates the process.

Research on serving schedulers for LLMs has focused on enhancing iterative batching by predicting the execution time of each step and performing jobs with shorter execution times first using the Shortest Remaining Time First (SRTF) scheduler. Zheng *et al.* conducted instruct fine–tuning so that LLM models could provide both their responses and the length of their responses simultaneously [41]. By executing jobs with shorter response lengths first, they achieved over an 80% increase in throughput. However, a critical drawback of this approach is that instruct fine-tuning the LLM model can affect the original model's accuracy. Qiu *et al.* used a BERT model to predict the inference speed of LLMs and processed requests with a Shortest Job First (SJF) scheduler [26].

Table 1. Comparison to prior works.

Related Work	Scheduling	Predict	Preemption
ORCA [35]	FCFS	X	X
Zheng *et al.* [41]	SJF	One-off	X
ELIS	*SRTF*	*Iterative*	*O*

* indicates equal contribution.
Correspondence to: Minsung Jang <minsung.jang@samsung.com>.

Their results showed a 39.1% reduction in total Job Completion Time (JCT) compared to FCFS and a 2.21-fold increase in throughput. However, the BERT predictor introduced in the study had low accuracy with an F1-score of 0.6 and did not have a fallback plan for incorrect LLM inference time predictions, which could still potentially lead to head-of-line blocking.

To address these challenges, we propose ELIS, an **E**fficient **L**LM **I**terative **S**cheduling system designed to reduce average JCT by efficiently scheduling tasks based on predicted response lengths. The distinct features of our research are as follows:

First, we developed an SRTF-based LLM request scheduler, *Iterative Shortest Remaining Time First* (*ISRTF*), using a response length predictor. ISRTF prioritizes tasks with shorter remaining times, where the remaining time is predicted over several iterations using the method described in Section 4.2. We designed a modular architecture for the predictor, allowing the scheduler to operate in a model-agnostic manner. As the prediction performance improves through retraining based on log data, the effectiveness of SRTF can also increase. In this study, an iterative predictor was developed and applied ($R^2 = 0.852$), and we confirmed that accuracy increases with each iteration (Section 3.3). When applying the predictor, the ISRTF scheduler reduced the average JCT by up to 19.58% compared to FCFS on NVIDIA A100 GPUs.

Second, we implemented ELIS at the production level based on industrial workloads. In this study, we extracted trace distributions based on two months of actual operation data from FabriX[1], a cloud-native LLM service managed by our organization. Using these traces, we developed a request generator based on a Gamma distribution. Additionally, ELIS was implemented with Kubernetes, the most widely used container orchestrator for production, leveraging its pod auto-scalability and reliability features. We conducted performance studies using vLLM [16], a commonly utilized execution engine in both academic and industry research. ELIS successfully distributed prompts across more than 10 nodes and displayed near-linear scaling performance, achieving 18.77 Requests Per Second (RPS) with 50 workers on NVIDIA H100 GPUs.

Third, from the standpoint of a cloud service provider, GPUs are finite resources. Therefore, stopping resources allocated to lower-priority jobs and prioritizing high-priority LLM requests can enhance GPU utilization efficiency. In this study, we investigate the efficacy of preempting tasks using trace data obtained while our organization was hosting FabriX, and we report our results in Section 3.4. We have also included the code for adjusting the frequency of preemption in the public code of ELIS.

[1]Full name to be disclosed in final version of the paper.

The primary contribution of this paper is the introduction of ELIS, an iterative priority-based LLM task serving system, which reduced the average JCT by up to 19.58% compared to FCFS by deploying ISRTF scheduling. In addition, we contribute to the research community by implementing ELIS with the vLLM execution engine as Kubernetes components, enabling easy deployment.

2 Background

In this section, we explain the architecture of LLMs, focusing on the auto-regressive inference process, which is divided into two distinct phases: prefill and decode. We also describe strategies for efficient LLM serving, such as batching and scheduling.

2.1 LLM Inference Process

Autoregressive Generation: Typically, the inference procedure of LLMs can be divided into two distinct phases: prefill and decode. In the prefill phase, which corresponds to the first iteration, the prompt is processed to generate the KV cache. In the decoding phase, the query, key, and value of new tokens are calculated step by step and added to the KV cache generated during the prefill phase [32].

The latency of LLM inference can be characterized by two metrics. One is the *Time To First Token (TTFT)*, which is the duration of the prefill phase, and the other is the *Time Per Output Token (TPOT)*, which represents the average time taken to generate each token during the decoding phase [32]. Total LLM latency can be calculated as *TTFT* + (*TPOT* × number of tokens to be generated). Although the LLM latency formula is clear, it is difficult to predict the latency of each request. This is because, due to the auto-regressive nature of LLMs, it is hard to predict the number of output tokens that will be generated for a given prompt, which has the greatest impact on LLM latency.

2.2 Efficient LLM Serving Strategies

LLMs such as GPT-3 [10] and PaLM have been deployed in services like Bing and Bard, handling a large number of requests every day. Providing satisfying service to users while reducing the inference cost of LLMs becomes a crucial issue [41]. Over the years, significant developments have been made in efficient LLM serving techniques, such as kernel fusion, pipeline parallelism, tensor parallelism, and quantization. In this section, we explain LLM inference batching and request scheduling methods to reduce LLM inference costs.

Continuous Batching: LLM services perform inference by batching inputs from multiple users at different intervals to increase throughput. Conventional batching starts a batch all at once and waits for all requests in the batch to complete their computation. In contrast, ORCA introduces *iteration-level batching*, also known as *continuous batching*, which does not wait for all requests to complete before starting a new batch. Instead, it continuously schedules a new request when a request in the processing batch completes and GPU slots are available [35]. However, because it uses a non-preemptive scheduling method, head-of-line blocking issues may occur, making it unsuitable for interactive services using LLMs [26].

Priority Scheduling: SJF scheduling has the advantage of guaranteeing minimum average waiting time compared to FCFS, by scheduling tasks with shorter execution times first. The execution time of convolutional neural networks or encoder-based models like BERT can be easily predicted based on the model size [4, 13]. However, this observation does not hold true for LLMs due to the non-deterministic nature of the auto-regressive decoding phase included in LLMs. FastServe attempted to reduce the average JCT by utilizing a Multi-Level Feedback Queue (MLFQ)-based scheduling approach [33]. MLFQ is less efficient because it needs to adjust the priority of each task through trial and error, causing head-of-line blocking in the process. Qiu *et al.* [26], who presented the most similar work, used BERT to predict the output token length of LLMs, which was used for priority and scheduling requests. However, predictions were made only once, making it difficult to fully capture the auto-regressive characteristics of LLMs.

3 Motivation

In this section, we describe the main intuitions of our work and introduce the rationales behind our design choices.

3.1 Ability to Comprehend Natural Language

To predict both the content and length of the generated answers according to the context, it is crucial that the embedding model understand the context of prompts. Hence, we pose the question: *"Can the embedding model represent the context of the user prompts?"* and aim to provide a valid answer in this section.

In this paper, we deploy the BGE (BAAI/bge-base-en-v1.5) model [34] as the base architecture for our LLM response length predictor. The BGE model is the state-of-the-art on the Massive Text Embedding Benchmark leaderboard (ranked first in July 2024; ranked second in October 2024).

We validated the effectiveness of the BGE model through the following experiment. Using ChatGPT, we generated two datasets: one consisting of 100 semantically similar sentences and another consisting of 100 semantically unrelated sentences. The similar sentences focused on the topic of weather, while the unrelated sentences covered various topics not related to weather. To ensure there was no overlap in vocabulary or duplicate sentences between the two datasets, we applied thorough pre-processing steps, including removing any sentences with shared words or identical content. The CLS token vector of each sentence was then extracted

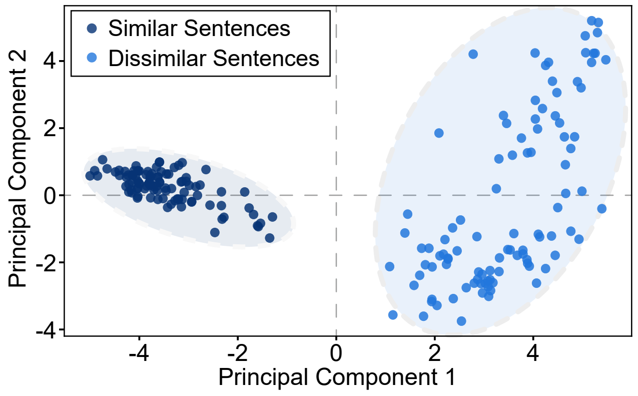

Figure 1. BGE CLS vector distance with different groups.

using the BGE model, resulting in 768-dimensional embeddings. These embeddings were reduced to two dimensions using principal component analysis for visualization.

Figure 1 illustrates that sentences with similar contexts (colored dark blue) are clustered closely together, whereas sentences with dissimilar contexts (colored light blue) are scattered farther apart. This observation confirms that the BGE model effectively captures the contextual information of sentences in the CLS token embeddings.

3.2 Capability of Predicting Response Length

We conducted the following experiment to assess the capability of BGE to predict response length:

The architecture of our response length prediction model consists of the pre-trained BGE model [34] and eight additional linear layers. In our research, we focused on the CLS token, which is known to contain significant input information [6]. The input prompt is processed by the BGE model to generate a CLS token embedding, which is then fed into the subsequent linear layers to predict the expected response length. The pre-trained parameters of the BGE model were frozen, and only the eight linear layers were trained during fine-tuning for output token length prediction.

We used the LMSYS-Chat-1M dataset for training [39], which was collected from the Vicuna demo and the Chatbot Arena website. This dataset contains one million chat records from 25 state-of-the-art LLMs, with 49% of the data generated by the Vicuna-13B model and the remainder from 24 other models. Since our focus in this section is to evaluate the feasibility of using BGE as a predictor, we did not apply any bias mitigation techniques.

Table 2 shows the results after fine-tuning the output length predictor with the LMSYS dataset. The evaluation metrics used were Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and R-squared (R^2). The fine-tuned BGE model demonstrated lower MAE and RMSE values compared to the

Table 2. BGE baseline prediction results.

Model	MAE	RMSE	R^2
Pre-trained BGE	175.99	224.98	-1.58
Fine-tuned BGE	71.48	101.29	0.48

pre-trained BGE, and its R^2 value was closer to one. Performance improvement was observed despite data imbalance and the use of default training parameters. This suggests that even higher performance could be achieved if the model were trained with data specifically designed for LLM scheduling. Qiu *et al.* found that when the accuracy of the output token predictor was 0.615, JCT decreased by 39% and throughput increased by 2.2 times [26].

Therefore, given that our fine-tuned BGE model achieves a high prediction accuracy, it is expected to provide significant performance benefits as an SRTF scheduler.

3.3 Improving Reliability with Iterative Prediction

The key intuition of this study is that, for LLMs, output is generated in *iterations* due to their auto-regressive nature. ORCA has designed iteration-level batching methods, such as continuous batching and in-flight batching, to accommodate this process. The question we pose in this section is: *"If the LLM delay predictor receives input in iterations, will the accuracy improve as more information is provided to the predictor incrementally?"*

Figure 2(a) depicts a graphical illustration of performing prediction in iterations. Let us assume that the generated output is 120 tokens in length when given a user prompt. In Step 1, the predictor should estimate an output length of 120 tokens, receiving only the user prompt as input. In Step 2, the predictor adjusts its estimate to 70 tokens, using both the prompt and the previously generated 50 tokens as input. As the process continues, the previously generated tokens are continuously fed back into the input, and the corresponding number of tokens is excluded from the output prediction, refining the estimate at each iteration. We have empirically determined that the optimal window size is 50 tokens through several experiments.

As shown in Figure 2(b), the MAE of the LLM latency predictor decreases as the iterations progress. Therefore, we conclude that the accuracy of our predictor can be improved when provided with partial output at every iteration.

3.4 Preemption and Real-world LLM Serving

Preemption is a widely studied scheduling method for ensuring priority among multiple tasks running on a system with limited resources. By preempting a lower-priority task and reallocating resources to higher-priority tasks, the higher-priority tasks are more likely to finish sooner. As a result,

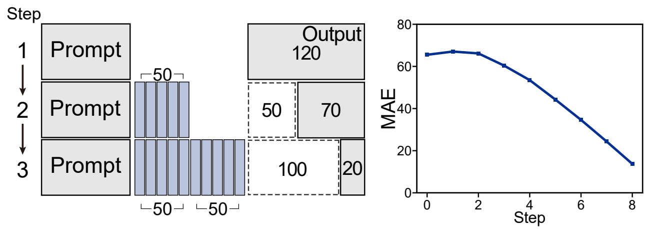

Figure 2. (a) Illustration of prediction procedure where each step (iteration) comprises of 50 tokens and (b) MAE of predictor for each step.

preemption has been widely researched and deployed in various machine learning systems [2, 16, 19, 29, 37], including vLLM, which is the inference engine used in ELIS.

According to our experimentation on vLLM with real-world prompts sampled from the LMSYS-Chat-1M dataset, we discovered that the probability of preempting a task due to limited resources is relatively low.

We conducted experiments on our prototype system to observe the minimum batch size that can induce preemption due to limited memory for storing the KV cache. These experiments were performed on an NVIDIA A100 GPU. Details and results of our experimentation are reported in Appendix A. LLaMA2-13B, being a relatively large model, has a higher likelihood of preemption and began preempting requests at a batch size of 120. Combining this with the average latency reported in Table 4, the average request rate required to constantly saturate this batch size is 13.9 requests per second ($= \frac{120}{8.61}$). Given that the maximum daily request rate observed from FabriX—an LLM-based service used by employees of Samsung SDS[2]—is below 3 requests per second, we can conclude that the probability of requiring preemption when serving LLM inference tasks is relatively low. Additionally, according to a survey on public Azure Machine Learning Service [7] trace data, the maximum rate did not exceed 2 requests per second. Hence, we did not include further experiments for preemption and instead focused on iterative priority scheduling. Nonetheless, we have designed and implemented policies that can adjust the frequency of preemption and prevent starvation for future research. All implementations will be included in the public code of ELIS.

4 Design

Figure 3 illustrates the overall architecture of ELIS, which consists of a request generator, a frontend scheduler that includes a predictor, and multiple backend workers. The following subsections provide a description of how each component interacts with the others and details regarding our predictor model.

[2]Organization will be disclosed in the final version of the paper.

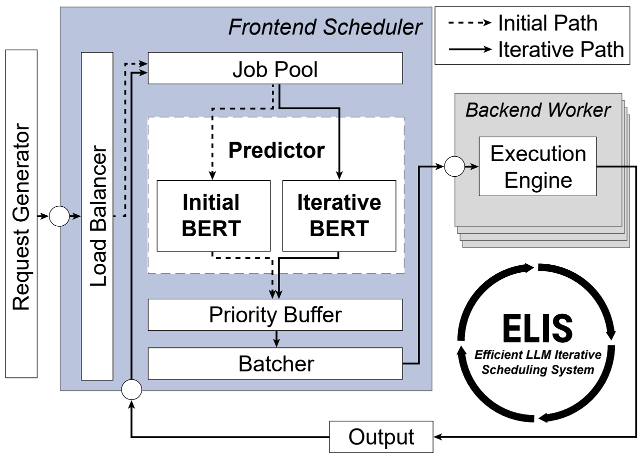

Figure 3. Overall architecture of ELIS.

Algorithm 1: Overall scheduling flow of ELIS.

```
Input: Set of Prompts $P$, Global State $G$
Output: Response of each prompt in $P$
1  for $prompt$ in $P$ do
2      store the text of $prompt$ in a new $job$;
3      $job.node \leftarrow LoadBalancer$.get_min_load($G$);
4      $JobPool$.push($job$);
5  end
6  for every $iteration$ do
7      if $JobPool$ is empty then
8          end $iteration$ loop;
9      end
10     for $job$ in $JobPool$ do
11         if $job.priority$ is none then
12             $job.priority \leftarrow Predictor$.init($job$);
13         else
14             $job.priority \leftarrow Predictor$.iter($job$);
15         end
16         pop $job$ from $JobPool$;
17         push $job$ to $PriorityBuffer$;
18     end
19     $BatchedPrompt \leftarrow Batcher$.batch($PriorityBuffer$);
20     $output \leftarrow BackendWorker$.exec($BatchedPrompt$);
21     for $job$ in $output$ do
22         if $job.response$ is finished then
23             return $job.response$;
24         else
25             add $response$ to $job.response$;
26             push $job$ back to $JobPool$;
27         end
28     end
29 end
```

4.1 Overall Scheduling Flow of ELIS

ELIS deploys a central frontend scheduler responsible for prioritizing prompts and assigning appropriate backend workers for execution. The scheduler makes scheduling decisions when new prompts arrive or when a scheduling iteration has completed and a new iteration begins. Each *scheduling iteration* processes *50* tokens of a batched prompt. Therefore, whenever a new decision is made, it influences the next iteration, which processes the next set of *50* tokens.

Scheduling Process: Algorithm 1 describes the overall scheduling procedure of ELIS, which receives a set of prompts P and returns the response for each prompt. Upon the arrival of a prompt, the frontend scheduler converts the prompt into a *job*, which is a data record managed internally by the scheduler *(line 2)*. The job is then assigned to a backend worker node by the load balancer. The load balancer greedily distributes the jobs among the worker processes. By consulting the global state G stored in the frontend, which includes the number of jobs running on each backend worker, the load balancer selects the worker executing the fewest number of jobs (*line 3*). After the backend worker is selected, the new job is pushed to the $JobPool$, which is a queue that stores jobs (*line 4*).

In each iteration, scheduling decisions are made until there are no jobs remaining in the $JobPool$ (*lines 6 to 9*). Based on the scheduling policy and whether a job's priority has been previously assigned, each job is given a priority and pushed to the $PriorityBuffer$ (*lines 10 to 18*). The $PriorityBuffer$ consists of multiple priority queues, where each queue stores jobs assigned to a specific *node*. Whenever a backend server becomes available, a batched prompt is formed, starting with the prompt with the highest priority (*line 19*). After forming a batched prompt, it is sent to the backend engine and executed for one window of *50* tokens (*line 20*). The frontend scheduler receives the *output* after one window, which includes a list of jobs and their corresponding *responses*. Each job in the *output* is checked to determine whether the response is complete (*lines 21 to 22*). If the *response* is complete, the full response is returned (*line 23*). If not, the partial response is stored in the job, and the job is pushed back to the $JobPool$ to be processed in the next scheduling iteration (*lines 24 to 26*).

Performance Optimization: We have implemented additional optimizations to ELIS to enhance performance and scalability. The scheduling process is divided into several sub-procedures, each executed asynchronously. Each procedure maintains a buffer for sharing and communicating data with the others.

To avoid overburdening the network, the input prompt of each job is sent to the backend only once, rather than being sent for every scheduling iteration. Batching is performed

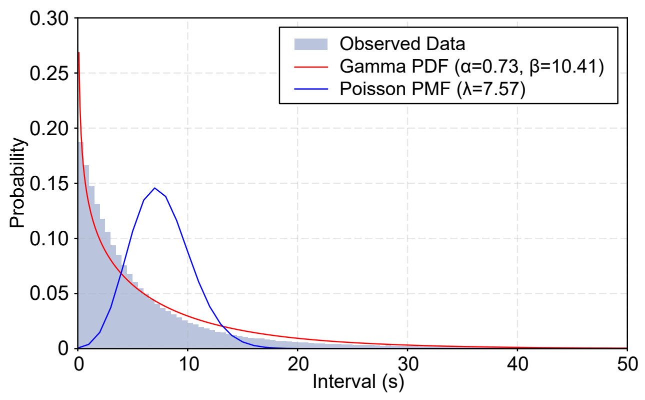

Figure 4. Request interval distribution of LLM serving. The Gamma PDF and Poisson PMF distributions were fitted based on the observed data.

using fixed window sizes of *50* tokens, and partial outputs are sent in these batches.

Real-World Request Analysis: Due to the lack of publicly available trace data for LLM services, most existing studies on LLM serving have assumed that request arrival times follow a Poisson process or have utilized Azure function trace data [15, 17, 26, 33]. However, in this study, we analyzed over 200,000 real-world trace data points gathered over two months from FabriX, an LLM service operated by Samsung SDS. We confirmed that the intervals between LLM requests follow a Gamma distribution more closely than a Poisson distribution, with shape parameter $\alpha = 0.73$ and scale parameter $\beta = 10.41$, as shown in Fig. 4. This result is similar to BurstGPT [7], which chose the Gamma distribution to better capture the burstiness of LLM service requests. In conclusion, we randomly sample requests from a Gamma distribution when evaluating ELIS in Section 6.

Backend Worker: Each backend worker of ELIS is responsible for relaying inputs to the inference engine, ensuring that the designated priority is maintained when executing input prompts and sending replies generated by the execution engine. In other words, the backend worker acts as a proxy to the inference engine. We chose vLLM [16] as our execution engine due to its popularity in both industry and academia.

To successfully control batched prompts in iterations, we have added two additional features to vLLM. The first feature is *iteration-wise execution*, which executes a batched prompt for K tokens (the window size of an iteration). A batched prompt passed to the vLLM engine is executed and returns partial responses when all prompts in the batch have produced K tokens or when a prompt has finished. The partial responses are sent to the frontend scheduler, and the backend worker checks if any pending commands have been sent by the frontend scheduler. If there are pending batches, the backend server checks and updates the next batch to be executed by vLLM. The second feature is *configurable priorities*, which is used to override the default priority of vLLM (FCFS). Whenever the backend server updates the next batch, the priority of each prompt is also updated. The priority assigned to each prompt affects the order in which tasks are preempted by vLLM. Whenever preemption is required, vLLM will check the priority of each task and preempt the tasks starting with the lowest priority. Please refer to Section 5 for additional details on how we have implemented each feature.

4.2 Response Length Prediction Model

In this study, we tested an online scenario by running vLLM as a backend worker with the ISRTF scheduler on Kubernetes. Output from 13 LLMs (listed in Table 7), including LLaMA1, LLaMA2, Vicuna, OPT, and GPT-NeoX, was collected by executing them on the vLLM framework. 11,000 prompts were randomly selected from the LMSYS-Chat-1M dataset, and the LLMs were run with the vLLM default settings to generate a total of 143,000 prompt-answer pairs. The collected data consist of model name, input tokens, output tokens, input token length, output token length, and execution time. During the dataset construction process, duplicate data were initially removed, and outliers were then removed using the Interquartile Range and log transformation methods. In this process, approximately 40,000 rows were removed, leaving 105,295 rows.

We treated the user's original prompt and the input consisting of the prompt attached with the answer as separate data. Therefore, step data for iteration prediction was prepared for each prompt. The entire dataset was shuffled and then divided into training, validation, and test sets in a 6:2:2 ratio, respectively.

The model architecture consists of the BGE model and eight Fully Connected (FC) layers. The CLS token and other token embeddings generated by the BGE model are passed through mean pooling, and the expected response length is predicted using eight FC layers. The activation function used was ReLU, and the hidden dimension of the FC layers was set to 1024. The learning rate used was 1×10^{-4}. Training was performed on an NVIDIA A100 GPU with a batch size of 16, and the loss converged after epoch 16. The results of the BGE fine-tuning on the vLLM dataset showed that the MAE was 19.923, the RMSE was 34.327, and the R^2 was 0.852. Additionally, the R^2 value increased by a factor of 1.78 compared to the fine-tuned BGE on the LMSYS dataset.

5 Implementation

System Software Architecture: As illustrated in Figure 3, our multi-GPU LLM serving system comprises a frontend scheduler server and multiple backend workers, each deploying a vLLM execution engine [16]. The serving system is developed in Python with approximately 3.1K lines of code, including both the frontend scheduler and backend worker implementations. The frontend scheduler consists of multiple software modules, each instantiated as a separate process. Each process collaborates through interprocess communication using shared memory, which stores the global state of each backend engine and the jobs running in the serving system. Each backend worker is responsible for executing inference for a specific model with vLLM, determined by loading a designated file (a pickled zip file with the extension `.pth`) when initiating the worker. We have made minor adjustments to the policy, scheduler, and entry points of vLLM (less than 100 lines of code) to execute batched prompts K tokens at a time and override the default preemption policy, which is FCFS. We have added a custom policy class to implement SRTF scheduling. This custom policy allows us to prioritize jobs based on their remaining execution time. The entry point was modified to return the request ID managed internally by vLLM. It is necessary for the worker to manage the mapping between jobs and request IDs, as the priority of each job will be updated using the internal request ID.

Deploying ELIS on Kubernetes: We have deployed our prototype as components within a Kubernetes cluster. The frontend scheduler and backend execution servers are deployed as pods within the Kubernetes cluster. Given the widespread use of Kubernetes for managing microservices, pods are treated as ephemeral resources that can be automatically replaced when necessary. However, each backend pod must be uniquely identifiable, as the frontend pod needs to communicate with the specific pod assigned to execute a batch. To achieve this, the backend pods are managed using a StatefulSet, ensuring each pod has a unique ID for proper identification and communication. Communication between the pods is facilitated through endpoints exposed as services within the Kubernetes environment. The YAML configuration files defining the Kubernetes components used in the experiments presented in this paper will be made publicly available.

6 Evaluation

6.1 Methodology

Experimental Setup: We have evaluated ELIS by deploying the prototype system on a Kubernetes cluster hosted on a server from our organization's private infrastructure. Table 3 provides the specifications of the server's software components and GPU.

To avoid network bottlenecks when generating requests from outside the cluster, the prompts are generated by the frontend scheduler. Nevertheless, we have included a stand-alone generator in our public code for future research.

Baseline Scheduling Algorithms: We have ported two priority schedulers into ELIS to compare performance. The two task scheduling algorithms are FCFS and SJF, with SJF serving as an oracle scheduler to indicate ideal performance. FCFS assigns priority based on the prompt's arrival time, while SJF assigns higher priority to prompts with shorter profiled latency. We compare the two schedulers above with our proposed scheduling method, ISRTF.

Table 3. The evaluated system specifications.

System Overview	
CPU	Dual AMD EPYC 7H12 64-core
GPU	8 NVIDIA A100
Memory Capacity	2 TB
Operating System	Ubuntu 20.04
CUDA	12.2
NVIDIA Driver	535.183
ML framework	vLLM v0.5.0
GPU Specification	
CUDA Cores	6,912
Memory Capacity	80 GB HBM2
Memory Bandwidth	2,039 GB/sec

Table 4. List of LLM models (and abbreviation) used in the evaluation.

Model	Parameter Size	AVG. Latency (ms)
OPT-6.7B (opt6.7) [38]	6.7B	1315.5
OPT-13B (opt13) [38]	13B	2643.2
LlaMA2-7B (lam7) [31]	7B	6522.2
LlaMA2-13B (lam13) [31]	13B	8610.2
Vicuna-13B (vic) [40]	13B	2964.9

Large Language Models: We have used the models listed in Table 4 for evaluation. We have prepared **five** LLMs and report the average latency of 500 prompts executed on an NVIDIA A100 GPU.

Simulated Workload: To evaluate ELIS under various workloads, we simulate streams of prompts. The prompts are sampled from the LMSYS dataset [39]. The interval between prompts is randomly sampled from a Gamma distribution in accordance with the conclusion of Section 4.1.

6.2 LLM Serving Performance Comparison

This section compares how well *ISRTF* performs compared to other schedulers, FCFS and SJF.

JCT Comparison: We measured the average JCT to evaluate the performance. To be more specific, JCT is measured from the point of arrival in the frontend scheduler to the point when the prompt's response has been completely formed and stored in the frontend scheduler. We average the JCT of 200 prompts sampled from LMSYS-Chat-1M data. To provide a fair comparison, we use the same set of sampled prompts but randomly shuffle them for each experiment and repeat this process for three iterations. Each experiment is conducted with a multiple of the average request rate, which is

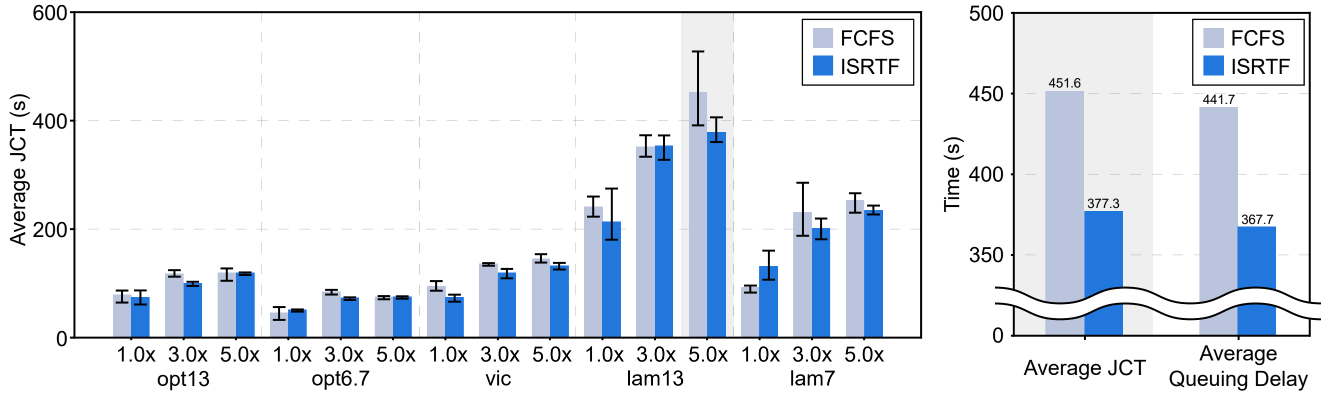

Figure 5. (LEFT) *JCT* comparison between FCFS and ISRTF where each experiment uses a multiple of average throughput. Bar represents the average value and each tick represents the minimum and the maximum value of each experiment. (RIGHT) Average *JCT* and queuing delay of lam13 with 5.0x RPS (case highlighted in gray shading).

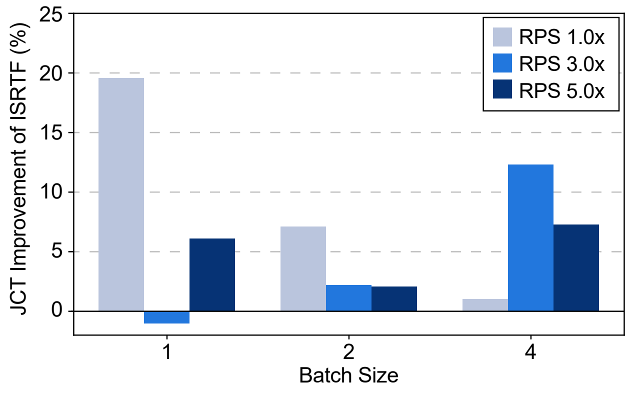

Figure 6. *JCT* improvement of ISRTF over FCFS.

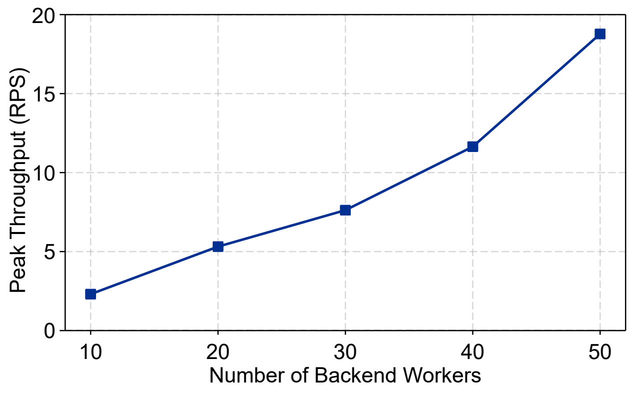

Figure 7. Peak request rate where the average queuing delay of each worker does not exceed *0.5 s* with different number of backend workers.

calculated with the equation listed below (using the average latency listed in Table 4):

$$AVG.RequestRate = \{\frac{1000}{AVG.Latency}\} \times batchsize$$

Figure 5 reports the minimum, average, and maximum *JCT* of each experiment. For almost all cases, *ISRTF* has shown enhanced performance by an average of 7.36% and a maximum of 21.40%. This is mostly due to our predictor assigning higher priority to prompts with fewer remaining tokens. By executing prompts with fewer remaining tokens earlier, ELIS prevents head-of-line blocking as much as possible, reducing queuing delay in the process. The ideal performance delivered by SJF is listed in Table 5.

Deeper Look into Performance Advantages: We take a deeper look into a certain case to showcase the main cause of the performance advantage of our result. The right-hand side figure in Fig. 5 reports the average *JCT* and the average queuing delay of the best case (which is the performance of LlaMA2-13B with a 5.0x RPS.) reported in the left-hand figure, The average *JCT* of *ISRTF* is 16.45% lower than that of FCFS. Additionally, the queuing delay of *ISRTF* is 16.75% smaller than the delay of FCFS. There is only a 0.30% difference between the two numbers. Hence, we can conclude that the major cause of reduced *JCT* is reduced queuing delay.

Additionally, the average scheduling overhead, including batching and the initial BERT predictor, is only 11.04 ms. Given the fact that the average latency of LlaMA2-13B (lam13) is 8610.2 ms, the overhead is marginal, accounting for only 0.13%. Therefore, it is safe to conclude that the advantage of reduced queuing delay outweighs the performance penalty introduced by additional scheduling overhead.

6.3 JCT Improvement over Different Batch Sizes

To provide a comprehensive evaluation, we perform the same experiment introduced in Section 6.2 with smaller batch sizes, *1* and *2*. Due to limited space, we omit detailed results but instead provide summarized performance of ISRTF for each batch size and RPS multiple compared to FCFS.

Figure 6 reports the amount of improvement (in percentage) compared to the performance of FCFS in terms of average *JCT* of *ISRTF*. For example, since the *JCT* enhancement for batch size 1 on RPS 1.0x is 19.58%, the *JCT* of *ISRTF* will be 4.90 seconds less if the *JCT* of FCFS is 25 seconds. For all (except one) setups, *ISRTF* has outperformed FCFS. We have observed that for experiments with lower batch sizes and high RPS multiples, the chances of *ISRTF* reporting higher *JCT* increase. This phenomenon is due to the larger number of queued prompts mitigating the effect of priority scheduling and making the performance more dependent on the system's throughput itself.

6.4 Scalability Evaluation

To evaluate whether the scheduler and load balancer of ELIS can successfully scale the number of backend workers, we measure the performance of our system by varying the number of workers and incoming request rate. We measure and report the *peak throughput*, which is the maximum request rate where the average queuing delay of each request does not exceed 0.5 seconds. Unlike conventional evaluating methods for *peak throughput*, we do not use a direct comparison of latency (i.e., average or P99) when measuring the overhead. This is because the latency of each prompt is not fixed and changes due to the auto-regressive nature of the decoding process. Instead, we use the queuing delay as an indicator of scalability since the delay includes the overhead of choosing among multiple workers and waiting time caused by a limited number of backend workers. Each backend worker handles requests with a maximum batch size of 4 for the LlaMA2-13B model, and the *ISRTF* scheduler was used for evaluation.

To assess the scalability of ELIS beyond our limited experimentation environment, we have experimented with ELIS on a larger cluster with NVIDIA H100 GPUs and we deploy one worker per GPU. Figure 7 reports the achieved *peak throughput* for each number of backend workers. We have increased the number of backend workers up to 50 by incrementing 10 workers at a time. Starting from 2.31 RPS for 10 backend workers, ELIS displayed near-linear scaling performance, achieving 18.77 RPS for 50 workers. This is due to two reasons: The first reason is that the load balancer successfully distributed prompts among multiple workers in an efficient manner. The second reason is that the sub-procedures of scheduling are executed asynchronously, as described in Section 4.1, giving a boost to scalability.

Table 5. Avg *JCT* (s) of each model and scheduling method. Experimented on NVIDIA A100 GPUs with batch size of 4.

Model	RPS	FCFS	ISRTF	SJF
opt13	1.0x	77.83	73.57	20.35
	3.0x	116.46	98.74	43.63
	5.0x	118.13	118.11	43.63
opt6.7	1.0x	45.08	50.52	13.21
	3.0x	83.42	72.33	24.62
	5.0x	73.93	74.41	31.91
vic	1.0x	93.42	73.43	32.34
	3.0x	134.96	118.22	58.39
	5.0x	144.23	131.38	60.98
lam13	1.0x	240.25	212.60	70.55
	3.0x	350.55	352.53	133.11
	5.0x	451.59	377.29	125.59
lam7	1.0x	91.28	130.71	37.02
	3.0x	229.64	200.34	59.37
	5.0x	251.66	234.08	89.64

7 Related Work

Over the years, numerous machine learning serving systems have been introduced in both industry and academia. We provide a brief overview of this research and highlight how ELIS differs from each of them.

LLM Response Prediction: We first introduce contemporary works related to predicting the response time of LLMs, starting with studies that have also used BERT models. S3 trained a DistilBERT model to predict the priority of each task [15]. Qiu *et al.* also conducted research on predicting the remaining number of tokens by leveraging the natural language understanding capability of BERT models [26]. While S3 and Qiu *et al.* both utilize BERT models, both studies rely on a single prediction and do not provide any backup plans when the prediction is wrong. In contrast, ELIS predicts priority iteratively and displays stable predictions as each iteration proceeds.

Other studies have also been presented which predict priorities without leveraging a BERT model. FastServe presented a priority scheduler by deploying a MLFQ and prioritizes tasks with the shortest remaining time in an adaptive manner [33]. Our work differs from FastServe since we predict the priority rather than adapting and adjusting the priority of a task. By predicting the priority, we save the overhead caused by the process of finding the priority. PiA performed instruction-tuning on an LLM by instructing it to predict the response length before generating a response [41]. Instruction-tuning LLMs may lead to decreased performance and accuracy; in several cases, it has been observed that the length of the actual response is limited to the initial prediction. This is not an issue for ELIS since we do not use the same model for predicting and generating the response.

LLM Serving Systems: We also provide a brief comparison to works related to LLM serving that focus on topics other than response prediction. Several papers have presented how to optimize the use of the KV cache [12, 16, 18], while other papers focus on how to load-balance the workload for each worker node in a distributed environment [11, 22, 30]. Additionally, a number of studies have researched how to improve performance by efficiently decoupling the prefill and decode phases [1, 21, 24]. However, the works mentioned above lack one or more of the main contributions of ELIS: (1) scaling to multiple nodes, (2) considering preemption, and (3) analyzing real-world data.

Optimizing Machine Learning Inference: Although our work focuses on LLMs, we introduce several categories of research related to optimizing the inference process of general machine learning models. Such previous works have inspired our work and can be categorized into: scaling a serving system in a cloud environment [27, 36]; reducing the problem space of serving heterogeneous models and goals [23, 28]; researching various parallelisms [5, 20, 25]; and leveraging special features provided by the GPU [3, 8, 14].

8 Conclusions

We present ELIS, an **E**fficient **L**LM **I**terative **S**cheduling system, which leverages the natural language understanding capabilities and relatively low computation cost of BERT model for priority scheduling. Based on the predictions of our model, we prioritize tasks that have fewer predicted remaining tokens to minimize head-of-line blocking and enhance performance. Not only have we tailored a BERT model for prediction, but we also provide an analysis of real-world data obtained by observing requests in FabriX. The prototype system is deployed as Kubernetes components, a widely used container orchestration platform in cloud industry, to facilitate future research and deployment of our system. Experimental results demonstrate that the iterative scheduling method deployed in ELIS reduces average JCT by up to 19.6%.

9 Acknowledgments

We extend our gratitude to the Samsung SDS AI research team for their advice on model training and Samsung SDS FabriX team for providing actual LLM service data.

References

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

[2] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, Carlsbad, CA, July 2022. USENIX Association.

[4] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline:latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[5] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert:pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Burstgpt: A real-world workload dataset to optimize llm serving systems. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 492–506, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Abhimany Dubey et al. The llama 3 herd of models. *arXiv preprint https://arxiv.org/pdf/2407.21783*, 2024.

[10] Tom B Brown et al. Language models are few-shot learners. *Neural Information Processing Systems*, 2020.

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, Santa Clara, CA, July 2024. USENIX Association.

[12] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, Santa Clara, CA, July 2024. USENIX Association.

[13] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Max planck, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[14] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 249–265, New York, NY, USA, 2023. Association for Computing Machinery.

[15] 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*, 2023.

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

[17] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with

pagedattention. In *In 29th Symposium on Operating Systems Principles*, 2023.

[18] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association.

[19] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 835–850, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[21] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 932–949, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1112–1127, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[24] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for llm inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 369–384, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[26] 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. In *The 5th International Workshop on Cloud Intelligence / AIOps at ASPLOS 2024*, volume 5, pages 1–7, San Diego, CA, USA, 2024. Association for Computing Machinery.

[27] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[28] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1075–1092, New York, NY, USA, 2024. Association for Computing Machinery.

[30] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, Santa Clara, CA, July 2024. USENIX Association.

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

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *Operating Systems Design and Implementation*, 2024.

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

[34] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. C-pack: Packaged resources to advance general chinese embedding, 2023.

[35] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[36] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[37] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[38] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

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

[40] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P.

Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

[41] Zangwei Zheng, Xiaozhe Ren, Ynag Luo Fuzhao Xue, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Appendix A Preemption Profiling

We present our profiling data obtained from investigating the probability of preemption due to limited memory space. This section includes the profiling setup and the minimum batch size that causes a preemption. All profiling was conducted on our prototype ELIS backed by NVIDIA A100 GPUs. We profiled each model listed in Table 4 with 10K prompts that were sampled from the LMSYS-Chat-1M dataset [39]. To simulate an environment where there are sufficient prompts to form large batches, we saturated the job pool in the frontend scheduler with a high request rate (10K requests per second). We gradually increased the batch size by 10, up to a maximum of 250, until we observed a preemption. If a preemption was not observed, we reduced the memory utilization of vLLM (from the default 90%) and repeated the process. All profiling results are reported in Table 6.

Table 6. Profiling results for preemption. *Batch size* represents the minimum batch size at which a preemption has occurred.

Model Name	Batch Size	vLLM Memory Limit
LlaMA2-13B	120	90%
LlaMA2-7B	40	30%
OPT-7B	30	40%
OPT-13B	60	40%
Vicuna-13B	90	40%

Appendix B Models Used for Training

We provide the list of models used for training the predictor presented in Section 4.2. Table 7 reports the size of each model and the organization that produced it.

Table 7. List of models for training.

Model Name	Size (B)	Producer
LlaMA-7B	7	Meta
LlaMA-13B	13	Meta
LlaMA2-7B	7	Huggyllama
LlaMA2-13B	13	Huggyllama
Vicuna-7B	7	LMSYS
Vicuna-13B	13	LMSYS
OPT-1B	1.3	Facebook
OPT-3B	2.7	Facebook
OPT-7B	6.7	Facebook
OPT-13B	13	Facebook
GPT-NeoX	20	EleutherAI
Gemma	7	Google
SOLAR	11	Upstage