# Justitia: Fair and Efficient Scheduling for LLM Applications

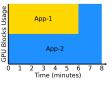
Mingyan Yang\*, Guanjie Wang\*, Manqi Luo, Yifei Liu, Chen Chen, Han Zhao, Yu Feng, Quan Chen, Minyi Guo Shanghai Jiao Tong University

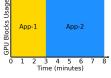
Abstract-In the era of Large Language Models (LLMs), it has been popular to launch a series of LLM inferences—we call an LLM application—to better solve real-world problems. When serving those applications in shared GPU servers, the schedulers are expected to attain fast application completions with guaranteed worst-case performance. However, mainstream LLM schedulers fail to behave well for LLM applications—due to head-of-line blocking or over-constrained resource allocation. In this paper, we propose to serve LLM applications in a fair and also efficient manner. To this end, we design Justitia, a novel scheduler with three key techniques. First, given that memory is prevalently a bottleneck for mainstream inference frameworks like vLLM, Justitia models the service cost of LLM applications in a memory-centric manner. Meanwhile, it uses a simple neural network model to conduct light-weight and also accurate demand prediction. Moreover, Justitia adopts a virtualtime based fair queuing algorithm to reduce the overall performance with guaranteed worst-case delay. We have implemented Justitia atop vLLM, and experimental results involving diverse LLM applications show that it can substantially enhance the scheduling efficiency with fairness preserved.

## I. INTRODUCTION

The recent years have witnessed the boom of large language models (LLMs) [1] in revolutionizing various fields [2], [3], [4]. In particular, for enlarged input size or enhanced output quality, it has become increasingly popular to address a real-world problem with a set of correlated LLM inferences [5], [6], which we call an LLM application. For example, to summarize a very large file, multiple parallel inference tasks would be launched each processing one file partition [7]; to solve a complex mathematical problem, multiple searching directions need to be explored until a satisfiable answer is obtained [8]. Such LLM applications are prevalently served in shared GPU clusters, where users usually expect to get the final output as early as possible [7], [9], without being remarkably delayed by competing users [10], [11].

However, existing LLM serving systems fail to behave well for scheduling LLM applications. For example, the mainstream LLM framework, vLLM [12], schedules LLM inference requests in a First-Come-First-Serve (FCFS) manner, which suffers from the head-of-line-blocking problem. Such a problem also exists for another recently-proposed scheduler, Parrot [7], which applies FCFS at the application level. Meanwhile, to ensure fairness, the VTC scheduler [10] is proposed to fairly allocate the serving resources among different users. However, this method forces each application to use its fair





(a) Concurrent Serving

(b) Sequential Serving

Fig. 1: Given two competing LLM applications, serving them sequentially in a saturated manner can reduce the average completion time with no per-application delay.

share in contention, resulting in postponed completion. It is thus desirable to devise a scheduler particularly for LLM applications that can behave well in both *efficiency* (i.e., short average completion time) and *fairness* (i.e., guarantee that no application be delayed by the mis-behaviors of others).

To attain efficient and also fair serving, our insight is that LLM applications shall be served in a *saturated* manner following the *fair completion order*. We first note that, regarding fairness, users primarily care about the *long-term* fairness (i.e., with guaranteed fast completion) instead of *short-term* fairness (i.e., always allocated an equal share of the total resources). We thus can trade such short-term fairness for higher efficiency. As shown in Fig. 1, when two applications are competing with each other, serving them—not *concurrently with the fair share*—but *sequentially in the fair completion order with all the resources*—can reduce the average completion time, with no application actually delayed. With such a serving manner—we call *saturated serving in fair completion order*, we can potentially do well in both fairness and efficiency.

Yet, it is a non-trivial task to apply the above insight into practice, which requires estimating the service cost of LLM applications and calculating their fair completion order. To be specific, the challenges are *three-fold*. First, LLM inferences consume both compute and memory (i.e., KV cache) resources on GPU servers, and the resource consumption amount is not a constant (due to the auto-regressive generation manner); such resource heterogeneity and volume-irregularity shall be properly handled when depicting an LLM application's service cost. Second, the overall service cost of an LLM application needs to be predicted upon its arrival; such prediction shall be made accurate and also efficient. Third, upon the arrival or completion of any application, we need to work out the expected completion order of all the pending applications under a fair scheduler; such decisions shall be made efficiently

<sup>\*</sup>Equal contribution.

(without frequent refreshing later) and also of high quality (yielding worst-case performance guarantee).

In this paper, we propose *Justitia*, a fair and efficient scheduler for LLM applications. In modeling the service cost of an LLM application, we find that the GPU memory is usually the true bottleneck; therefore, rather than adopting computecentric cost modeling as in [10], we propose memory-centric cost modeling, which takes into consideration both the space and time of memory occupation. In demand prediction, we adopt the Multi-Level Perception model, which is light-weight (in terms of the training and inference cost) and also accurate (by leveraging the application-specific demand similarity). In determining the application queuing order, we borrow the virtual-time based fair queuing algorithm originally proposed for packet scheduling in the network community [13], [14]. That algorithm can efficiently calculate the application completion order under an idealized fair scheduler-in one shot with no need to consider later-arrival ones. Moreover, we theoretically prove that, the maximal delay an application could encounter under Justitia—when compared to under the idealized fair scheduler—is always bounded by a constant.

We have implemented Justitia above vLLM [12], the mainstream LLM serving framework. Given a set of diverse LLM applications, Justitia can remarkably reduce the average completion time compared to existing schedulers. Specifically, compare to the state-of-the-art fair scheduler, VTC [10], our Justitia scheduler can reduce the average application completion time by 57.5%—with only 8% delayed at a slight level. Moreover, further studies confirm that it is indeed indispensable to adopt memory-centric service cost modeling and MLP-based demand prediction for LLM applications; besides, the scheduling overhead of Justitia is also negligible.

## II. BACKGROUND

# A. LLM Application Serving: The Basics

LLM inference: an auto-regressive process relying on KVcache. Large language models (LLMs) have demonstrated strong capability for diverse tasks like text processing [15], code generation [3] and task planning [16]. Leveraging LLM intelligence requires conducting auto-regressive LLM inferences. Due to the compute-intensive nature, LLM inferences are typically conducted with cutting-edge accelerators like GPUs, and for high GPU utilization, multiple input sequences are batched together and processed in an iterative manner [17]. In particular, to avoid repeated computations, the intermediate feature states of the generated tokens are cached for later usage during the inference process (called KV cache) [18]. In mainstream inference frameworks like vLLM [12], such KV cache is accommodated within a designated GPU memory space of limited size, which bounds the maximum number of sequences that can be concurrently processed in each iteration.

**LLM applications: the emerging trend and the scheduling objectives.** To apply LLMs for real-world problems with enhanced processing capability and output quality, there recently emerges a trend to organize multiple LLM inferences

together—in the form of LLM applications. For example, as shown in Fig. 2a, since the context window size of an LLM inference is limited (e.g., 10M tokens [19]), to summarize a large text file, multiple parallel inferences—each processing a slice of the original file-need to be launched and finally have their results merged [7]. Similarly, in Fig. 2b, to improve the merging quality of multiple documents, several merging inferences are submitted in parallel—each followed by a scoring inference—and finally the highest-score result is selected [20]. In Fig. 2c, to address the hallucination problem [21], each claim within the LLM-generated output needs to be additionally verified by a dedicated LLM inference (e.g., with the FacTool framework [5]). In Fig. 2d, to solve complex mathematical problems, sophisticated LLM reasoning algorithms like self-consistency (SC) [6] are increasingly popular, which works by expanding multiple reasoning trajectories and applying majority voting to find the best answer.

LLM applications have substantially unleashed the potential of LLMs in reliably solving realistic problems, and would potentially become a dominant cloud workload. For such LLM applications, the final output desired by the end-users is only available when all the inference requests constituting the LLM application complete. Therefore, when serving those LLM applications in a GPU cluster, it is of paramount significance to reduce their *end-to-end execution time* [7]. In the meantime, since different applications—usually submitted by different users—would compete for the limited processing capability on GPU servers, it is also necessary to ensure service fairness among LLM applications, so as to avoid negative interferences [10] (e.g., head-of-line-blocking or starvation). In summary, attaining *efficient* and also *fair* scheduling is critical for serving LLM applications in the clusters.

## B. Related Works and Their Limitations

Regarding LLM inference serving, a series of scheduling methods have been proposed recently, in both *inter-inference* and *inter-application* level. Here we revisit such scheduling practices and summarize their limitations.

Inference-level Scheduling. Given that the ultimate output length of the auto-regressive generation process is non-deterministic a priori, mainstream LLM serving frameworks like vLLM [12] and SGLang [22] commonly adopt the *First-Come-First-Serve* (FCFS) scheduling algorithm—at the inference level without awareness to the existence of LLM applications. They schedule each incoming inference request based on its arrival time. Such a FCFS scheduling method suffers the head-of-line-blocking problem, meaning that a long inference would impede the execution of those behind it. Another work FastServe [23] employs multi-level-feedback-queue to address this problem, although frequent queue switching would bring non-negligible overhead. Recently, there also emerge some prediction-based methods that seek to enforce Shortest-Job-First in request scheduling [24], [25], [26].

However, all such inference-level scheduling methods fail to provide *end-to-end* performance optimizations for LLM

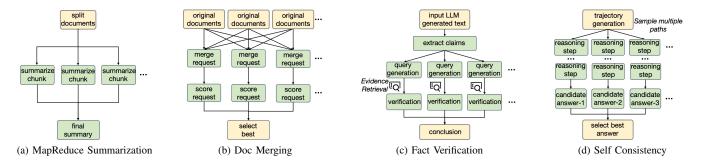


Fig. 2: Examples of typical LLM applications.

applications: in contention-intensive scenarios, the constituting inferences of an LLM application may be served in an interleaved manner, compromising the ultimate application completion time [7].

Application-level Scheduling. Given the deficiency of inference-level request scheduling, recent works have also noticed the need to optimize application-level scheduling performance. In that regard, Parrot [7] devises novel programming abstractions for LLM applications, which supports executing the correlated inferences in a non-interleaved manner; yet it determines the inter-application scheduling order still following the FCFS policy, thus also suffering head-of-lineblocking. Noticing the need to ensure inter-application (tenant) fairness, Sheng et al. proposed a fair scheduling algorithm called Virtual Token Counter (VTC) [10], which tracks the services received for each tenant and prioritize the ones with the least services received (a similar method is used by another work FairServe [11]). However, by enforcing each tenant (application) to only use its fair share, the end-to-end application completion time—which users truly care aboutwould be delayed compared to the monopolizing cases.

To summarize, existing LLM scheduling methods fail to attain efficient and fair scheduling at the application level, hurting the ultimate user experience. Our objective in this paper is thus to make up that research gap by designing a fair and also efficient scheduler specifically for LLM applications. On the one hand, we want to optimize application-level scheduling efficiency by reducing the average application completion time; on the other hand, we want to provide service guarantee on the worst-case service degradation any application could experience compared to the case where it is allocated an equal resource share.

## III. MOTIVATION

# A. Insight

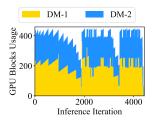
**Short-term fairness or long-term fairness?** We note that the fairness-centric schedulers [10], [11] are inefficient essentially because they seek to fairly share the computing resources at each instant moment, i.e., stick to *short-term fairness*. However, in expecting service fairness, users primarily desire a service guarantee on the worst-case completion time at the

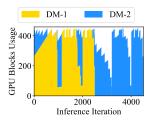
application level (as explained in Sec. II-A, the application completion time is what users truly care about). In that sense, long-term fairness (i.e., with performance guarantee on the application completion time) would suffice for end users (as echoed by existing works adopting the fairness metric called *finish-time fairness* [27]). We thus materialize out fairness objective as long-term fairness, which, as we show next, allows for more scheduling flexibility to attain higher efficiency.

Saturated serving in fair completion order. For a set of competing LLM applications, instantaneous fair sharing would limit the usable resources of each application to only the average share. Instead, we can prioritize the applications one by one based on their relative completion order under fair sharing, allowing each prioritized application to use unlimited resources it desires. In this way, because a prioritized application can complete faster, the average application completion time can potentially be reduced. In particular, if all applications arrive at the same time, we would be essentially mimicking the shortest-job-first (SJF) scheduling, which is known to be efficiency-optimal. Meanwhile, this method can also behave well in the fairness aspect: since the prioritized applications would yield their resources also earlier, and the faircompletion-order can prevent head-of-line blocking or infinite starvation, each application may still finish no later than its expected completion time under fair sharing. To summarize, it is promising that we can simultaneously attain high efficiency and long-term fairness. We call such a serving methodology saturated serving in fair completion order.

To verify the effectiveness of the above insight, we conduct a testbed experiment as shown in Fig. 3. We submitted two DocMerging applications simultaneously to an LLaMA2-7B model deployed on a single A100 GPU, with a total GPU block number of 459. As shown in Fig. 3a, under instantaneous fair sharing, each application is restricted to its fair share (as indicated by the KV block usage), resulting in an average JCT of 210 s. However, if we prioritize the applications sequentially (based on the job completion order in Fig. 3b), the average JCT drops to 166 s—without delaying any single application compared to the fair-sharing case in Fig. 3a.

**Predictability of application-level resource demands.** Note that to attain the previous scheduling effect, we need to





- (a) Instantaneous Fair Sharing
- (b) Saturated Serving in Fair Completion Order

Fig. 3: KV block usage when running two DocMerging (DM) applications under different scheduling schemes.

estimate the application completion order prior to application execution. While existing serving methods [7], [10], [11], [12] treat the LLM inference duration as a blackbox, we find that would be too conservative for LLM applications.

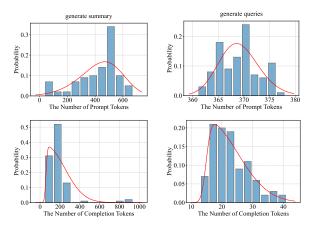
In fact, we find it is indeed promising to realize demandaware application scheduling. Since each LLM application is designed for specific use cases, the resource demands of which are relatively stable across different execution runs. Fig. 4 shows the execution information (input/output length) of specific inference stages in the two LLM applications over 100 trial runs, which exhibits strong demand correlations. For example, generate-queries inferences in Fact-Verification application all have an input token length between 360 and 380. Meanwhile, since the LLM applications are often hosted on the cloud as a public service [28], the applicationspecific execution information previously profiled would be readily available to the service providers. In that sense, the application viewpoint can bring promising opportunities for demand estimation. Moreover, our previous scheduling insight can bear a certain level of demand prediction errors, as long as the resultant scheduling order is not affected.

## B. Challenges

While it is promising to attain fair and efficient scheduling for LLM applications with saturated serving in fair completion order, when applying that insight in practice, there nonetheless exist several critical challenges.

First, how to *quantify the service cost of an LLM application*? Serving LLM workloads require both compute and memory (e.g., KV cache space as explained in Sec. II-A) resources, it is unclear which perspective is more suitable for demand depiction. Meanwhile, due to the auto-regressive nature (i.e., token number keeps increasing), the demand volumes on both resource types keep expanding. Therefore, we need to properly address such type-heterogeneity and volume-irregularity in resource demand when modeling the overall service cost of an LLM application.

Second, how to make application-level demand prediction as accurately as possible? While Fig. 4 previously suggests that there exists certain similarity among the resource demands in different trial runs of a given application, merely using such historical information is not sufficient for accurate demand



## (a) Summarization

# (b) Fact Verification

Fig. 4: Prompt and decoding length distribution respectively for the generate-summary and generate-queries inferences in the MapReduce-Summarization application and the Fact-Verification application. In each case, we divide the length range into 10 buckets, and calculate the value appearance probability in each bucket (accompanied by the fitted curves assuming skewed Guassian distribution reference).

prediction in the ongoing run. In fact, the runtime application input also crucially affects the overall resource demand in the current execution round. For example, for the SC application [6], a longer question input is usually more difficult to solve, and the resultant application serving duration would also be longer. Ignorance to such runtime hints would render the demand prediction as well as the completion order estimation inaccurate; meanwhile, such predictions must be also made light-weight.

Third, how to efficiently make high-quality decisions on the application queuing order, so as to reduce the average application completion time while theoretically guaranteeing the worst-case performance? Enforcing our previous insight in Sec. III-A requires obtaining—when each application arrives—its expected completion time under a fair scheduler, which is hard to make given that the completion time of an LLM application may be affected by the later-arriving ones. Moreover, to make our solution generally applicable, it is desirable to provide a formal performance guarantee on the worst-case performance impact of any application.

# IV. SOLUTION

In this section, we present *Justitia*, a fair and also efficient scheduler for LLM applications, with all the above challenges addressed. We first introduce our cost modeling method in Sec. IV-A, and then elaborate how to predict the application-level resource demand in Sec. IV-B. Finally we describe the queuing strategy of Justitia in Sec. IV-C and theoretically analyze its fairness properties in Sec. IV-D.

# A. Memory-Centric Cost Modeling

To determine the application scheduling order, we need to quantify the overall serving cost of each LLM application, and

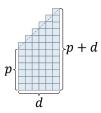


Fig. 5: Demand modeling of an LLM inference. p and d are respectively the prefill and decoding token length. The service cost is then depicted as the accumulative KV cache occupation (i.e., KV token-time):  $pd + \frac{d^2}{2}$ .

the key is to describe the serving cost of an LLM inference.

Given that both computation and memory resources are demanded in LLM inference process, its serving cost can be captured from either the *computation* perspective or the *memory* perspective. The VTC work [10] chooses the former: it measures the serving cost as a weighted combination of the prefilling (input) tokens and decoding (output) tokens. However, such a method is over-simplified by ignoring the impact of KV cache consumptions. In fact, in modern LLM serving frameworks like vLLM [12], the inference throughput is bounded commonly on the GPU memory (*KV cache space*): new sequences can always be added to the *running* queue as long as there are sufficient KV cache space; otherwise some running sequences would be interrupted and moved to the *swapped* queue. In that sense, inference serving cost shall be better depicted from the *memory* perspective.

Moreover, the resource demand of an LLM inference is not fixed during its lifetime. During the generative inference process, the sequence length keeps increasing, leading to increasing KV-cache occupations. Therefore, in this paper we propose a *memory-centric* cost modeling method that depicts the cumulative service cost of an LLM inference in both temporal and spatial dimensions. To be specific, let p and d respectively represent the prefill and decode token length, then we devise the cost metric, KV token-time, as:

$$c = \sum_{i=1}^{d} (p+i) = pd + \frac{d^2}{2}.$$
 (1)

This formula adds up the KV cache occupation<sup>1</sup> over all the iterations<sup>2</sup> of that inference. From the above formula, we can learn that the relationship between the cost volume and the generation length (d) is indeed *quadratic*. This is more reasonable compared to the *linear* relationship in the cost model of VTC [10] given the inflation effect of d on both memory occupation size and duration.

<sup>1</sup>The unit of the KV cache occupation is the number of KV cache blocks corresponded to one token (over all the LLM layers and heads), which is fixed for a given LLM. This is more concise for analysis than recording the exact KV block number. Hereafter in this paper, we describe the total KV cache space also in such units.

<sup>2</sup>For simplicity we do not consider the time difference when generating different tokens. In realistic execution, an LLM sequence would be batched with sequences from other inference requests [17]; all such sequences jointly determine the per-iteration inference latency. The inference time of such runtime batches with mixed sequence is statistically stable.

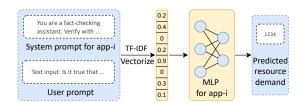


Fig. 6: Predict the resource demand from the input prompt with MLP model.

Furthermore, the overall serving cost of an LLM application can be naturally defined as the sum of the KV token-time of all its constituting inferences. Such a memory-centric modeling method can express the true serving cost of LLM applications.

#### B. MLP-based Demand Prediction

Given the above cost modeling method, to support our previous insight of "saturated serving in fair completion order", we need to predict the cost volume of each application. Recently, there are some attempts [29], [30] for demand prediction of LLM workloads. For example, the  $S^3$  [30] method proposes to fine-tune a language model, Distillbert, to predict the output length range given the inference prompt. However, simply adopting such a method for application cost prediction is inappropriate in efficiency and accuracy. First, fine-tuning the Distillbert model (with 66 million parameters) is time-consuming and requires collecting a large number of application execution samples, which is hard to obtain in practice. Meanwhile, using the Distillbert model for prediction—which is essentially yet another LLM inference—would incur non-negligible runtime overhead. Moreover, the  $S^3$ -like method uses a single model to predict all the workloads, yet different applications may exhibit heterogeneous cost distribution patterns (as shown in Fig. 4), rendering the unified prediction scheme inaccurate.

Therefore, in this paper we seek to devise a light-weight and also sufficiently-accurate cost prediction method for LLM applications. For each application type, we maintain a Multi-Layer Perceptron (MLP) model to predict the its runtime service cost based on the given application input. Since such MLP models have a simple structure, they can be efficiently trained even with limited historical data; meanwhile, an MLP model requires minimal computational resources to make predictions, suitable for real-time scheduling tasks. Besides, in the accuracy aspect, given the application-level similarity shown in Fig. 4, such an application-specific prediction method can attain better accuracy performance.

Fig. 6 elaborates the workflow of our MLP-based cost prediction method. For the runtime input prompt, we first perform vectorization using TF-IDF [31] (Term Frequency-Inverse Document Frequency). TF-IDF is a lightweight and efficient method for converting text into numerical vectors, focusing on word importance rather than deep semantic analysis. It's ideal for quick processing with minimal overhead. Then the vectorized input will be passed into the app-specific 4-layer MLP to get the predicted application cost. The number

of neurons in the first layer is proportional to the average application input size. The training is conducted on 100 samples per application, optimized via gradient descent with Mean Squared Error (with L2 regularization). Experimental results later in Sec. V-D demonstrate that such an MLP-based prediction method achieves high accuracy with low overhead.

## C. Application-level Fair Queuing

In this part we elaborate the key queuing algorithm of Justitia. In Justitia, each inference is queued based on the overall demand of the LLM application it belongs to, so that all the inference requests of a high-priority application can be served promptly without being interleaved. In particular, considering the preemption overhead (e.g., KV cache swapping), we follow the original non-preemptive scheduling principle<sup>3</sup> in vLLM [12]: any pending inference request in the waiting queue—regardless of its scheduling priority—cannot preempt a running inference, thus application-level preemption only occurs when an entire inference request finishes.

Recall that our insight is to conduct saturated application serving following the fair completion order, and the key is to acquire that fair completion order efficiently—at application arrival time. One choice is to use the VTC scheduler [10] as the reference system. Yet, idealized fair sharing requires each application sticks to its fair share, whereas an inference scheduled in VTC can take an arbitrarily large KV cache space as needed to accommodate its prompt, disrupting the expected execution status. In that sense, getting the VTC completion order requires a trial run (or simulation) based on the full application demand knowledge (including those arriving later). Moreover, estimating the fair completion order requires continuously refreshing—upon the arrival or completion of any application—the remaining resource demand and the latest resource fair share, which is cumbersome.

Fortunately, we find that the problem we face now is similar to the packet scheduling problem in the networking field. In network scheduling, the network port can only send one packet at a time (which is non-preemptable), yet it is expected that different packets can fair share the network resource at each instant (to attain flow-level fairness). In that case, the packet scheduler shall deliberately select the next packet to transmit, with the objective to minimize the delay any packet may encounter compared to the fair-sharing case. The classical solution to the packet scheduling problem is fair queuing [13], [14]. In principle, the fair queuing algorithm also use the completion order of different packets under an idealized fair sharing scheme as their scheduling priorities. That idealized fair sharing scheme is called Generalized Processor Sharing (GPS), where the backend resources are arbitrarily divisible and equally allocated to each active tenant. As our problem,

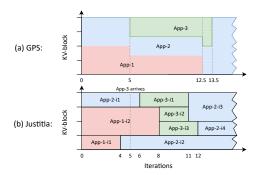


Fig. 7: In Justitia, applications are prioritized according to their relative completion order under GPS.

it is hard to estimate the completion time in GPS at packet arrival time—without knowing the runtime packet contention status; to get the relative scheduling order in one shot at packet arrival time, the fair queuing algorithm employs a notion called *virtual time*, which seeks to adjust the time elapsing rate instead of each application's expected fair completion time when the application contention state changes. We thus borrow such virtual-time based fair queuing method for scheduling LLM applications.

Fig. 7 presents the key queuing mechanism of our proposed Justitia scheduler (for simplicity we ignore demand volume inflation during inference). For LLM application scheduling, we also use the GPS scheme to gauge the fair completion order (i.e., the scheduling priority) of LLM applications. As shown in Fig. 7, suppose the application completion order under the idealized fair-sharing system, GPS, is App-1, App-3, and App-2, then we set that order as their scheduling priority. In that sense, when App-2-i1 (the first inference request of App-2) completes, the requests from App-3 would take the service opportunity. Specifically, for efficient completion order estimation, we adopt the notion of virtual time V(t), which is defined as a function of the real time t:

$$V(0) = 0,$$

$$\frac{\mathrm{d}}{\mathrm{d}t}V(t) = M/N_t.$$
(2)

Here, M is the total KV cache space, and  $N_t$  is the number of active applications in GPS at time t;  $M/N_t$  thus represents the instantaneous fair share. Further, V(t) increases at the marginal rate at which each application receives service in GPS. When App-j arrives at time  $a_j$  (with  $C_j$  be its KV token-time cost), Justitia calculates its *virtual finish time*  $F_j$ —the time at which the application would complete in GPS—as:

$$\bar{f}_j = V(a_j) + C_j. \tag{3}$$

The virtual finish time of an application, once calculated, requires no update in the future. While the arrival of later applications would change the fair-serving rate, they do not change the relative completion order among existing applications (i.e., the relative order in  $\{F_j\}$ ), because each active application would always be serviced with the same rate. Justitia further adopts the  $\{F_i\}$  order as the scheduling priority. In

<sup>&</sup>lt;sup>3</sup>To avoid interference vLLM is designed to be non-preemptive [12]: if the KV cache space is used up, the running inference would be placed into the swapped queue, which has a higher priority than the waiting queue; when the KV cache space is available again, the applications in the swapped queue (if any) would be served first. In that sense, when viewing the swapped queue as part of the running queue, any running inference would never be preempted by those pending requests in the waiting queue.

TABLE I: Summary of important notations and definitions.

M	The number of KV-blocks in a server
	The number of KV-blocks in a server
$\bar{f}_j$	The application completion time for app- $j$ in GPS
$f_j$	The application completion time for app- $j$ in Justitia
$C_j$	The total KV token-time required by app-j
$C_{\max}$	The maximum KV token-time required by any application
$c_{\rm max}$	The maximum KV token-time required by any single inference
$a_j$	The arrival time of app- $j$
$e_i$	The ending time of the slowdown period of app-j

this way we can attain low scheduling overhead: the status refreshing overhead on application arrival or completion is *constant*, and the complexity to select the next application to serve is  $O(\log N_t)$ .

## D. Delay Analysis

In this part we analyze the fairness properties of Justitia. In practice, an inference cannot be started if the available KV block volume is smaller than its prompt size, which may cause the memory fragmentation problem. For ease of analysis, we assume the prompt size of typical inferences are much smaller in scale than the total KV cache space. Our measurement show that, in the released Mooncache dataset [32], each request in average only takes 3.20% of the total KV cache size when running LLaMA-3.3 70B on H100 (with 28.65GB KV cache space). Therefore, we can faithfully neglect the fragmentation problem. That is, if there are inferences waiting in the queue, all the KV-blocks on the server would be fully utilized.

**Symbols.** We let applications be indexed in ascending order of their inference start time. That is, app-j is the  $j^{\text{th}}$  application that is allocated KV-blocks in Justitia. Meanwhile, we let  $C_j$  represent the total KV token-time of app-j, which is the sum of all the KV token-time of all its inferences. We also let  $C_{max}$  denote the maximum total KV token-time among all applications. Similarly, we let  $c_{max}$  represent the maximum KV token-time required by any single inference across all applications. Finally, we let  $f_j$  denote the completion time of app-j in Justitia, and  $\bar{f}_j$  the GPS completion time. Table I summarizes the notations used in the analysis. Given these definitions, we then establish the constant delay bound of Justitia through the following theorem.

**Theorem 1** (Constant delay bound). With Justitia, an application is guaranteed to complete within a constant time after its completion in GPS, i.e., for each app-j, we have

$$f_i - \bar{f}_i \le 2c_{\text{max}} + C_{\text{max}}/M. \tag{4}$$

*Proof.* Our analysis critically focuses on the *slowdown period* of an application. In particular, we say an application is *slowed down* at time t if it has a *backlogged* inference waiting for service. In other words, at any moment in the slowdown period, the application could have run more inferences if receiving more KV-blocks. Because slowdown delays the application completion, bounding the timespan of the slowdown periods is the key to analyzing the longest possible delay.

For each app-j, we let  $a_j$  be the arrival time of app-j. Depending on the number of available KV-blocks at time  $a_j$ ,

app-j is either slowed down, or allocated a sufficient number of KV-blocks to run all inferences right after the arrival. In particular, we let  $e_j$  be the time when the slowdown period of app-j ends. We then have  $a_j < e_j$  if app-j experiences slowdown, and  $a_j = e_j$  otherwise. In either case, after the slowdown period, app-j runs all the backlogged inferences in parallel, and is guaranteed to complete after at most the maximal inference runtime, i.e.,

$$f_j \le e_j + c_{\max}. \tag{5}$$

Therefore, to bound the application completion time, it is critical to analyze when the slowdown period ends (i.e.,  $e_i$ ).

We consider the most general case where app-j is slowed down right after the arrival, i.e.,  $a_j < e_j$ . During the slowdown period  $[a_j, e_j]$ , all the KV-blocks are busy. While Justitia preferentially offers KV-blocks to applications in ascending order of their GPS completion times, applications may start services out of order due to the dynamic arrivals. In particular, an applications that completes before app-j in GPS may arrive late, after app-j starts in Justitia. Let  $\mathcal A$  be the set of all these applications, i.e.,  $\mathcal A = \{k \mid k > j \text{ and } \bar f_k \leq \bar f_j\}$ .

On the other hand, an application that completes after app-j in GPS may start earlier in Justitia, before app-j arrives. Let app-m be such an application that is serviced the most recently. That is, m is the largest integer satisfying both  $m \leq j-1$  and  $\bar{f}_m > \bar{f}_j$ , i.e.,  $\bar{f}_m > \bar{f}_j \geq f_k$  for all m < k < j. In other words, app-m completes after applications  $m+1,\ldots,j$  in GPS, but is allocated KV-blocks before all these applications in Justitia. We let  $\mathcal{B}$  be the set of all these applications that complete after app-j in GPS but start earlier in Justitia, i.e.,  $\mathcal{B} = \{k \mid m < k \leq j\}$ .

By definition, applications in A and B, though completing earlier than app-m in GPS, are serviced no earlier than app-m in Justitia. These applications must have not yet arrived before app-m is allocated KV-blocks—otherwise Justitia would have serviced them before app-m. We then have

$$\min_{k \in \mathcal{A} \cup \mathcal{B}} \{a_k\} \ge b_m,\tag{6}$$

where  $b_m$  is the first time when app-m is allocated KV-blocks in Justitia. This suggests that since  $b_m$ , GPS has completely serviced, at least, all applications in  $A \cup B$  by the time app-m finishes, i.e.,

$$\bar{f}_j \ge b_m + \frac{1}{M} \sum_{k \in \mathcal{A} \cup \mathcal{B}} C_k.$$
 (7)

We next analyze  $f_j$ , the completion time of app-m in Justitia. During time interval  $[b_m,e_j]$ , Justitia may have completed all the applications in  $\mathcal A$  and  $\mathcal B$ , along with at most M-1 inferences requiring a maximal runtime. In addition, app-m is allocated KV-blocks at  $b_m$ , and may also complete before  $e_j$ . Finishing these works using all M KV-blocks takes at most

$$e_j \le b_m + \frac{1}{M} \left[ \sum_{k \in \mathcal{A} \cup \mathcal{B}} C_k + C_m + (M - 1)c_{\max} \right]. \tag{8}$$

Plugging (8) into (5) and subtracting (7), we have

$$f_j - \bar{f}_j \le c_{\max} + \frac{M-1}{M}c_{\max} + C_m/M < 2c_{\max} + C_{\max}/M.$$

This completes the proof.

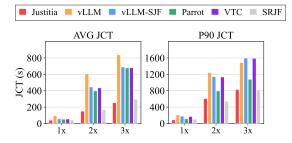


Fig. 8: JCT performance with the LLaMA-7B model.

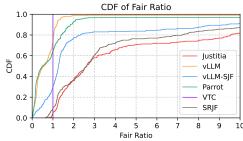


Fig. 10: CDF of applications' finish-time fair ratios (i.e., realistic JCT normalized by VTC-JCT) under  $3 \times$  density.

#### V. EVALUATION

#### A. Setup

Hardware Platform. We have implemented Justitia atop vLLM [12], the mainstream LLM serving framework. We evaluate Justitia with both single-GPU and multi-GPU experiments<sup>4</sup>. The large language models that we use are LLaMA-7B (on single-GPU) and LLaMA-13B [33] (on multi-GPU). The evaluations conducted on a single GPU utilize a server equipped with four 16-core AMD EPYC 7302 CPUs, 128 GB of RAM and one NVIDIA A100 PCIe 40 GB GPU. The multi-GPU evaluations are performed on a server with four 20-core Intel Xeon Gold 6133 CPUs, 258 GB of RAM and four Tesla V100 PCIe 16 GB GPUs.

**Workloads.** For our experiment, we created a mixed workload suite with 300 LLM applications, each with distinct inputs from the original datasets. To be specific, we include the following application classes: (a) MapReduce Summarization (MRS) [34], (b) Plan-and-Execution (PE) [35], (c) Code Checking (CC) [5], (d) Knowledge-Based-Query-Answering Verification (KBQAV) [5], (e) Equation Verification (EV) [5], (f) Fact Verification (FV) [36], (g) ALFWorld Interaction (ALFWI) [36]. (h) Document Merging (DM) [20] and (i) Self Consistency (SC) [6]. Similar to prior work [37], [38], we set the sampling probability of small (EV, FV, CC, ALFWI and KBQAV—usually less than 1 min), medium (CG, PE and SC—usually between 1 and 10 min) and large (DM and MRS—usually longer than 10 min) applications to be 72%, 26%, and 2%, respectively. Regarding application submission, we follow the request arrival time in the production traces released by Mooncake [32], with the submission window

<sup>4</sup>Such a setup is similar to existing scheduling works [7], [26]. Our experimental conclusions also apply to larger clusters with a global queue.

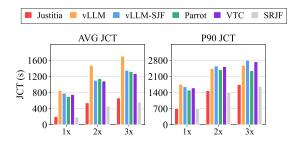


Fig. 9: JCT performance with the LLaMA-13B model.

respectively set to 6, 9, 18 mins (i.e., with workload intensity respectively be  $3\times$ ,  $2\times$  and  $1\times$ ).

**Baselines.** We compare Justitia with five baselines: (a) vLLM [12], which adopts FCFS policy at the inference level; (b) vLLM-SJF [26], which schedules LLM inferences following SJF policy with the Distillbert-predicted inference durations; (c) Parrot [7], which adopts the FCFS policy at the application level; (d) VTC [10], which seeks to approximate the instantaneous fair-sharing policy at the application level; (e) SRJF, which uses our predicted serving cost to enforce the shortest-remaining-job-first policy at the *application* level.

Metrics. Justitia is expected to behave well in both efficiency and fairness. Regarding efficiency, we adopt the average and P90 job completion time (JCT, meaning the duration between job arrival and completion); here a job refers to a running application triggered by a given user input. Regarding fairness, we adopt the notion of finish-time fair ratio, which is the relative ratio between a job's realistic completion time and its completion time under a fair scheduler (we use VTC as the baseline). The higher that ratio, the better the fairness level.

## B. End-to-end Scheduling Performance

**Efficiency performance.** We first evaluate the efficiency performance of Justitia. Fig. 8 and Fig. 9 respectively show the overall serving performance under LLaMA-7B and LLaMA-13B setups. From the two figures, Justitia can substantially outperform the mainstream schedulers. Specifically, the average JCT under Justitia is 57.5% (61.1%) better than that under VTC (Parrot). In the meantime, Justitia in fact attains a very close JCT performance with SRJF, indicating that it can attain near-optimal efficiency.

Fairness performance. Regarding the fairness performance, we further depict the cumulative distribution function (CDF) of all the applications' finish-time fair ratio, with the results shown in Fig. 10. It shows that 92% applications can complete under Justitia no later than it would have under VTC (with the worst-case delay be 26.0%, which is much smaller than others). This confirms the fairness superiority of Justitia. Note that SRJF can also attain a relatively good fairness performance, because it can avoid head-of-line blocking and, many prioritized applications cannot saturate the service backend—thus the remaining KV resources are multiplexed by others, exhibiting a fair sharing effect to some extent. Yet, SRJF may

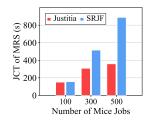


Fig. 11: Behaviors when serving a large application with many small ones.

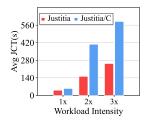


Fig. 12: Performance comparison between different cost modeling methods.

starve the large applications, which we next verify with a micro-benchmark experiment.

#### C. Micro-benchmark Experiment on Starvation

A well-known deficiency of SRJF is the starvation problem: when short applications keep arriving, the execution of long applications may be infinitely delayed. To confirm that, we conduct a micro-benchmark experiment. To be specific, we first submit a large "elephant" application—MapReduce Summarization, and then keep submitting a set of "mice" applications—randomly from KBQAV, CC, ALFWI—once per second. In Fig. 11, we show the relationship between the JCT of the elephant application and the number of "mice" applications respectively under SRJF and Justitia. It clearly demonstrates that, with an increasing number of mice applications, the elephant application may potentially be delayed forever under SRJF; in contrast, that delay under Justitia is bounded regardless of the number of competing applications.

#### D. Ablation Study

In Sec. IV-A and Sec. IV-B we have respectively adopted *memory-centric cost modeling* and *MLP-based demand prediction*; here we check their effectiveness with ablation studies.

Necessity of memory-centric cost modeling. To check the necessity of memory-centric cost modeling, we introduce a variant of Justitia by replacing its cost model with that of VTC (i.e., p+2d [10]), which we call Justitia/C. We repeat the experiment in Fig. 8, and compare the average and P90 JCT under Justitia/C with the vanilla Justitia. As shown in Fig. 12, compute-centric cost modeling would incur a JCT performance degradation of up to 42.3%, which confirms the necessity to adopt memory-centric cost modeling.

Effectiveness of MLP-based demand prediction method. To check the superiority of MLP-based demand prediction, we introduce another Justitia variant: Justitia-S3, which uses a Distillbert model for cost prediction in Justitia. In Justitia-S3, the Distillbert model is trained on the same dataset as the MLP-based models. In Table II we compare the two methods in multiple aspects: average prediction error (prediction gap normalized by the ground-truth), average prediction overhead, average JCT, as well as the model training time. Table 2 suggests that our MLP-based method can remarkably outperform the distillbert-based method in all those aspects.

TABLE II: Performance comparison between MLP and Distillbert in prediction (under 2× workload density). MLP and Distillbert model training is conducted on a V100 GPU.

Prediction Model	Average Relative Error (%)	Average Inference Overhead (ms)	Average JCT (s)	Training Time
MLP	53.0	2.16	151.1	$\sim$ 1 min $\sim$ 2 h
Distillbert	452	55.7	366.7	

TABLE III: Scalability performance

Arrival Rate (APP/min)	15	20	30	50	100
Scheduling Overhead (ms)	0.778	1.827	3.076	5.190	8.093

## E. Overhead Analysis

The overhead in Justitia stems from two sources: (1) the one-shot prediction performed when an application arrives, and (2) the updating of system virtual time and the selection of highest-priority application upon the arrival or completion of any application. As can be seen also in Table II, the prediction overhead for a single new application is approximately 2.16 ms, which is a negligible cost. Table III further demonstrates the average scheduling latency of Justitia under varying arrival rates (reflecting different scheduling scales). The results confirm that the scheduling overhead remains consistently low (under 10 ms) in all scenarios.

# VI. ADDITIONAL RELATED WORKS

Apart from the scheduling works in Sec. II-B, many additional works seek to accelerate individual LLM requests in a series of aspects. For example, FlashAttention [39] and FlashDecoding [40] algorithms seek to accelerate LLM inference at the operator level; model-quantization [41] and prefill-decode-separation [42] methods have been proposed to maximize backend utilization. Besides, speculative decoding methods like [43] are also applied to improve the inference throughput by accelerating the decoding process. These works are orthogonal to us and can work together with Justitia.

## VII. CONCLUSION

In this paper, we propose Justitia, an efficient and also fair scheduler for LLM applications. Justitia works by scheduling LLM applications in a saturated manner following their fair completion order. Specifically, it quantifies the true service cost of LLM applications in a memory-centric manner, and also adopts a MLP-based prediction method to obtain the application service cost at its arrival time. Moreover, Justitia adopts the virtual-time based fair queuing method to efficiently get the expected application completion order under an idealized fair scheduler, with an additional theoretical fairness guarantee. Testbed measurements with diverse workloads confirm that Justitia can behave well in both fairness and efficiency.

#### REFERENCES

- H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," arXiv preprint arXiv:2307.06435, 2023.
- [2] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, "Large language models in medicine," *Nature medicine*, vol. 29, no. 8, pp. 1930–1940, 2023.
- [3] Q. Gu, "Llm-based code generation method for golang compiler testing," in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 2201–2203.
- [4] K. M. Jablonka, Q. Ai, A. Al-Feghali, S. Badhwar, J. D. Bocarsly, A. M. Bran, S. Bringuier, L. C. Brinson, K. Choudhary, D. Circi et al., "14 examples of how llms can transform materials science and chemistry: a reflection on a large language model hackathon," *Digital discovery*, vol. 2, no. 5, pp. 1233–1250, 2023.
- [5] I. Chern, S. Chern, S. Chen, W. Yuan, K. Feng, C. Zhou, J. He, G. Neubig, P. Liu et al., "Factool: Factuality detection in generative ai–a tool augmented framework for multi-task and multi-domain scenarios," arXiv preprint arXiv:2307.13528, 2023.
- [6] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," arXiv preprint arXiv:2203.11171, 2022.
- [7] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of {LLM-based} applications with semantic variable," in 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), 2024, pp. 929–945.
- [8] Y. Fu, J. Chen, S. Zhu, Z. Fu, Z. Dai, A. Qiao, and H. Zhang, "Efficiently serving llm reasoning programs with certaindex," arXiv preprint arXiv:2412.20993, 2024.
- [9] X. Tan, Y. Jiang, Y. Yang, and H. Xu, "Towards end-to-end optimization of llm-based applications with ayo," in ACM ASPLOS, 2025.
- [10] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in Serving Large Language Models," arXiv e-prints, p. arXiv:2401.00588, Dec. 2023.
- [11] R. Ibne Seraj Khan, K. Jain, H. Shen, A. Mallick, A. Parayil, A. Kulkarni, S. Kofsky, P. Choudhary, R. St. Amant, R. Wang, Y. Cheng, A. R. Butt, V. Rühle, C. Bansal, and S. Rajmohan, "Ensuring Fair LLM Serving Amid Diverse Applications," arXiv e-prints, p. arXiv:2411.15997, Nov. 2024.
- [12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.
- [13] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," ACM SIGCOMM Computer Communication Review, vol. 19, no. 4, pp. 1–12, 1989.
- [14] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the singlenode case," *IEEE/ACM transactions on networking*, vol. 1, no. 3, pp. 344–357, 1993.
- [15] J. Wu, S. Yang, R. Zhan, Y. Yuan, L. S. Chao, and D. F. Wong, "A survey on Ilm-generated text detection: Necessity, methods, and future directions," *Computational Linguistics*, pp. 1–66, 2025.
- [16] S. S. Kannan, V. L. Venkatesh, and B.-C. Min, "Smart-Ilm: Smart multi-agent robot task planning using large language models," in 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2024, pp. 12140–12147.
- [17] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, pp. 521–538.
- [18] S. Gao, Y. Chen, and J. Shu, "Fast state restoration in Ilm serving with heache," in *Proceedings of the Twentieth European Conference* on Computer Systems, 2025, pp. 128–143.
- [19] "Gemini: Next-generation AI model," https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#sundar-note, 2024.
- [20] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk et al., "Graph of thoughts: Solving elaborate problems with large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, 2024, pp. 17682–17690.

- [21] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," ACM Computing Surveys, vol. 55, no. 12, pp. 1–38, 2023.
- [22] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez et al., "Sglang: Efficient execution of structured language model programs," Advances in Neural Information Processing Systems, vol. 37, pp. 62557–62583, 2024.
- [23] B. Wu, Y. Zhong, Z. Zhang, S. Liu, F. Liu, Y. Sun, G. Huang, X. Liu, and X. Jin, "Fast distributed inference serving for large language models," arXiv preprint arXiv:2305.05920, 2023.
- [24] Y. Fu, S. Zhu, R. Su, A. Qiao, I. Stoica, and H. Zhang, "Efficient Ilm scheduling by learning to rank," arXiv preprint arXiv:2408.15792, 2024.
- [25] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Efficient interactive llm serving with proxy model-based sequence length prediction," arXiv preprint arXiv:2404.08509, 2024.
- [26] R. Shahout, E. Malach, C. Liu, W. Jiang, M. Yu, and M. Mitzenmacher, "Don't stop me now: Embedding based scheduling for llms," in *ICLR*, 2025.
- [27] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient {GPU} cluster scheduling," in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 289–304.
- [28] "OpenAI Assistants API." https://platform.openai.com/docs/assistants/ overview, 2024.
- [29] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An Ilm-empowered Ilm inference pipeline," Advances in Neural Information Processing Systems, vol. 36, 2024.
- [30] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "s<sup>3</sup>: Increasing gpu utilization during generative inference for higher throughput," *Advances* in Neural Information Processing Systems, vol. 36, pp. 18015–18027, 2023
- [31] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.
- [32] R. Qin, Z. Li, W. He, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Moon-cake: A kvcache-centric disaggregated architecture for llm serving," arXiv preprint arXiv:2407.00079, 2024.
- [33] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al., "Llama 2: Open foundation and fine-tuned chat models," arXiv preprint arXiv:2307.09288, 2023.
- [34] "Langchain." https://github.com/langchain-ai/langchain, 2025.
- [35] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face," *NeurIPS*, 2023.
- [36] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in International Conference on Learning Representations (ICLR), 2023.
- [37] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in 15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21), 2021.
- [38] S. Jayaram Subramanya, D. Arfeen, S. Lin, A. Qiao, Z. Jia, and G. R. Ganger, "Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling," in ACM SOSP, 2023.
- [39] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *NeurIPS*, 2022.
- [40] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, Y. Dong, Y. Wang et al., "Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics," *Proceedings* of Machine Learning and Systems, vol. 6, pp. 148–161, 2024.
- [41] S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer, "Squeezellm: Dense-and-sparse quantization," arXiv preprint arXiv:2306.07629, 2023.
- [42] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving," in *USENIX OSDI*, 2024.
- [43] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao, "Medusa: Simple Ilm inference acceleration framework with multiple decoding heads," arXiv preprint arXiv:2401.10774, 2024.