Efficient Serving of LLM Applications with Probabilistic Demand Modeling

Yifei Liu
Shanghai Jiao Tong University
China

Zuo Gan
Shanghai Jiao Tong University
China

Zhenghao Gan
Shanghai Jiao Tong University
China

Weiye Wang
Shanghai Jiao Tong University
China

Chen Chen*
Shanghai Jiao Tong University
China

Yizhou Shan
Huawei Cloud
China

Xusheng Chen
Huawei Cloud
China

Zhenhua Han
Unaffiliated
China

Yifei Zhu
Shanghai Jiao Tong University
China

Shixuan Sun
Shanghai Jiao Tong University
China

Minyi Guo
Shanghai Jiao Tong University
China

Abstract

Applications based on Large Language Models (LLMs) contains a series of tasks to address real-world problems with boosted capability, which have dynamic demand volumes on diverse backends. Existing serving systems treat the resource demands of LLM applications as a blackbox, compromising end-to-end efficiency due to improper queuing order and backend warm up latency. We find that the resource demands of LLM applications can be modeled in a general and accurate manner with *Probabilistic Demand Graph* (PDGraph). We then propose Hermes, which leverages PDGraph for efficient serving of LLM applications. Confronting probabilistic demand description, Hermes applies the Gittins policy to determine the scheduling order that can minimize the average application completion time. It also uses the PDGraph model to help prewarm cold backends at proper moments. Experiments with diverse LLM applications confirm that Hermes can effectively improve the application serving efficiency, reducing the average completion time by over 70% and the P95 completion time by over 80%.

1 Introduction

Large Language Models (LLMs) [23, 30, 74] have demonstrated its strong capability in language understanding and generation. Yet, even though LLMs are evolving rapidly [52, 62, 72], it is commonly recognized that a single LLM request is often deficient for many real-world problems [11]. For example, the input context windows of typical LLMs are often of limited sizes (e.g., less than 10M tokens) [26, 50], thus processing a large document would require issuing multiple parallel inference requests. Meanwhile, the output of an LLM request may be unreliable (i.e., suffering *hallucination* [42]), and additional LLM requests would be required to ensure output quality (e.g., with self-reflection [43]). Moreover, the built-in knowledge and interaction modality of LLMs are also limited, and non-LLM tasks like docker execution [4, 75] or third-party tool calling [68, 69] are often integrated to augment LLM capabilities [17]. We call such a set of correlated LLM and non-LLM tasks—which collaborate to address a realistic problem—as an *LLM application*. LLM applications would be a mainstream AI workload paradigm in the future.

LLM applications are often hosted on the cloud [2, 40, 51], and it becomes critical to serve them efficiently—attaining fast application completion such that users can promptly get the valid final output. Nevertheless, compared with traditional workloads in OS and big data fields, LLM applications have two distinct characteristics. First, the resource demands of an LLM application (e.g., the token generation length of each request and the inter-request structure)—dependent to the runtime inputs—are uncertain a priori. Given the difficulty to know the total application demand volume, existing serving systems like vLLM [13] and Parrot [51] choose a simple scheduling algorithm like FCFS, which hurts the scheduling efficiency due to the head-of-line blocking problem. Second, serving LLM applications often involves diverse backend resources (like the docker container for code testing [4] or the KV cache for inference acceleration [74]), many of which are prepared in an on-demand manner [33]. Consequently, the application completion time may be delayed due to the intermittent warm-up latency on cold backends. In summary, the absence of application demand information renders existing LLM serving systems inefficient in both queuing management and backend preparation.

The key to efficient serving of LLM applications is to obtain accurate demand information. In fact, although LLM applications exhibit substantial demand dynamicity, it does not mean that the serving system has to be demand-agnostic:

*Chen Chen is the corresponding author.

the application viewpoint brings promising opportunities for demand perception. A typical LLM application is composed of multiple functional units (e.g., an inference task with a fixed system prompt to verify a just-generated claim); the resource demand of a given unit—due to its distinct functionality characteristics, is relatively stable across different runs. Since the LLM applications are usually recurring with their code files hosted on the cloud, it is possible to apply static and dynamic program analysis to model its resource demand.

However, making accurate demand modeling for general LLM applications is a non-trivial task. Different applications may involve different backend types and have different functional unit structures, and we need to design proper demand description primitives for generality. Meanwhile, in each application run, the user input affects the triggered function units, and the demand volume on each function unit may also deviate from the average value previously profiled. In that sense, any fixed demand representation would be over-assertive; meanwhile, we also need to conduct online estimation refinement to more precisely estimate the resource demands in the ongoing run.

In this paper, we design Hermes, an efficient system for serving LLM applications. In Hermes, we propose to model the resource demands of an LLM application as a *Probabilistic Demand Graph* (PDGraph). A PDGraph organizes the diverse functional units of a LLM application with a graph structure: each PDGraph node describes the demand quantity of the corresponding functional unit with a distribution function, and records the downstream dependencies with an associated branch-taking probability. Moreover, by analyzing the demand correlation between upstream and downstream units, we can keep refining the demand estimation with the latest execution status of the ongoing run. With PDGraph, we can thus faithfully estimate, in a probabilistic manner, the resource demands of the entire application as well as of the upcoming unit. We can then leverage such information to optimize the queuing order and to determine the backend preparation moment.

In queuing optimization, the problem we now face is how to determine the scheduling order of applications whose demand is expressed as a distribution rather than a deterministic value. In that case, scheduling applications under the classical shortest-remaining-time-first (SRTF) algorithm (based on the mean value of the demand distribution) is no longer optimal for minimizing the average appliation completion time. To adapt to demand uncertainty, Hermes adopts the Gittins policy. Gittins policy [65] has been proven optimal in scheduling jobs with *unknown demands but known demand distributions*, a good fit for scheduling LLM applications. It works by calculating a Gittins index which is a runtime estimator of an application's true remaining processing time. Additionally, we also consider the cases where each application is associated with a deadline; with the estimated demand distribution information, Hermes adopts the least-slack-time-first (LSTF) algorithm that prioritizes applications with higher risk of deadline violation.

Apart from queuing optimization, Hermes also leverages PDGraph for backend prewarming. With the demand information recorded in PDGraph, during the execution of a functional unit, we can prewarm the backends needed by its downstream units prior to their arrival. Since a function unit in LLM applications may have multiple downstream units and their arrival times are also uncertain (depending on the execution time of the current unit), setting the backend type as well as the moment to prewarm presents as a clear trade-off. Given that prewarming wrong backends or prewarming backends too early would waste resources, Hermes introduces a knob to tune the trade-off between the latency reduction effect and the resource wastage incurred. Such a prewarming principle can be generally applied to both LLM backends (like KV cache) and non-LLM backends (like docker containers).

We have implemented Hermes with over 4,000 lines of Python code, and have also prepared a workload suite containing representative LLM applications. The performance of Hermes is evaluated against mainstream serving systems with the diverse set of LLM applications. Experimental results show that Hermes can reduce the average application completion time by over 70%, and can also make an improvement of over 1× for cases with explicit deadlines. Moreover, ablation studies further confirm the effectiveness of Hermes in mitigating demand uncertainty as well as in efficient resource provisioning on diverse backends.

In summary, this paper makes the following contributions:

- We identify the limitations of existing systems in serving LLM applications (which have distinct characteristics of dynamic demands and diverse backends), i.e., the long queuing delay and backend warm-up delay.
- We propose to model the demands of LLM application in a probabilistic and structured manner with PDGraph, and further design Hermes to leverage PDGraph for queuing order optimization as well as for backend preparation.
- We build a workload suite containing typical LLM applications, and confirm the effectiveness of Hermes with testbed experiments, demonstrating a salient efficiency improvement over existing systems.

2 Background and Motivation

2.1 LLM Applications: A Primer

LLM applications. Large language models (LLMs) [23, 30, 74] with their built-in world knowledge can potentially be adopted for various fields like finance [49], arts [53] and science [63]. Due to the prohibitively high training cost and the exhausted data corpus sources, there is an emerging

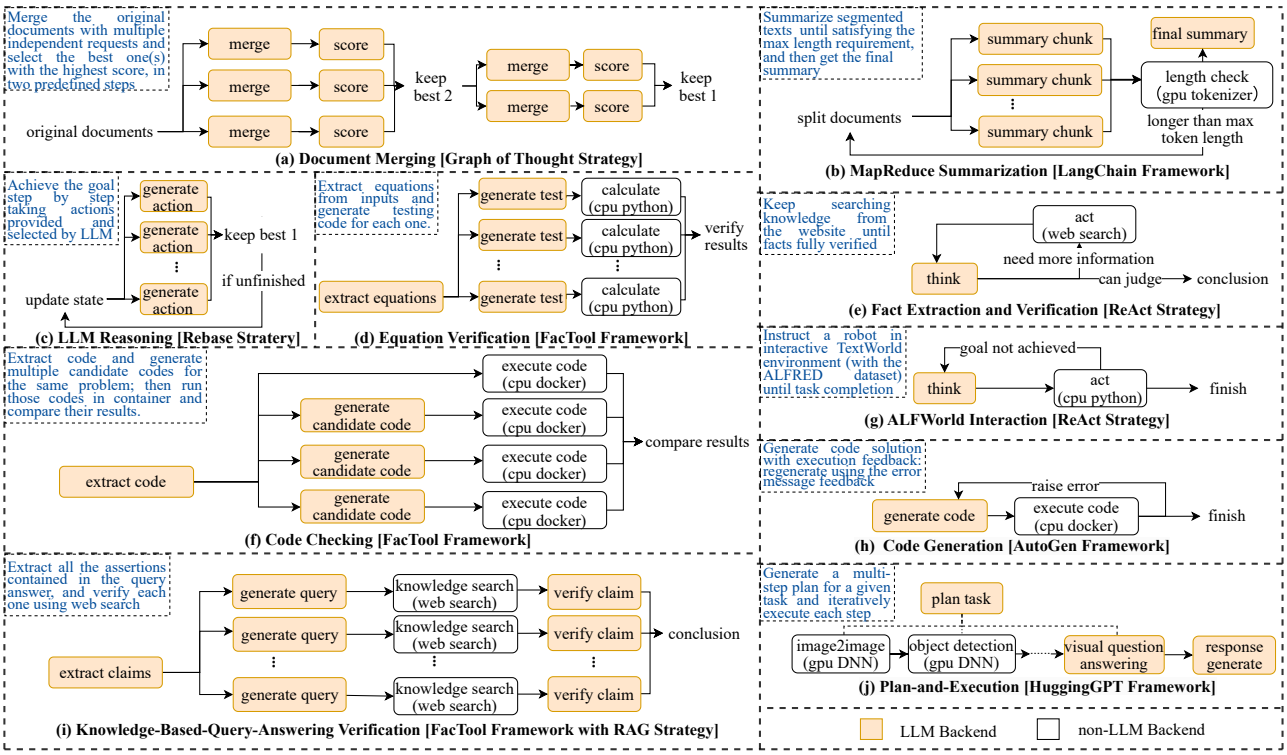


Figure 1. Ten representative LLM applications: (a) Document Merging (DM) [6]; (b) MapReduce Summarization (MRS) [12]; (c) LLM Reasoning (LLMR) [32]; (d) Equation Verification (EV) [7]; (e) Fact Extraction and Verification (FEV) [8]; (f) Code Checking (CC) [4]; (g) ALFWorld Interaction (ALFWI) [3]; (h) Code Generation (CG) [5]; (i) Knowledge-Based-Query-Answering Verification (KBQAV) [9]; (j) Plan-and-Execution (PE) [10].

trend to boost the power of existing LLMs by *scaling-out* instead of by *scaling-up* [11]. That is, to enlarge the input length [44, 50], improve the output reliability [27, 43], or integrate more functionalities [68, 69], LLM practitioners often need to jointly execute multiple LLM inference tasks and the necessary non-LLM tasks, which form a compound task workflow—we call an *LLM application*. To elaborate, we create a comprehensive workload suite of LLM applications, as shown in Fig. 1. That workload suite involves ten representative applications built with specific frameworks (e.g., FacTool [27]) or following certain control strategies (e.g., Graph of Thought [22] and ReAct [43]).

In particular, we note that LLM applications differ from traditional workloads (in OS or big data) in *demand dynamicity* and *backend diversity*. First, the execution status of an LLM application is highly dynamic: due to mechanism like ReActing [43, 78] and LLM-planning [68], the inter-task structure can only be determined at runtime; meanwhile, both the input and output token lengths of each LLM task are also unknown beforehand [33, 77]. Second, serving LLM applications requires provisioning diverse LLM and non-LLM backends. Regarding LLM backends, different applications may prefer different foundation models or fine-tuned adapters [25, 38]. Regarding non-LLM backends, a diverse set of backends may also be needed: for example, in the code-generation application, the LLM-generated codes need to be tested on CPU backends (e.g., with a docker container [4, 75] to provide an isolated environment); meanwhile, the HuggingGPT application [10] may require loading diverse non-LLM models as tools.

Cloud-based serving of LLM applications. LLM applications, just like OpenAI assistant API [2], are usually hosted on the cloud for ease of user access [1, 40, 51, 71]. The code files of those applications are maintained by the service provider, which would be launched when users submit their application inputs. When serving multiple LLM applications from different users [40, 71], the service provider needs to first organize the incoming tasks in a global queue, and then respectively dispatch each task to the proper LLM or non-LLM backend. A good serving system shall attain high execution efficiency and also high resource utilization. Specifically, when launching an LLM application, users primarily focus on the *application completion time*—that is, the time when the high-quality output (like the valid code) is finally returned [51]. Therefore, the efficiency objective shall be set at the application level, i.e., minimize the average application completion time (ACT) or—if the expected deadline is provided—maximize the goodput (number of applications

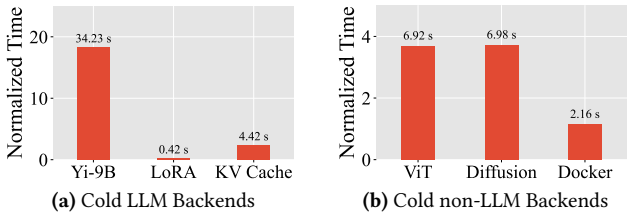
that complete before the deadline). Meanwhile, for high resource utilization, LLM service provider shall avoid holding resources idly in serving the LLM applications.

2.2 Existing Practices and Their Limitations

For LLM applications, due to their strong demand dynamicity and backend diversity, existing serving systems often fail to yield high scheduling efficiency.

First, confronting uncertain resource demands, existing practices usually adopt a simple heuristic in task queuing (like FCFS), which is known to be inefficient. A mainstream LLM serving framework nowadays is vLLM [13], which schedules LLM inference tasks in a FCFS manner—without awareness to the high-level existence of LLM applications; in resource contention, the constituting tasks of an LLM application may be interleaved by tasks from other applications, suffering delayed application completion. Recently, Parrot [51] proposes to schedule the inference tasks of an application together; however, without concrete knowledge of resource demand volume, it schedules LLM applications also following FCFS, thus suffering the head-of-line-blocking problem. Our testbed measurement shows that, executing a small application (KBQAV in Fig. 1(i)) after a large application (DM in Fig. 1(a)) yields an average ACT of 52.8s, yet with the reversed order it is only 40.3s. Another work VTC [71] proposes to fairly allocate the computing capability between different applications (tenants) to avoid head-of-line blocking, yet it still suffers low efficiency by forcing each application to use only its fair share under contention.

Second, since LLM applications may call diverse backends at uncertain moments, it is inefficient to make such backends standby all the time. For example, among the 130 official input examples provided by HuggingGPT [68], our measurement shows that only 20 runs triggered the text-to-video model. To avoid resource wastage, such serving backends shall be prepared in an on-demand manner, which however incurs cold start delay on diverse backends. For instances, regarding LLM backends, users may choose different foundation models based on their cost-quality preference [25] or require customized LoRA adaptors [38] for domain-specific capability, and meanwhile KV-cache reuse may also be adopted to accelerate inference speed [33, 47]; such model or cache content need to be pre-loaded into the GPU HBM before inference starts. Regarding non-LLM backends, for code-generation applications the docker containers need to be launched before code testing, and for HuggingGPT applications [68] the DNN models like object recognition also need to be loaded into GPUs before task execution. Those preparation processes may take non-negligible time and slow down the end-to-end application completion. In Fig. 2a and Fig. 2b, we show the typical warm-up time of a series of backend contents, which are up to 18× of the inference time of a typical inference task (with 1000 input tokens and 100 output tokens). Noticing the need to conduct KV cache prewarming, the recent CachedAttention work [33] proposes to conduct KV cache eviction and prefetching based on the scheduler waiting queue, a method we call Evict/Prefetch-Waiting-Queue (EPWQ). However, EPWQ cannot prefetch the KV cache for unbacklogged requests (e.g., a future request not yet submitted); because EPWQ does not have the application viewpoint, a downstream task of an LLM application—even if it has the highest priority—may still bear a KV cache loading delay after arrival.



(a) Cold LLM Backends **(b)** Cold non-LLM Backends

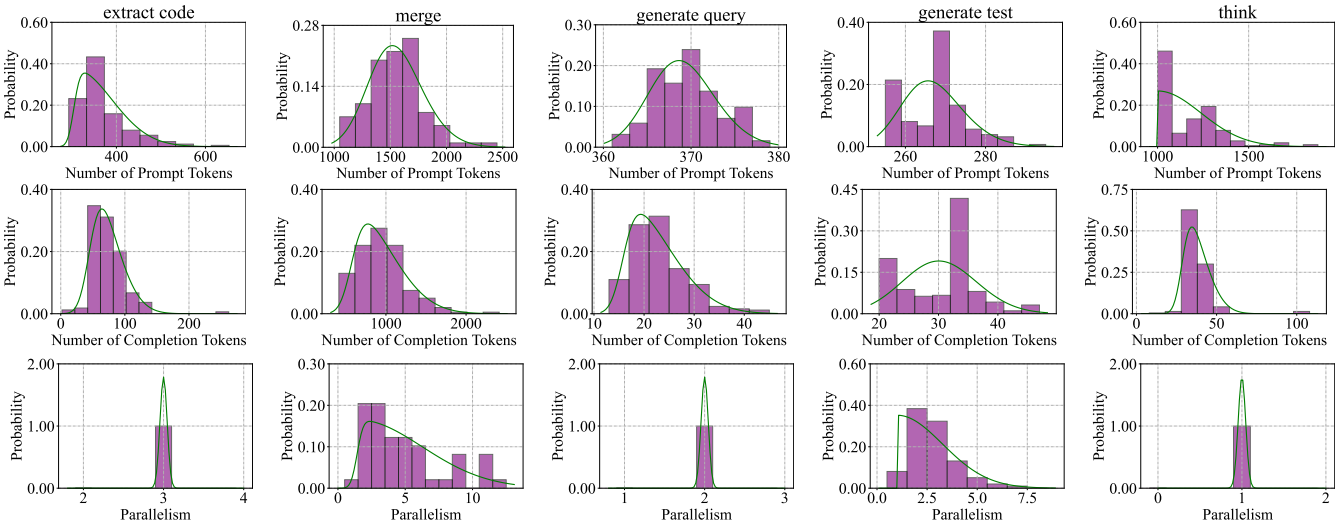
Figure 2. Costs to warm up code backends (normalized to the execution time of a typical LLM inference with 1000/100 input/output tokens). The LoRA is for LLaMA-7B with rank 8 and the KV cache size is 128K (loaded to A100 GPUs). The docker image is `python:3.10-slim`.

To summarize, confronting dynamic demands on diverse backends, existing LLM serving systems have to employ static queuing policies and also conduct reactive resource provisioning. Such a *demand-agnostic* serving methodology hurts the overall efficiency of LLM applications.

2.3 Insight and Challenges

Insight. To enhance the serving efficiency of LLM applications, the key is to obtain their demand information in advance. While the demands of LLM applications are dynamic, we note this does not mean that they are totally unpredictable. In fact, LLM applications are usually recurring, making it possible to make thorough performance profiling offline. As shown in Fig. 1, each LLM application is composed of multiple functional units: tasks in an LLM unit (e.g., the `merge` unit in the DM application) share the same system prompt, and tasks in a non-LLM unit require distinct backend type (e.g., the `execute-code` unit require specific docker container in CG application). With program analysis, we can profile the execution information of each function unit, which can potentially be applied for demand estimation.

To elaborate, we execute five LLM applications from Fig. 1 for 100 times each, replaying different inputs sampled from the official dataset. Fig. 3 presents the execution statistics, including input/output token lengths and request parallelism, for an arbitrarily selected functional unit of each application. As illustrated in Fig. 3, while the resource demands are non-deterministic, they still exhibit significant stability across multiple trial runs. For example, the output token length



(a) Code Checking **(b)** Document Merging **(c)** KBQAV **(d)** Equation Verification **(e)** FEV

Figure 3. Prompt and completion token length distribution for an arbitrarily-selected request (marked at the top)—in five applications each conducting over 100 trial runs. 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 for reference).

of the `merge` request in DM application is around 1000, yet for the `generate-query` request in KBQAV application it is between 10 and 50. That phenomenon is indeed reasonable because demand volume is to some extent an inner property based on the usage scenario of an LLM application: the output of the `merge` request is a document, whereas the `generate-query` request produces a short query. Consequently, the latter's length distribution exhibits a significantly smaller mean and deviation. Such stability also exists in the request parallelism aspect.

Therefore, we can potentially exploit dynamic program analysis to estimate the resource demands of LLM applications. Such information can help optimize task queuing and backend provisioning for the LLM serving system. Ideally, if we know the total execution time of each LLM application, we can apply shortest-remaining-processing-time (SRPT) to minimize the average ACT; if we know which backend to use next and the moment to use it, we can calculate the best time to prewarm the cold backend, attaining fast application completion without any resource wastage.

Challenges. Yet, while promising, it is however a non-trivial task to leverage program profiling of LLM applications for the best efficiency performance, and the challenges are twofold.

On the one hand, demand dynamicity and backend diversity are built-in properties of LLM applications, and the demand modeling must be conducted in a general and also accurate manner. By "general", the modeling method needs to be compatible with diverse backends and variant demand dependency structures; by "accurate", the modeling method needs to maintain a proper level of uncertainty—trying to narrow down the estimation range yet without being over-assertive. To elaborate, in each application run, the type-/number of functional units and the task durations—which crucially relate to the runtime inputs—may substantially deviate from the average case of the profiling results; simply recording the average profiling results would be inaccurate.

On the other hand, profiling-based performance modeling can only mitigate but not eliminate demand uncertainty, rendering it hard to directly attain the optimal serving performance. In fact, we will show later that the classical SRTF policy is no longer optimal when scheduling jobs with uncertain demand. Therefore, our queue management and backend prewarming methods must adapt to such uncertainty for the best possible performance. We will address such challenges in the later section.

3 Hermes Design

3.1 Overview

We present Hermes, an efficient serving system for LLM applications hosted on clouds. Hermes is built upon a performance modeling paradigm called *Probabilistic Demand Graph* (PDGraph), which can yield accurate demand estimation for general LLM applications in an *offline+online* manner (§3.2). As shown in Fig. 4, Hermes maintains a knowledge base storing the profiled information for each application. Once a user launches a cloud-hosted LLM application with its tasks submitted to the `HermesScheduler`, the `HermesScheduler` retrieves the PDGraph for that application to assist the scheduling process. To be specific, the `HermesScheduler` schedules each task based on the application priority it belongs to, which is calculated based on the estimated demand

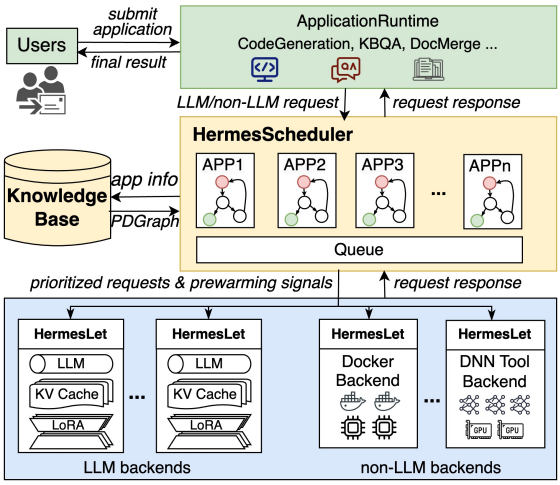


Figure 4. The system architecture of Hermes.

```
LLM-application:
    unit-1:
        backend-spec:
            LLM: LLaMA-3
        backend-consumption:
            input-length: [879,764 …] # List
            output-length: [210,355 …] # List
            parallelism: [5,3 …] # List
            significance-Mask: [T, F, F, T, F]
        next-unit:
            unit-1: 0.75 # Probability
            unit-2: 0.25
    unit-2:
        backend-spec:
            docker: python:3.10-slim
        backend-consumption:
            duration: [7.6s,5.4s …] # List
        next-unit:
            unit-3: 1.0 # Probability
    …
    unit-N: …
```

Figure 5. PDGraph example of an LLM application.

information (§3.3). Moreover, based on the instantaneous execution status, the `HermesScheduler` also issues backend prewarming signals to the `HermesLet` at proper moments to avoid code start latency (§3.4).

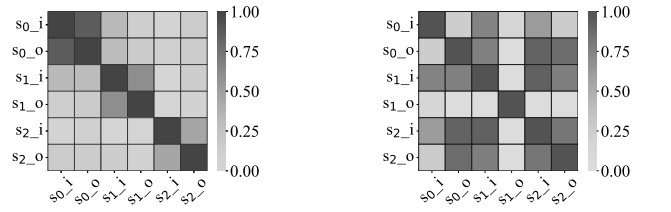
3.2 Demand Modeling with PDGraph

In this part, we seek to conduct accurate demand modeling for general LLM applications, with the objective to optimize queue management and backend prewarming strategies. Recall that in Fig. 3, we learned that the demand volume of a given functional unit is relatively stable across different trial runs. However, the inter-unit structure—due to mechanisms like reacting and LLM-planning—is not determined apriori, and meanwhile the specific demand volume in a single run may deviate from the average case. Therefore, in demand modeling of LLM applications, we need to cover the dynamicity of inter-unit structure for *generality*, and also combine historical execution information with runtime hints for *accuracy*. To that end, we propose to model the LLM applications as a probabilistic graph of functional units, and assign proper properties to each unit node. We call thus a modeling paradigm as *Probabilistic Demand Graph* (PDGraph).

Demand modeling for general LLM applications with PDGraph. As shown in Fig. 5, in our demand modeling paradigm with PDGraph, each application is recorded as a list of functional units. For each functional unit, we record three data types: *backend-spec*—describing the resource type and configuration specifics, *backend-consumption*—describing resource consumption amount on that backend, and *next-unit*—describing the probabilistic jumping relationship between dependent functional units. In this way, we can model the resource demands of general applications with diverse structures and backends: each backend (even user-defined ones like OpenAI function calling [15]) can be uniformly described as a backend-spec item, and the static-structure applications can also be covered by a PDGraph where the jumping probability between any two units is always 1. In particular, for each LLM unit, we record the input/output length as well as the request parallelism instead of the absolute time to enable adaption of diverse runtime execution platforms (e.g., A100 and H100). Those properties can facilitate multiple efficiency optimization aspects: recording the backend consumptions can facilitate the prioritization of short applications; recording the backend specifics and the jumping relationship can help prewarm the soon-called backends.

Moreover, to handle dynamic backend consumptions and probabilistic jumping relationships, we express the profiled resource demand as a distribution instead of as a single value. To be specific, after each profiling run, we append the execution information of each functional unit to a list. For dynamic backend consumptions, we note that recording the raw values is better than recording the coefficients of the fitted skewed norm distribution due to the depiction fidelity (the true distribution is in fact irregular) and computation efficiency: it takes much time (up to seconds) to fit out the coefficients as well as to support our later calculations. For probabilistic jumping relationships, we calculate the historical jumping frequency as the branch-taken probability. Meanwhile, we set the maximal number of recorded values to 1000 (evicted in FIFO manner), and the storage cost is indeed negligible.

Further, we use the Monte Carlo method to estimate the total demand of the entire application. To be specific, we perform random walk along the PDGraph (sampling the downstream branch as well as the unit demand value), until a sufficient number of samples are collected. Note that this can be done efficiently because of the limited number of nodes and list sizes in typical PDGraphs. by summing up the LLM execution time (input/output token lengths are transferred to the absolute service time based on the average per-token processing time in the runtime environment) and



(a) KBQAV **(b)** Document Merging (DM)

Figure 6. The Pearson correlation coefficients between demands of dependent functional units. We show three units respectively in KBQAV and Document Merging, where s_x_i, s_x_o represent the input/output token length of the x-th unit.

non-LLM execution time, we can get an estimation of the total execution cost for an application.

Online estimation refinement for better accuracy. Accurate demand estimation is crucial for the overall serving efficiency of LLM applications. Yet, naively using the historical execution is insufficient: it is merely a prior knowledge and the runtime execution information—the posterior knowledge—is also valuable for refining the estimation results. In fact, due to the structural dependencies among stages, the resource demands of a stage are often correlated to the upstream ones. Specifically, we note that there typically exist three cross-unit demand correlation patterns:

- A unit's input length may correlate to the upstream unit's input/output length. For example, for the DM application, the input of each request in the `scoring` unit is a superset of a request's output in the (upstream) `aggregate` unit (plus a fixed system prompt). Meanwhile, for the looping unit, downstream requests would share the same prompt template as the upstream ones, indicating similarity in the input length.
- A unit's output length may be correlated to its input length as well as the upstream unit's output length. For example, for the `generate-code` unit in CG application, a more complex input task—typically associated with a longer prompt—tends to produce a longer code segment. Similarly, the output lengths of requests are also similar across looping units.
- A unit's parallelism may be correlated to the upstream unit's parallelism. For example, as shown in KBQAV applications, for each inference request in the (upstream) `generate-queries` unit, a corresponding inference request would be launched at the (downstream) `verify-claim` unit.

To verify the existence of the above correlation types, we resort to *Pearson Correlation Analysis* [66] based on our profiled data over 100 trial runs. We divide the demand range into 10 buckets and use $P(X = i)(i = 0, 1, \ldots, 9)$ to denote the probability that the demand quantity variable X falls into bucket-i. Then we calculate the Pearson correlation coefficient between two demand variables X and Y as $\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathbb{E}((X-\mu_X)(Y-\mu_Y))}{\sigma_X \sigma_Y}$. In Fig. 6, for KBQAV and Document Merging, we further depict the Pearson correlation coefficients between any two demand variables (each represents the input/output length of a unit). For KBQAV, each unit's output length is strongly correlated to its input length; yet for DM, each unit's input length is highly-related to the upstream unit's input length.

Fig. 6 also suggests that each application has distinct correlation patterns; for prediction efficiency, we only consider the demand correlations with a coefficient (ρ) larger than 0.5. To that end, in each unit we add a five-tuple ($M_I^{\tilde{I}}$, $M_I^{\tilde{O}}$, $M_O^{\tilde{O}}$, M_O^I, $M_P^{\tilde{P}}$) to mask whether the corresponded demand correlation holds: M_Y^X represents whether the demand variable X affects Y; I, O and P ($\tilde{I}$, $\tilde{O}$ and $\tilde{P}$) respectively represents the input length, output length and request parallelism of the current (upstream) unit.

The above correlation analysis enables more precise online demand prediction. Upon the completion of a unit, its execution information can be immediately adopted for demand prediction of the future units. In that case, we are facing a *conditional prediction* problem. Suppose the request input and output length of the just-finished unit is respectively in bucket i and o—with both correlated with the input length of the downstream unit, and we need to predict $P(I|\tilde{I} = I, \tilde{O} = o)$. To accomplish that, we join the historical execution records of the two dependent functional units (i.e., yielding $(\tilde{I}, \tilde{O}, I)$ tuples respectively from each trial round), and filter out the profiled tuples satisfying $\tilde{I} = i$ and $\tilde{O} = o$; then the I distribution within the filtered records, which guides our Monto Carlo sampling process, can yield a more accurate estimation of the resource demand in the current round.

3.3 PDGraph-based Queue Management

With PDGraph, we can now obtain a probabilistic demand estimation of an LLM application's overall resource demand. In this part, we explore how to leverage such demand estimation to optimize the overall queuing performance.

Optimizing average ACT based on the *Gittins* policy. Regarding efficient scheduling of LLM applications, a primary metric is the average *application completion time* (ACT). A classical algorithm proven optimal in that regard is *shortest-remaining-processing-time* (SRPT) [64]. However, naively applying SRPT is not appropriate for our problem: SRPT relies on deterministic demand knowledge to make scheduling decisions, yet the resource demands of LLM applications are uncertain a priori (our PDGraph strategy mitigates but does not totally eliminate such uncertainty). Although it is possible to use the expected value (mean) of a demand distribution to emulate SRPT scheduling, this method often fails to work

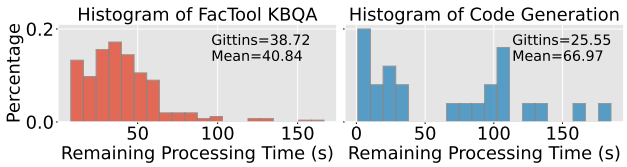


Figure 7. Given two running LLM applications, FacTool_KBQA and Code_Generation, under SPRT it is the former that is scheduled first, yet under the Gittins policy it is the latter (with a lower Gittins rank). Regarding the probability to complete within 9 seconds, for FacTool_KBQA it is 0.3%, yet for Code_Generation it is 16%; this suggests that prioritizing Code_Generation indeed has a higher reward in practice.

well due to its blindness to instantaneous execution progress. For example, for a request with an expected output length of 20 tokens, it is possible that its token generation process does not stop even after 100 tokens. In that case, we need to timely refresh the demand estimation based on the latest progress, rather than sticking to the prior expectations. Otherwise, the remaining processing time—the expected execution time minus the executed time—may ironically become *negative*. Therefore, we need to introduce uncertainty-awareness in designing our scheduling algorithm.

The problem we now face—minimizing the average completion time for jobs with *unknown durations* but *known duration distributions*—has been studied in the literature, for which the *Gittins* policy [16, 34, 65] has been proven optimal. The basic idea of Gittins policy is to calculate a *Gittins rank* for each job—based on its executed time so far and its size distribution—as the scheduling priority. Specifically, let an application's duration distribution be $\mathcal{D}$, and it has been executed for a time period of a, then its Gittins rank G can be expressed as:

$$G(\mathcal{D}, a) = \inf_{\Delta>0} \frac{\mathrm{E}[\min\{X_{\mathcal{D}} - a, \Delta\} \mid X_{\mathcal{D}} > a]}{\mathrm{P}\{X_{\mathcal{D}} - a \le \Delta \mid X_{\mathcal{D}} > a\}}, \tag{1}$$

where $X_{\mathcal{D}}$ is the random variable under $\mathcal{D}$. Given a service budget Δ (i.e., the number of additional tokens allowed to be generated henceforth), the denominator in Eq. 1 represents the possibility that the generation process can finish before Δ (which contributes to a small average completion time), and the numerator represents the corresponding resource consumption (the cost paid to serve it). To determine the scheduling priority, the Gittins Index essentially computes the maximum achievable cost-to-return ratio under any possible cost budget, and existing works [34, 65] have shown that this method can theoretically yield the minimal average completion time. In Fig. 7, we show a case where the Gittins policy yields a different yet better scheduling order than SPRT. Besides, for computation efficiency, in practice we update the estimation of Gittins index only after each bucket period.

Extend to other efficiency[1] objectives. Regarding efficient LLM application scheduling, apart from the ACT criterion, users may also associate a deadline with their application request, and expect that the final output can be returned before that deadline. In that cases, the efficiency criterion is the ratio of applications that can satisfy the deadline requirements. For deadline-based scheduling, we note that PDGraph-based demand modeling can also help to attain better performance—surpassing the demand-agnostic scheduling methods exemplified by Earliest-deadline-first (EDF) [20]. EDF always prioritizes the application with the earliest deadline, yet an earlier deadline does not mean a higher urgency—it also depends on the demand volume. Based on PDGraph, we define the worst-case slack time S relative to the deadline as

$$S(\mathcal{D}, a) = t_{\mathrm{ddl}} - t_{\mathrm{now}} - (\sup X_{\mathcal{D}} - a), \tag{2}$$

where S represents the worst-case remaining time till deadline. We then prioritize applications in ascending order of S, emulating the *Least Slack Time First* (LSTF) policy [29].

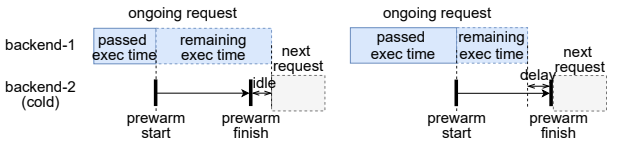
3.4 PDGraph-based Backend Prewarming

As elaborated in §2.2, cold backends would delay the completion of an LLM application. With PDGraph, Hermes can estimate the arrival of the downstream requests as well as their desired backend types. By prewarming cold backend to be used by the downstream unit before the completion of the current unit, we can get rid of the warm-up delay from the critical path. However, backend prewarming is not free-of-charge. On the one hand, a functional unit may have multiple downstream units, and it is possible that the prewarmed backends are never used; on the other hand, the current unit may complete much later than expected, meaning that the prewarmed backend would wait idly for a long time. Therefore, as shown in Fig. 8, there exists a trade-off in determining the prewarming triggering moment: more aggressive prewarming can help attain faster application completion—yet at the cost of larger resource wastage.

We then introduce a knob K, called *expected prewarming effectiveness*, to tune that trade-off. To be specific, given a running function unit, if one of its downstream backends is not active, we then determine *whether* and *when* to trigger prewarming with an analytical method. We let p_s be the probability of selecting the target unit, t_c be the actual completion time of the current unit, t_s be the moment to start prewarming, and t_p be the duration of the prewarming operation. Then the probability that prewarming is effective (i.e., when the target downstream request finally arrives, its backend is already well-warmed) can be expressed by

$$p_e = p_s * P(t_c > t_s + t_p). \tag{3}$$

[1]Note that, while we primarily focus on the efficiency aspect of LLM application scheduling in this paper, with the demand information from PDGraph, we can in fact also enhance the fairness aspect by enabling demand-aware fair scheduling methods like weighted fair queuing [21].



(a) Effective Prewarming **(b)** Ineffective Prewarming

Figure 8. Trade-off in setting up the prewarming triggering time: prewarming too early wastes resources, yet prewarming too late delays application completion.

Given the knob K, if $p_s < K$, then Hermes does not trigger prewarming; otherwise, it triggers prewarming at the time such that $p_e = K$. Such a knob K can be deemed as a kind of service level agreement: a more significant application (e.g., from premium users) can be assigned with a smaller K value.

The above prewarming technique applies to the non-LLM backends like docker and DNNs, and also applies to the cache resources in LLM backends. In fact, the cache space on LLM servers is significant for efficient LLM serving. Existing works [17, 47] have shown that KV-cache can remarkably improve the token generation speed for requests sharing the same prompt tokens. Meanwhile, Low-Rank Adaptation (LoRA) [38] is prevalently adopted to support LLM serving with customized models; those LoRA adaptors take non-negligible time to load and should thus be also cached in memory [48] (2.2). In that sense, preloading the desired KV cache or LoRA is also necessary for the efficient execution of LLM applications, which will also be evaluated later in §5.4.

4 Implementation

We have implemented Hermes with 4100 lines of Python codes. As illustrated in Fig. 4, our implementation comprises two key components: the `HermesScheduler` and the `HermesLet`. We uses vLLM 0.4.3 [47] as the low-level inference engine and ZeroMQ-based RPC [14] to exchange control messages between the `HermesScheduler` and the `HermesLet`. Additionally, we build a comprehensive benchmark suite consisting of 11,300 lines of Python code, which includes all of the application types depicted in Fig. 1.

In building the PDGraph models, we profile each application for 1000 times, and store their PDGraphs in a JSON file. The average JSON file size for one application is around 100KB. After loading such JSON file, the `HermesScheduler` launches a background RPC thread to keep refreshing the demand estimation (at per-bucket granularity) based on the latest execution status reported from the `HermesLet`. The `HermesScheduler` would accordingly refresh the application scheduling priority (based on the Gittins index or worst-case slack time) and notify the `HermesLet` once the priority changes. The `HermesScheduler` also issues backend prewarming signal at desired time to the `HermesLet`. The `HermesLet` also maintains a dedicated background RPC client for continuous communication with the `HermesScheduler`, through which it reports execution status while receiving updated request priorities and prewarming signals. It also employs two dedicated background threads to prefetch KV cache blocks or LoRA adapters as well as to prewarm docker containers, in accordance with the `HermesScheduler`'s decisions.

5 Evaluation

5.1 General Setups

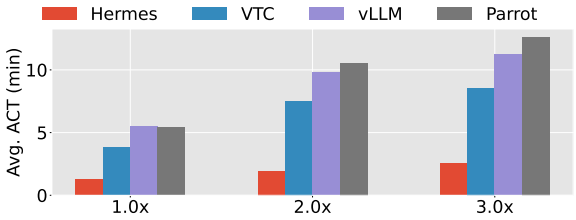
Hardware platform. We evaluate Hermes respectively with single- and multi-GPU experiments. The single-GPU evaluations use a server with one A100 GPU, and the multi-GPU evaluations use a server with eight H800 GPUs. The single-GPU server has four 16-core AMD EPYC 7302 CPUs, 128GB DRAM, 4TB SSDs and one NVIDIA A100-PCIe-40GB GPU. The multi-GPU server has four 48-core Intel Xeon Platinum 8558 CPUs, 2TB DRAM, 28TB SSDs and eight NVIDIA H800-80GB GPUs. Unless otherwise specified, we deployed LLaMA2-7B and LLaMA2-13B on a single A100-PCIe-40GB GPU, and two LLaMA3-70B models across eight H800-80GB GPUs using 4-way tensor parallelism to further evaluate Hermes' performance at scale.

Workloads. To investigate the performance of Hermes in real-world environments, we sampled the request arrival time distribution from the trace published by MoonCake [60], from which we can clearly observe the bursty arrival patterns in real-world scenarios. We utilized its arrival time distribution while sampling real inputs from the application families shown in Fig. 1. Specifically, similar to prior work [41, 59, 81], we set the sampling probability of *small* (EV, FEV, CC, ALFWI and KBQAV—usually less than 1 min), *medium* (CG and PE—usually between 1 and 10 min) and *large* (DM and MRS—usually longer than 10 min) applications to be 72%, 26%, and 2%, respectively.

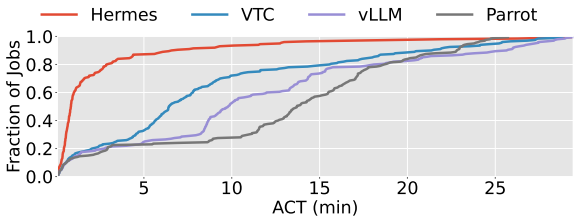
Baselines. We compare Hermes with three scheduling strategies: vLLM [13], Parrot [51], and VTC [71]. As explained in §2.2, vLLM adopts the FCFS policy at the request level, and Parrot adopts the FCFS policy at the application level. VTC seeks to fairly serve the tasks from different users. For scenarios with deadlines, we additionally include the Earliest-Deadline-First (EDF) policy [20] as a baseline. We defer the introduction of other baselines to each specific micro-benchmark experiment. Regarding the default hyper-parameter setup in Hermes, the threshold on Pearson correlation coefficient (§3.2) is set to 0.5, the expected prewarming effectiveness knob K (§3.4) is set to 0.5, and the number of buckets for distribution description (§3.2 and §3.4) is set to 10.

5.2 End-to-end Scheduling Performance

Minimizing average ACT. We first evaluate how Hermes can improve the application serving efficiency measured

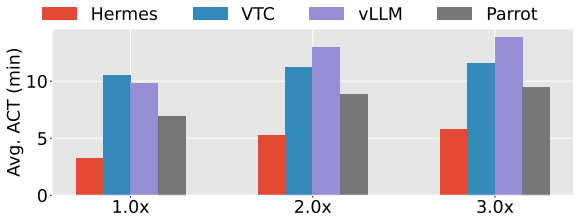


(a) The average ACT with varying application arrival intensities.

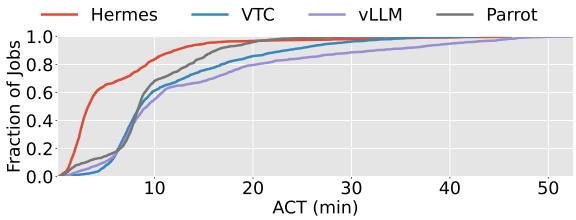


(b) The CDF of ACTs under different schedulers.

Figure 9. The experimental results from a single LLaMA2-7B model running on an A100 GPU.



(a) The average ACT with varying application arrival intensities.



(b) The CDF of ACTs under different schedulers.

Figure 10. The experimental results from two LLaMA3-70B models running on eight H800 GPUs.

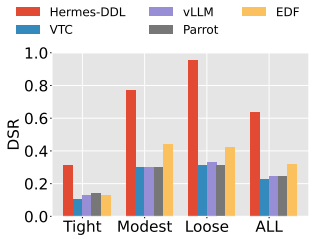


Figure 11. The DDL Satisfactory Ratio under different schedulers.

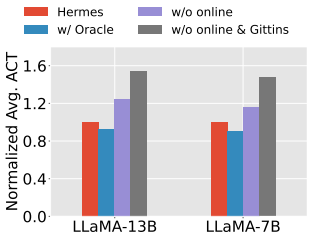


Figure 12. Ablation study on the methods tackling demand dynamicity.

by the average application completion time. We conducted two sets of experiments, running LLaMA2-7B and LLaMA3-70B respectively, and submitted 300 and 3000 applications correspondingly based on different hardware computing capabilities. We evaluated the performance across submission windows of 10, 15, and 30 minutes (corresponding to workload intensity levels of 3×, 2×, and 1×), and reported the ACT distribution. As shown in Fig. 9a and Fig. 10a, Hermes can perform much better than vLLM, Parrot, and VTC in each case. For example, in the LLaMA2-7B experiment under 1× workload intensity, the ACT performance of Hermes is 77.0% (76.7% and 66.7%) better than vLLM (Parrot and VTC); in the LLaMA3-70B experiment under 3× workload intensity, the ACT performance of Hermes is 58.5% (39.1% and 50.3%) better. For tail application performance, in the LLaMA2-7B experiment under 1× workload intensity, the P95 ACT performance of Hermes is 82.4% (69.0% and 74.1%) better than vLLM (Parrot and VTC); in the LLaMA3-70B experiment under 3× workload intensity, the P95 ACT performance of Hermes is 62.2% (21.7% and 45.8%) better.

Maximizing DDL satisfactory ratio. We next evaluate Hermes-DDL, a variant of Hermes specifically designed for deadline-constrained scenarios (based on methods in §3.3). Following the LLaMA2-7B experimental setup, we submit the same 300 applications within a 15-minute submission window. Additionally, we assign distinct deadlines to each task by scaling the original execution time with random factors of 1.2× (*tight*), 1.5× (*modest*), and 2× (*loose*), consistent with methodologies employed in prior studies [35, 55]. Fig. 11 reports the Deadline Satisfaction Ratio (DSR, meaning the ratio of applications that can complete before the specified deadline)—for all the applications as well as for applications in each DDL-scaling category. It shows that Hermes-DDL achieves the highest DSR among all the schemes evaluated, delivering a 1× improvement over EDF. This improvement primarily stems from Hermes-DDL's ability to leverage application demand information, with which it prioritizes the most urgent applications while deferring less critical ones.

5.3 Effect on Tackling Demand Dynamicity

Setup. Recall that to address demand dynamicity, we have adopted multiple techniques including offline profiling, online refinement as well as the Gittins policy. We now verify their respective effectiveness by comparing the performance between four schemes: Hermes, Hermes without online refinement, Hermes without online refinement or Gittins, and Hermes with Oracle (representing the ideal case where the exact demands are available—by conducting a trial run with

`temperature` set to 0). We conduct this experiment respectively with LLaMA-7B and LLaMA-13B as the LLM backend, and submit 300 applications within a time window of 10 minutes.

Results. Fig. 12 shows the average ACT in each case (normalized by that under Hermes). Without online correlation analysis, the average ACT would be inflated by 15.3% with the LLaMA-7B model. When the Gittins policy is further disabled, that performance degradation would increase to 47.5%. Such results confirm the indispensability of both online refinement and the Gittins policy in tackling demand uncertainty. Meanwhile, the performance gap between Hermes and Hermes-Oracle is less than 10%, suggesting that Hermes can deliver near-optimal scheduling performance for LLM applications. We also notice that in each case, the performance of LLaMA-7B and LLaMA-13B are quite similar.

5.4 Effect of Backend Prewarming

Effect of LLM backend prewarming. As explained in §3.4, prewarming the KV and LoRA cache can facilitate LLM serving. To confirm that, we submit 500 applications to the A100 server within 15 minutes. We adopt two cache management baselines: LRU and Evict/Prefetch-Waiting-Queue (EPWQ) [33, 70]. LRU conducts reactive swapping with prewarming, and EPWQ prewarms the KV cache only when the request is already in the waiting queue—which is often too late. Under each method, we measure the overall cache hit ratio (i.e., KV cache be well warmed when the request comes) against different cache sizes (8GB, 16GB, and 32GB). As shown in Fig. 13(a), Hermes consistently achieves the highest cache efficiency, improving the overall cache hit ratio by up to 1.11× (0.33×) compared to LRU (EPWQ). It reduces the average ACT by 18% (6%) compared to LRU (EPWQ).

We then evaluate the benefit to prewarm the LoRA cache. We set `max-loras` (the maximum number of parallel processes per iteration) to 10, `max-cpu-loras` (the number of LoRA caches that can be stored in CPU memory) to 20, and use totally 200 LoRA adapters for LLaMA-7B with a rank of 8. We submit a 25-minute workload containing 1,000 applications, with each application randomly assigned to one of the LoRA adapters (all requests of an application use the same LoRA). Fig. 13(b) shows the cache hit ratio also respectively under LRU, EPWQ and Hermes. Compared to the second best (EPWQ), Hermes can increase the cache hit ratio by 21.1%.

Effect of non-LLM backend prewarming. To evaluate our prewarming strategy in non-LLM backends, we selected two applications, CG and PE, which respectively demands Docker and DNN backends. We separately run each application with varying levels of prewarming aggressiveness (i.e., the expected prewarming effectiveness hyperparameter, K). Fig. 14 illustrates the average latency reduction and resource

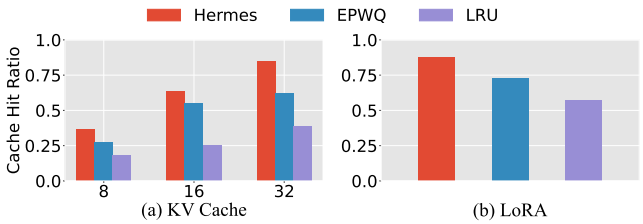


Figure 13. The cache hit ratio of (a) the KV Cache and (b) the LoRA adapter across different cache management strategies.

wastage due to prewarming, which verifies the effectiveness of prewarming in non-LLM backends (for DNN backends, we only analyze the PE applications calling a ViT or Diffusion model). Moreover, Fig. 14 also shows that a smaller K can in general yield faster completion, yet at the cost of higher resource wastage—consistent with our analysis in §3.4.

5.5 Overhead Analysis

The main overhead of Hermes lies in determining the application scheduling priority with Gittins, which is iteratively executed once after a time length of the bucket size (buckets are used in describing the demand distribution as in Fig. 3). In Fig. 15a, we measure the average policy runtime of Gittins under different arrival rates (indicating different scheduling scales). It shows that the scheduling overhead is indeed quite small in each case (less than 3 ms). This is because our PD-Graph models are indeed not large and can be efficiently processed. Further in Fig. 15b, we measure how the Gittins policy runtime varies with the bucket number . It shows that using more buckets to describe the demand distribution almost linearly increases the policy runtime—but does not help improve the scheduling performance.

6 Related Work and Discussions

Accelerating individual LLM requests. Many works seek to accelerate individual LLM requests in a series of aspects. In *operator implementation* aspect, the FlashAttention [28] and FlashDecoding [36] style algorithms are commonly adopted to realize efficient LLM inference by integrating IO awareness. In *model deployment* aspect, model-quantization [31, 46] and paged-attention [47] methods have been proposed to maximize backend utilization. In *inference algorithm* aspect, continuous batching [79] and speculative decoding methods [24, 54] are also applied to improve the inference throughput by accelerating the decoding process. These works are orthogonal to us.

Scheduling multiple LLM requests. Some recent works do notice the need to optimize scheduling performance at the request level. FastServe [77] adopts multi-level feedback queue for LLM inference serving (yet at the cost of relatively high preemption overhead); Llumnix [73] and

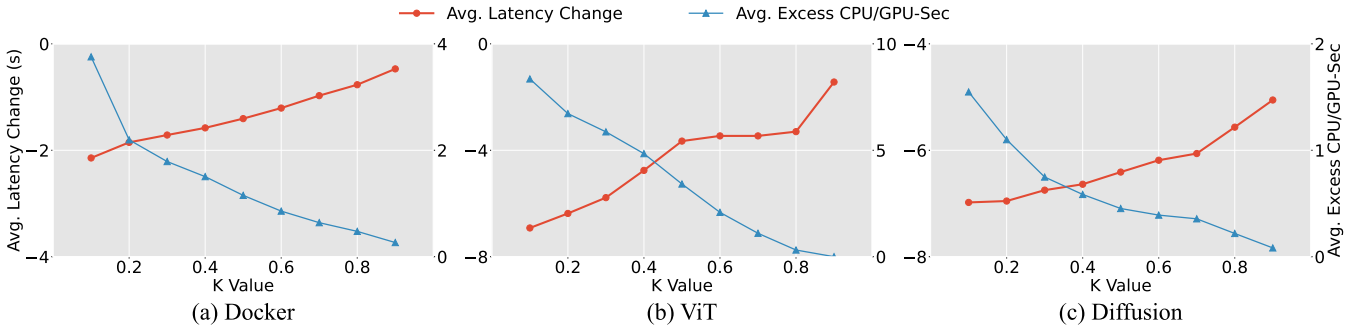
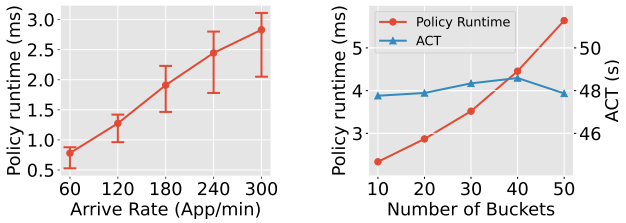


Figure 14. Benefit and cost with different prewarming aggressiveness levels (K) for non-LLM backends. A lower K value indicates more aggressive prewarming.



(a) The policy runtime of Gittins over different application arrive rates.

(b) How Gittins policy runtime and ACT vary with the bucket numbers.

Figure 15. Scheduling policy runtime for Hermes under various application arriveal rates using proportionally sized MoonCake traces.

LoongServe [76] enable runtime request migration to improve load-balancing and isolation. In the meantime, some other works seek to adopt SJF-like scheduling algorithms [37, 57, 67]. While those methods can reduce the average completion time of requests, they are application-agnostic and, as explained in §2.2, fail to yield fast application completion. Meanwhile, Hermes can be easily extended to facilitate multi-backend routing, enabling interference avoidance when co-locating requests on the same engine. For example, with the input/output length information recorded in PDGraph, it is possible to pack demand-complementary requests (e.g., those with long outputs which are memory-intensive and those with short outputs which are compute-intensive) onto a specific engine.

Job demand prediction. Demand prediction is important for efficient job scheduling. In traditional fields, a series of research works have been proposed to make accurate performance prediction with testbed profiling [19, 80] or mathematical modeling [39, 58]. Yet LLM workloads exhibit distinct demand uncertainty due to its generative manner, which is never captured before. Some recent methods have been proposed to predict the output token length of an LLM request—with LLMs themselves [45, 61, 67, 82], at the cost of lengthy model fine-tuning processes. Moreover, none of those methods make demand prediction from the application point of view; blind to the task dependencies, their prediction accuracy is low, and in the meantime they cannot support speculative prewarming of diverse backends.

Extensibility of Hermes in advanced inference architectures. Prefill-decode-separation [18, 83] has recently emerged as a popular LLM deployment paradigm that can avoid inter-request interferences. Hermes can work smoothly with such disaggregated inference architectures. On the one hand, the queuing algorithm in Hermes works at the global level, and the resultant priority applies to all the backends; on the other hand, model or KV cache prewarming remains a common need for each prefill or decode instance under the PD-disaggregation architecture. Meanwhile, Hermes can also be easily extended to facilitate multi-backend routing [37, 56], enabling interference avoidance when co-locating requests on the same engine. For example, with the input/output length information recorded in PDGraph, it is possible to pack demand-complementary requests (e.g., those with long outputs which are memory-intensive and those with short outputs which are compute-intensive) onto a specific engine. We will explore such functionalities with Hermes in the future.

7 Conclusion

In this paper, we propose Hermes, an efficient serving system designed for LLM applications. Hermes employs a probabilistic demand graph (PDGraph) to model the resource demands of LLM applications. Using the PDGraph, Hermes adopts the Gittins policy in queuing management to minimize the average application completion time, and adopts the LSTF algorithm to maximize the deadline satisfactory ratio for the cases with explicit deadlines. Hermes also leverages the PDGraph model to determine when and what backend to prewarm. Experimental results on popular workloads show that Hermes can remarkably improve the serving efficiency of LLM applications, attaining an improvement of over 70%.

References

[1] Multi-Agent-as-a-Service. https://medium.com/data-science/multi-agent-as-a-service-a-senior-engineers-overview-fc759f5bbcfa, 2024.

[2] OpenAI Assistants API. https://platform.openai.com/docs/assistants/overview, 2024.

[3] The origin code of alfworld interaction. https://github.com/ysymyth/ReAct/blob/6bdb3a1fd38b8188fc7ba4102969fe483df8fdc9/alfworld.ipynb, 2024. [Online; accessed 19-Oct-2024].

[4] The origin code of code checking. https://github.com/GAIR-NLP/factool/tree/3f3914bc090b644be044b7e0005113c135d8b20f/factool/code, 2024. [Online; accessed 19-Oct-2024].

[5] The origin code of code generation. https://github.com/microsoft/autogen/tree/0560bdd645dfbc579a71f2f0fea98ea83dd3bb3f?tab=readme-ov-file#quickstart, 2024. [Online; accessed 19-Oct-2024].

[6] The origin code of document merging. https://github.com/spcl/graph-of-thoughts/tree/a939a4577c07c80b8ecb194793b5a4169d99b31b/examples/doc_merge, 2024. [Online; accessed 19-Oct-2024].

[7] The origin code of equation verification. https://github.com/GAIR-NLP/factool/tree/3f3914bc090b644be044b7e0005113c135d8b20f/factool/math, 2024. [Online; accessed 19-Oct-2024].

[8] The origin code of fact extraction and verification. https://github.com/ysymyth/ReAct/blob/6bdb3a1fd38b8188fc7ba4102969fe483df8fdc9/FEVER.ipynb, 2024. [Online; accessed 19-Oct-2024].

[9] The origin code of knowledge-based-query-answering verification. https://github.com/GAIR-NLP/factool/tree/3f3914bc090b644be044b7e0005113c135d8b20f/factool/knowledge_qa, 2024. [Online; accessed 19-Oct-2024].

[10] The origin code of plan-and-execution. https://github.com/microsoft/JARVIS/tree/c62e0faac76c4a2907cabe2cfe4bbe5f2e613400, 2024. [Online; accessed 19-Oct-2024].

[11] The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024. [Online; accessed 21-September-2024].

[12] The tutorials of how to implement mapreduce summarization. https://python.langchain.com/v0.2/docs/tutorials/summarization/#go-deeper-1, 2024. [Online; accessed 19-Oct-2024].

[13] vLLM: Easy, fast, and cheap LLM serving for everyone. https://docs.vllm.ai/en/stable/, 2024. [Online; accessed 19-July-2024].

[14] Zeromq - an open-source universal messaging library. https://platform.openai.com/docs/overview, 2024. [Online; accessed 19-Oct-2024].

[15] OpenAI Function Calling. https://platform.openai.com/docs/guides/function-calling, 2025. [Online; accessed 10-Jan-2025].

[16] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. On the gittins index in the m/g/1 queue. *Queueing Systems*, 63:437–458, 2009.

[17] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. Infercept: Efficient intercept support for augmented large language model inference. In *Forty-first International Conference on Machine Learning*, 2024.

[18] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *USENIX OSDI*, 2024.

[19] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[20] Matthew Andrews. Probabilistic end-to-end delay bounds for earliest deadline first scheduling. In *IEEE INFOCOM*, 2000.

[21] Jon CR Bennett and Hui Zhang. Wf/sup 2/q: worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, volume 1, pages 120–128. IEEE, 1996.

[22] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.

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

[24] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[25] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[26] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[27] I Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei 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.

[28] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[29] Robert I Davis, Ken W Tindell, and Alan Burns. Scheduling slack time in fixed priority pre-emptive systems. In *1993 Proceedings Real-Time Systems Symposium*, pages 222–231. IEEE, 1993.

[30] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[31] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[32] Yichao Fu, Junda Chen, Siqi Zhu, Zheyu Fu, Zhongdongming Dai, Aurick Qiao, and Hao Zhang. Efficiently serving llm reasoning programs with certaindex. *arXiv preprint arXiv:2412.20993*, 2024.

[33] 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, 2024.

[34] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[35] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *ACM ASPLOS*, 2023.

[36] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Yuhan Dong, Yu Wang, et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161, 2024.

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

[38] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[39] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 457–472, 2023.

[40] Redwan Ibne Seraj Khan, Kunal Jain, Haiying Shen, Ankur Mallick, Anjaly Parayil, Anoop Kulkarni, Steve Kofsky, Pankhuri Choudhary, Renèe St Amant, Rujia Wang, et al. Ensuring fair llm serving amid diverse applications. *arXiv e-prints*, pages arXiv–2411, 2024.

[41] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *ACM SOSP*, 2023.

[42] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[43] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. Towards mitigating llm hallucination via self reflection. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1827–1843, 2023.

[44] Hanlei Jin, Yang Zhang, Dan Meng, Jun Wang, and Jinghua Tan. A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods. *arXiv preprint arXiv:2403.02901*, 2024.

[45] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.

[46] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

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

[48] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. Caraserve: Cpu-assisted and rank-aware lora serving for generative llm inference. *arXiv preprint arXiv:2401.11240*, 2024.

[49] Yinheng Li, Shaofei Wang, Han Ding, and Hang Chen. Large language models in finance: A survey. In *Proceedings of the fourth ACM international conference on AI in finance*, pages 374–382, 2023.

[50] Bin Lin, Tao Peng, Chen Zhang, Minmin Sun, Lanbo Li, Hanyu Zhao, Wencong Xiao, Qi Xu, Xiafei Qiu, Shen Li, et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

[51] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. In *USENIX OSDI*, 2024.

[52] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[53] Christos Makridis. The impact of generative artificial intelligence on artists. *Available at SSRN*, 2025.

[54] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 1(2):4, 2023.

[55] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *USENIX OSDI*, 2020.

[56] Chengyi Nie, Rodrigo Fonseca, and Zhenhua Liu. Aladdin: Joint placement and scaling for slo-aware llm serving. *arXiv preprint arXiv:2405.06856*, 2024.

[57] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 18–35, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[59] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric 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.

[60] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

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

[62] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[63] Shuo Ren, Pu Jian, Zhenjiang Ren, Chunlin Leng, Can Xie, and Jiajun Zhang. Towards scientific intelligence: A survey of llm-based scientific agents. *arXiv preprint arXiv:2503.24047*, 2025.

[64] Linus Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[65] Ziv Scully and Mor Harchol-Balter. The gittins policy in the m/g/1 queue. In *2021 19th International Symposium on Modeling and Optimization in Mobile, Ad hoc, and Wireless Networks (WiOpt)*, pages 1–8. IEEE, 2021.

[66] Philip Sedgwick. Pearson's correlation coefficient. *Bmj*, 345, 2012.

[67] Rana Shahout, Eran Malach, Chunwei Liu, Weifan Jiang, Minlan Yu, and Michael Mitzenmacher. Don't stop me now: Embedding based scheduling for llms. *arXiv preprint arXiv:2410.01035*, 2024.

[68] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36, 2024.

[69] Zhuocheng Shen. Llm with tools: A survey. *arXiv preprint arXiv:2409.18807*, 2024.

[70] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

[71] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, 2024.

[72] Yuki Sonoda, Ryo Kurokawa, Yuta Nakamura, Jun Kanzawa, Mariko Kurokawa, Yuji Ohizumi, Wataru Gonoi, and Osamu Abe. Diagnostic performances of gpt-4o, claude 3 opus, and gemini 1.5 pro in "diagnosis please" cases. *Japanese Journal of Radiology*, pages 1–5, 2024.

[73] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *USENIX OSDI*, 2024.

[74] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[75] Olga Vrousgou. Code execution is now by default inside docker container. https://microsoft.github.io/autogen/0.2/blog/2024/01/23/Code-execution-in-docker/. [Online; accessed 19-March-2025].

[76] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 640–654, 2024.

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

[78] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[79] 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 *USENIX OSDI*, 2022.

[80] Peng Zheng, Wendi Feng, Arvind Narayanan, and Zhi-Li Zhang. Nfv performance profiling on multi-core servers. In *2020 IFIP Networking Conference (Networking)*, pages 91–99. IEEE, 2020.

[81] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *USENIX NSDI*, 2023.

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

[83] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *USENIX OSDI*, 2024.