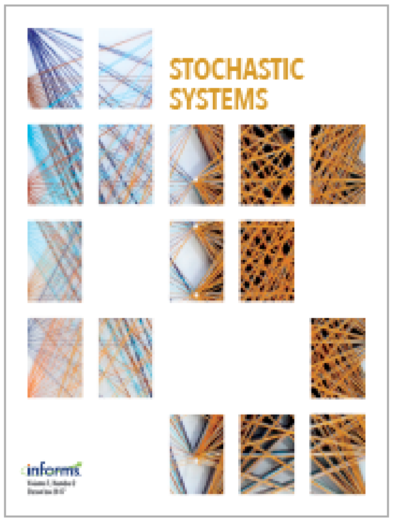


Stochastic Systems

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Queueing, Predictions, and Large Language Models: Challenges and Open Problems

Michael Mitzenmacher, Rana Shahout

To cite this article:
Michael Mitzenmacher, Rana Shahout (2025) Queueing, Predictions, and Large Language Models: Challenges and Open Problems. Stochastic Systems 15(3):195-219. https://doi.org/10.1287/stsy.2025.0106

https://pubsonline.informs.org/journal/stsy

STOCHASTIC SYSTEMS
Vol. 15, No. 3, September 2025, pp. 195–219
ISSN 1946-5238 (online)

Queueing, Predictions, and Large Language Models: Challenges and Open Problems

Michael Mitzenmacher,[a] **Rana Shahout**[a,*]

[a] Computer Science Department, Harvard University, Cambridge, Massachusetts 02138
*Corresponding author
Contact: michaelm@eecs.harvard.edu (MM); rana@seas.harvard.edu, https://orcid.org/0000-0002-9254-8529 (RS)

Received: March 7, 2025
Revised: May 20, 2025
Accepted: June 16, 2025
Published Online in Articles in Advance: July 22, 2025

https://doi.org/10.1287/stsy.2025.0106

Copyright: © 2025 The Author(s)

Abstract. Queueing systems present many opportunities for applying machine learning predictions, such as estimated service times, to improve system performance. This integration raises numerous open questions about how predictions can be effectively leveraged to improve scheduling decisions. Recent studies explore queues with predicted service times, typically aiming to minimize job time in the system. We review these works, highlight the effectiveness of predictions, and present open questions on queue performance. We then move to consider an important practical example of using predictions in scheduling, namely large language model (LLM) systems, which presents novel scheduling challenges and highlights the potential for predictions to improve performance. In particular, we consider LLMs performing inference. Inference requests (jobs) in LLM systems are inherently complex; they have variable inference times, dynamic memory footprints that are constrained by key-value store memory limitations, and multiple possible preemption approaches that affect performance differently. We provide background on the important aspects of scheduling in LLM systems and introduce new models and open problems that arise from them. We argue that there are significant opportunities for applying insights and analysis from queueing theory to scheduling in LLM systems.

Funding: M. Mitzenmacher was supported in part by the National Science Foundation [Grant CCF-2101140]. R. Shahout was supported in part by the Schmidt Futures Initiative and the Zuckerman Institute. M. Mitzenmacher and R. Shahout were supported in part the National Science Foundation [Grants CNS-2107078 and DMS-2023528].

Keywords: queueing theory • predictions • LLM inference scheduling • LLM inference resource allocation

1. Introduction

In this paper, we survey recent work on using predictions in queueing systems as well as recent work on the specific setting of scheduling in large language model (LLM) systems, where predictions seem both useful and natural. We focus on presenting several open questions for consideration. Our purpose is to highlight the work in these areas and encourage researchers to tackle the many interesting problems raised by systems that make use of predictions.[1]

To introduce the queueing theoretic problems, let us consider a standard queue, such as an $M/G/1$ queue—where jobs arrive to a single-server queue according to a Poisson arrival process with general independent and identically distributed service times. When no information about job sizes is available, policies, such as first come, first served (FCFS; also called first in, first out (FIFO)), processor sharing, and least attained service (also called foreground-background), can be used, with FCFS being the most widely adopted. When job service times are known, size-based policies, like shortest job first (SJF) or shortest remaining processing time (SRPT), become feasible.

In practice, however, there are many settings where the exact service time is not known in advance. A potentially promising approach for such settings is to utilize predictions, which may be generated by machine learning models; these models can estimate service times and inform scheduling decisions. Several recent studies have explored queues that use predicted rather than exact service times to reduce the average response time (Wierman and Nuyens 2008; Dell'Amico et al. 2014, 2015; Mailach and Down 2017; Mitzenmacher 2021; Scully et al. 2022; Akbari-Moghaddam and Down 2023). Recent work has also considered the setting of scheduling jobs with deadlines (Salman et al. 2023a, b), demonstrating that predictions can improve scheduling decisions in that setting as well.

More generally, the integration of predictions with algorithm design has given rise to the field of "algorithms with predictions," also known as learning-augmented algorithms. In this area, classical algorithms are improved by incorporating advice or predictions from machine learning models (or other sources), ideally with provable performance guarantees. Although the idea of using additional information to improve algorithms and data structures has some history—the study of online algorithms with advice is a notable example (Boyar et al. 2017)—the specific motivation to make use of predictions from machine learning algorithms has led to new definitions, models, and results. Learning-augmented algorithms have demonstrated their effectiveness across a range of areas as shown in the collection of papers (Algorithms with Predictions Project 2024) on the subject and as discussed in the surveys (Mitzenmacher and Vassilvitskii 2020, 2022).

There are many important and basic queueing theory questions that arise once one focuses on predicted service time. Accordingly, our first goal in this paper is to survey some of the recent work on using predictions for scheduling in single queues and queueing systems and describe open problems and research directions in this area.

From this starting point, we then consider a more concrete and timely application setting: LLM systems. LLM systems have transformed many domains by enabling powerful artificial intelligence (AI)-driven applications, rapidly integrating into workflows and decision-making processes. These systems consist of two main phases: training and inference. Training is a computationally intensive offline process where models learn from massive data sets, whereas inference, the focus of this work, is the real-time execution phase where pretrained models generate responses to user requests.[2] At inference, LLM models generate text token (usually a word or a part of a word) by token, with each new token relying on the context of previously generated tokens in an autoregressive process (where each output is fed back as input for the next prediction). Given the increasing availability of open-source models, such as Llama (Touvron et al. 2023) and DeepSeek (Liu et al. 2024, Guo et al. 2025), inference has become the most common mode of interaction with LLMs, and that will be our focus in this work.

LLM systems present a wealth of new queueing and scheduling challenges that expand on problems from traditional queueing systems. Although modeling LLM systems remain largely unexplored in the queueing community, we believe that queueing theory insights may lead to better scheduling systems. We briefly describe some of the problems and issues in LLM scheduling here and elaborate on them later in the paper.

One issue for LLM systems is that they operate with multiple goals; cost (e.g., computational and financial) and response quality play a critical role alongside traditional performance metrics, like latency and throughput. Optimizing across goals leads to interesting challenges.

Another challenge that LLM inference introduces that is not present in standard queueing models is the need for an ever-growing key-value (KV) cache (Pope et al. 2023). The KV cache is crucial for reusing intermediate computations, improving efficiency by avoiding redundant recalculations at each step of autoregressive token generation. In autoregressive models, each token is generated sequentially, and each new token is conditioned on all previously generated tokens. Thus, the cache accumulates more data as the response grows. However, its presence introduces memory constraints that complicate scheduling as each request consumes graphics processing unit (GPU) memory that cannot be freed until the request is completed.

Moreover, preemption, where a running job can be interrupted to prioritize a more urgent request, is nontrivial in LLM inference because of KV cache management. In LLM systems, preempting a request requires dealing with its allocated KV cache memory, which must be kept in GPU memory (which uses space that other jobs may need), deleted from memory (which requires recomputation), or transferred to the central processing unit (CPU) memory (which incurs some cost in time). The high memory footprint of LLMs combined with the inefficiencies of frequent cache transfers makes preemption costly yet necessary to prevent long-running requests from blocking shorter ones and to mitigate head-of-line blocking effects. These complexities highlight the need for scheduling strategies designed explicitly for LLM inference, accounting for memory constraints.

LLM systems also vary in complexity depending on their components and deployment settings. As an example, requests in LLM inference progress through two distinct phases: a prefill phase and a decode phase. The prefill phase is compute bound; the decode phase is memory-bandwidth bound. The scheduler must efficiently balance these phases. More complex systems, like compound AI systems, integrate multiple interacting components rather than relying on a single model. LLMs in these systems often issue external application programming interface (API) calls for retrieval or computation, introducing new scheduling constraints and the need to decide how to manage the KV cache during API calls. Multiple LLM systems provide further problems. In some settings, multiple LLMs of different sizes are available, requiring a routing strategy to balance cost, response time, and answer quality. Other systems involve multistep pipelines, where requests passing through multiple LLMs in sequence create dependencies that influence scheduling.

Finally, new LLM reasoning systems introduce additional scheduling challenges. These systems extend traditional inference by generating structured reasoning steps before reaching a final answer. Reasoning-based systems can benefit from evaluating early results to determine whether continued computation is necessary or if resources should be reallocated to other tasks. Additionally, some requests may resolve quickly, whereas others require extensive exploration. As a result, the notion of request size extends beyond the token count to include the number of reasoning steps required for convergence.

The following sections provide a survey and suggest open problems in these areas. Section 2 surveys recent works on using predictions for scheduling in single queues and queueing systems and outlines open problems and research directions. Section 3 provides background on LLM systems, discusses their challenges, and explains how they differ from standard queueing models. Sections 4, 5, and 6 deal with LLM systems. We examine three types of LLM systems that offer challenges for work in scheduling and load balancing: (1) a single instance of an LLM, (2) compound AI systems, and (3) reasoning LLM systems. Section 7 concludes with our summary.

2. Scheduling with Job-Size Predictions

2.1. Extensions of Standard Queueing Models

A natural starting point for considering scheduling with job-size predictions is the standard $M/G/1$ queue. Typically, such queues use FIFO scheduling when job sizes[3] are not known and use SRPT when job sizes are known. These scheduling policies are well understood, as are related policies that use known job sizes, such as SJF or preemptive shortest job first (PSJF). It seems natural to extend these standard size-based scheduling policies to the setting where job sizes are not known exactly but instead, predicted.

Mitzenmacher (2020) considers a simple model that is in the spirit of traditional queueing theory problems where jobs may have priority classes. Instead of just each job's size being modeled as independently selected from a fixed distribution, the model is extended, so now, each job independently has both a size and a predicted size selected from a fixed (two-dimensional) distribution. That is, there is a density function $g(x,y)$ for the probability that a job has service time x and predicted service time y. Given this model for predictions, the equations for the expected response time of the variants of SRPT, SJF, and PSJF that use the predicted sizes to schedule jobs are derived. (There is no settled naming convention for these variants, but following Mitzenmacher (2020), we will refer to these as shortest predicted remaining processing time (SPRPT), shortest predicted job first (SPJF), and preemptive shortest predicted job first (PSPJF).)

It is not clear what are realistic models for predicted job times. The work by Mitzenmacher (2020) introduces and studies some artificial prediction distributions that are mathematically natural, including where a job with true job size x has a predicted job size that is exponentially distributed with mean x or uniformly distributed in the range $[(1-\alpha)x,(1+\alpha)x]$ for some parameter α.

As an example, Table 1, taken from the results in Mitzenmacher (2020), shows the expected response time in equilibrium for SPJF and SPRPT and compares them with the expected response times for FIFO, SJF, and SRPT. The primary takeaway from this example is that although using predictions is naturally not as good as using exact information, it provides significant gains over using FIFO, which does not use any information about job times.

Although Mitzenmacher (2020) derives equations for these prediction-based policies directly, following similar previous derivations for the standard variants without predictions (see, e.g., Harchol-Balter 2013), it should be noted that these particular prediction-based scheduling schemes are amenable to analysis using the

Table 1. Results from Equations (to Four Decimal Places) for FIFO, Shortest Job First, Shortest Predicted Job First, Preemptive Shortest Job First, Preemptive Shortest Predicted Job First, Shortest Remaining Processing Time, and Shortest Predicted Remaining Processing Time, Where Service Times Are Exponential with Mean 1 and the Predicted Time for a Job with Size x Is Exponentially Distributed with Mean x

λ	FIFO	SJF	SPJF	PSJF	PSPJF	SRPT	SPRPT
0.5	2.0000	1.7127	1.7948	1.5314	1.6636	1.4254	1.6531
0.6	2.5000	1.9625	2.1086	1.7526	1.9527	1.6041	1.9305
0.7	3.3333	2.3122	2.5726	2.0839	2.3970	1.8746	2.3539
0.8	5.0000	2.8822	3.3758	2.6589	3.1943	2.3528	3.1168
0.9	10.0000	4.1969	5.3610	4.0518	5.2232	3.5521	5.0481
0.95	20.0000	6.2640	8.6537	6.2648	8.6166	5.5410	8.3221
0.98	50.0000	11.2849	16.9502	11.5513	17.1090	10.4947	16.6239
0.99	100.0000	18.4507	29.0536	18.9556	29.3783	17.6269	28.7302

Source. Mitzenmacher (2020).
Note. The symbol λ denotes the arrival rate.

Schedule Ordered by Age-based Priority (SOAP) methodology of Scully et al. (2018) (see also Scully and Harchol-Balter 2018). The SOAP methodology requires that jobs be serviced according to some ranking function that depends only on a job's "type" and the amount of time that it has been served. A job's type could correspond to a job class in a system with job classes, the job's size, or both. In this setting, a job's type corresponds to the pair (x, y) representing its actual and predicted service size. When scheduling by shortest predicted remaining processing time, for example, if a job has been served for a units of time, its rank is simply $y - a$ (the predicted remaining service time) as long as $a \le x$ (because after being served for time x, the job completes). Figure 1 shows the rank functions of size-estimate-based policies. The SOAP methodology provides a very general (albeit sometimes difficult) approach for analyzing $M/G/1$ queueing variants using predictions as long as the conditions for using SOAP are satisfied, which is a significant limitation. For example, the SOAP methodology cannot be applied when the scheduling policy depends on how many jobs are in the queue awaiting service.

2.1.1. Open Questions

- Are there natural models of predictions and the resulting prediction errors in queueing that are realistic across a range of problems and/or particularly worthy of future study?
- Can we design a tool that readily numerically computes results for standard queueing policies for $M/G/1$ queues given a prediction model in some standardized form?
- Are there analysis approaches (extending SOAP or otherwise) to deal with more general settings with predictions, such as using predictions only when the number of items in the queue is sufficiently high?
- The Gittins policy (Scully and Harchol-Balter 2021) should be optimal in this setting. Are there conditions under which implementing a Gittins policy would be natural?
- Here, we have described predictions as being real valued. A prediction, however, could take the form of a predicted distribution for a job as opposed to a value (see, e.g., Dinitz et al. 2024). The analysis of effective scheduling approaches when predictions take the form of distributions remains open.
- Predictors can, for various reasons, possibly become poor or degrade. Can we model systems where predictions are monitored and possibly ignored if they appear sufficiently incorrect? In such a system, there may be a mix of predicted and unknown service times.

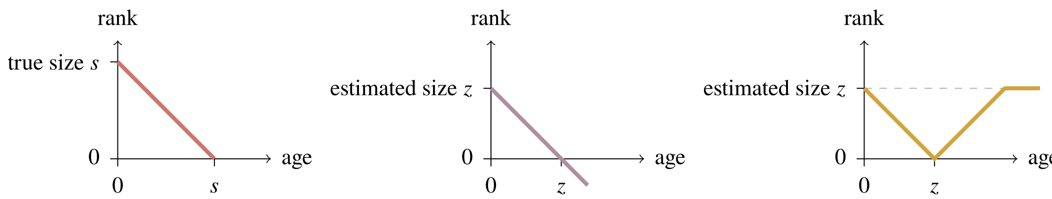
2.2. One-Bit Predictions

Additional recent work continues to develop the use of predictions. Mitzenmacher (2021) studies the viability of using one-bit predictions (which correspond to two-class classifiers), where the bit corresponds to prediction that leads to a simple implementation; short jobs can be placed at the front of the queue, whereas long jobs can be placed at the back of the queue. Such an implementation could be preemptive, so a new short job preempts any running job or not.

From the theoretical standpoint, one-bit predictions lead to a mathematically more tractable system. For example, consider the standard $M/M/1$ model with Poisson arrivals of rate λ and where jobs have exponentially distributed service times with mean 1, but now, we add the exponential prediction model. A job with true service time x can be thought of as having a predicted time y, where y is exponentially distributed with mean x; if $y > T$, the job is marked as long, and otherwise, it is marked as short. The response time without preemption is shown to have the form

$$t_1 = \frac{\lambda(1 - \lambda(1 - 2\sqrt{T}K_1(2\sqrt{T})))}{(1 - \lambda)(1 - \lambda(1 - 2TK_2(2\sqrt{T})))} + 1,$$

Figure 1. Rank Functions of Size-Estimate-Based Policies



Notes. The rank function for SRPT is in the left panel; the rank decreases as $s - a$, where s is the true size and a is the age. The rank function for SPRPT is in the center panel; it decreases as $z - a$, where z is now the estimated size. Note that a job can have negative rank, at which point it cannot be preempted. The SPRPT-with-bounce rank function from Scully et al. (2022) is in the right panel; the rank decreases from the estimate z to zero but bounces back up according to the function $\max(|z - a|, z)$. This rank bounce tempers the effect of long jobs that are predicted to be short, delaying short jobs from being served.

Table 2. Simulation Results (Except for FIFO) for Exponentially Distributed Service Times Using Exponential Predictions and the Optimal Threshold

λ	FIFO	Threshold no preempt	Threshold preempt	SRPT	Prediction no preempt	Prediction preempt	SPRPT
0.50	2.000	1.783	1.564	1.425	1.850	1.698	1.659
0.60	2.500	2.089	1.814	1.604	2.209	2.013	1.940
0.70	3.333	2.542	2.203	1.875	2.761	2.517	2.369
0.80	5.000	3.329	2.910	2.355	3.757	3.451	3.143
0.90	10.00	5.278	4.755	3.552	6.366	5.960	5.097
0.95	20.00	8.535	7.914	5.532	10.848	10.372	8.424
0.98	50.00	16.495	15.735	10.436	22.418	21.909	16.696

Source. First published in Mitzenmacher (2021). Copyright © 2021 by SIAM.

and the response time with preemption is shown to be

$$t_2 = \frac{1-\lambda+\lambda 2\sqrt{T}K_1(2\sqrt{T})}{(1-\lambda)(1-\lambda(1-2TK_2(2\sqrt{T})))},$$

where K_1 and K_2 are modified Bessel functions of the second kind (with different parameters, one and two). In this particular setting, preemption is always helpful; in fact, $t_1 = \lambda t_2 + 1$. We suggest that it is perhaps surprising that these models yield equations with such compact closed forms, albeit in terms of the arguably somewhat obscure modified Bessel functions. The paper derives other interesting closed forms for cases where predictions are uniform over $[0, 2x]$ for a job of size x and where the service distribution is a particular Weibull distribution.

Tables 2 and 3, based on data from Mitzenmacher (2021), consider simulation results for schemes without predictions and with predictions. The schemes labeled threshold classify jobs as short or long based on the actual service time (without prediction), showing the impact of having one bit of accurate advice as a comparison point. They consider the $M/G/1$ setting with exponentially distributed service times and service times governed by a heavy-tailed Weibull distribution (with cumulative distribution function $F(x) = 1 - e^{-\sqrt{2x}}$). Perhaps not surprisingly, one-bit predictions obtain a large fraction of the benefit of full predictions (which corresponds to SPRPT) in the cases studied.

Another interesting point is that predictions are even more significant in the context of the heavy-tailed Weibull distribution. Arguably, this is obvious (at least in hindsight), and there is a useful intuition for this. Large queueing delays are caused by longer jobs blocking shorter jobs; this is why the performance of SRPT can be substantially better than FIFO. Predictions, even if they only get the order mostly right, prevent long jobs from blocking shorter jobs very often. However, this result provides information back to those designing machine learning models that the right performance metric for one-bit predictions is not the fraction of correct predictions because predicting long jobs correctly is more important than predicting short jobs correctly. A mispredicted long job can be placed in front of several shorter jobs, blocking them from service and significantly increasing all of their times in the system. A mispredicted short job, however, is only itself hurt when placed at the back of a queue (see also Dell'Amico et al. 2015 for a discussion of this). More generally, even with predictions of the actual service times, it is better to have good predictions for longer jobs.

Chen and Dong (2021) also similarly study a two-class priority rule in which jobs predicted as short preempt those predicted as long, focusing on the analysis of the asymptotic behavior of such a system. Their theoretical

Table 3. Simulation Results (Except for FIFO) for Weibull Distributed Service Times Using Exponential Predictions and the Optimal Threshold

λ	FIFO	Threshold no preempt	Threshold preempt	SRPT	Prediction no preempt	Prediction preempt	SPRPT
0.50	4.000	3.012	1.608	1.411	3.155	1.736	1.940
0.60	5.500	3.676	1.867	1.574	3.918	2.062	2.280
0.70	8.000	4.565	2.258	1.813	4.983	2.568	2.750
0.80	13.00	5.955	2.951	2.217	6.721	3.481	3.519
0.90	29.00	8.940	4.649	3.154	10.630	5.790	5.224
0.95	58.00	13.223	7.448	4.517	16.546	9.846	7.788
0.98	148.0	22.451	15.194	7.666	29.346	20.918	13.404

Source. First published in Mitzenmacher (2021). Copyright © 2021 by SIAM.

and numerical results demonstrate that even with imperfect service-time estimates, the two-class rule delivers substantial gains over FCFS, providing further results regarding the robustness of one-bit prediction schemes.

2.2.1. Open Questions

- What other natural models of prediction errors are there for one-bit predictions?
- Can we formalize how to optimize the predictions that we would like to obtain from machine-learned predictions under some specific scheduling policy, such as SPRPT or one-bit predictions from thresholds?
- One-bit prediction analysis can readily be generalized to k-bit prediction analysis (or from two classes to greater than two classes). How do performance characteristics, such as the expected response time or the behavior of the tail of the response time, vary as k increases? Note that SOAP-based analyses can also be used to analyze the tail of the response time (Scully et al. 2020b), and several recent results have made progress on optimizing the asymptotic tail of response time for $M/G/1$ settings (Charlet and Van Houdt 2024, Harlev et al. 2024, Yu and Scully 2024).

2.3. Uniform Bounds

Although the ability to derive exact formulae for certain standard queueing models with the addition of prediction is valuable, it can sometimes be difficult to gain more general insights from the specific equations. The analysis methods used for online algorithms, where the input arrives data item by data item and the algorithm must react as each item arrives, motivate another approach. Scheduling problems can naturally be seen as online problems, although in the online setting, one typically looks at worst-case inputs rather than stochastic inputs as one generally does in queueing theory. Many problems in online algorithms have been re-examined in the context of learning-augmented algorithms (see, e.g., Mitzenmacher and Vassilvitskii 2020 and Algorithms with Predictions Project 2024), and the two areas fit together quite naturally. Because in online problems, the whole input is not given at the start, achieving an optimal result is not generally possible, and the standard performance measure in online algorithms is the *competitive ratio*, which is the ratio of the value of the solution obtained by the proposed online algorithm and the value of the optimal solution. (For randomized algorithms, the competitive ratio is usually defined as the ratio between the expected value of the solution obtained by the online algorithm and the optimal.) The question becomes as follows. Can the competitive ratio be improved when there is suitable advice?

In the context of online algorithms with predictions, early work defined two key goals: consistency, which requires near-optimal performance with small error, and robustness, which requires bounded approximation ratio under arbitrary error. Formally, as stated in Lykouris and Vassilvitskii (2021), we may say that an algorithm is α-consistent if its competitive ratio tends to α as the error in the predictions goes to zero and β-robust if the competitive ratio is bounded by β, even with arbitrarily bad predictions.

The work by Scully et al. (2022) extends these ideas to the setting of $M/G/1$ queues with predictions. The assumption made is that the predictions have bounded multiplicative error so that a job of size s has predicted size in the range $[\beta s, \alpha s]$ for some $\beta < 1$ and $\alpha > 1$. (Additionally, the job size and the prediction come from a joint distribution as in the previous work; that is, the prediction is not chosen adversarially.) The work first shows that this assumption of bounded multiplicative error is necessary to achieve constant robustness; that is, without bounded multiplicative error, there are cases where robustness cannot be achieved. Accordingly, they set a goal of finding a scheduling strategy with the following properties.

- Consistency. As α and β go to one, the expected response time converges to the expected response time for SRPT (the optimal scheduling algorithm).
- Graceful degradation. The ratio of the expected response time of the system using the scheduling strategy and the expected response time of the system using SRPT is bounded by $C\frac{\alpha}{\beta}$ for some constant C and any α and β.

The first goal is a natural form of consistency in this setting. Graceful degradation (also called smoothness), where the performance bound degrades gracefully with the quality of the prediction, appears to be a useful aim in its own right and is now often considered in works on algorithms with predictions (see, e.g., Brand et al. 2024 and Blum and Srinivas 2025).

Summarizing their work, there are several key points.

- Simply using SPRPT does not yield expected response times (time in system) bounded within a constant factor of SRPT, even with bounded multiplicative errors.
- A variant of SPRPT, where the job's rank increases again after reaching zero, is both consistent and provides graceful degradation (with a constant C of 3.5).
- PSPJF also yields graceful degradation, and the constant C proven for it is in fact better than the constant proven for the SPRPT variant (here, $C = 1.5$ is proven).

The analyses utilize ideas from SOAP analysis along with work integral methods designed by Scully et al. (2018, 2020a). In particular, there is a careful comparison of rank functions to bound the expected response time of the new SPRPT variant.

It is worth noting that the more traditional worst-case scheduling problems have similarly been studied as online algorithms problems. Most similarly, Azar et al. (2021) examine the classic online problem of scheduling on a single machine to minimize total flow time when job times may be distorted by up to a multiplicative factor of μ. Their first work (Azar et al. 2021) shows that for every distortion factor μ, there is an $O(\mu^2)$-competitive algorithm, but the algorithm needs to know μ in advance. In later work (Azar et al. 2022), they improve this to provide a specific $O(\mu \log \mu)$-competitive algorithm that does not know μ in advance. These works, therefore, obtain similar results to Scully et al. (2022) without making stochastic assumptions on the job sizes but with a more complex algorithm and a slightly larger-than-linear competitive ratio.

More recently, Moseley et al. (2025) also consider an unknown-size setting similar to Scully et al. (2022), although their analysis focuses on a batch model rather than a queueing system. Specifically, they study the preemptive scheduling of a fixed batch of stochastic jobs when only imperfect predicted size distributions are available. They show that the classic Gittins index policy, although optimal under perfect predictions, can suffer unbounded performance degradation under small distributional errors. To address this, they propose a robust variant of the Gittins policy that incorporates a measure of error between the true and predicted distributions and prove that it achieves near-optimal mean completion time when predictions are sufficiently accurate.

2.3.1. Open Questions

- Can we tighten the various bounds of Scully et al. (2022), in particular improving the constants C related to graceful degradation?
- Could more complex scheduling algorithms, going beyond rank-based algorithms, yield better performance bounds?
- The results focus on the expected response time. Are there other important performance measures, and how should they be evaluated in this setting?
- Can we weaken the assumption of bounded multiplicative error and obtain similar results to Scully et al. (2022)? For example, could some bound on the prediction error's variance suffice?

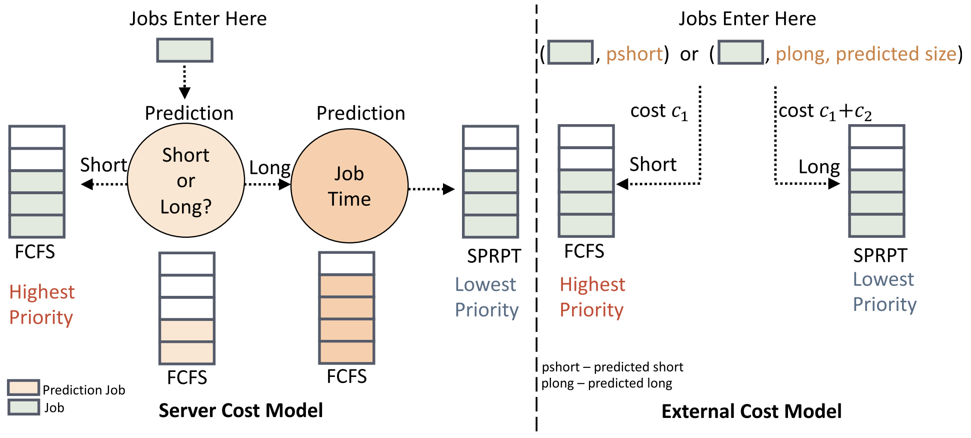
2.4. Accounting for Prediction Costs

The results presented above demonstrate the great potential for using predictions to improve scheduling performance. However, the models that we have discussed all suffer a somewhat glaring flaw; they do not model the resources required to obtain such predictions but instead, assume that predictions are provided "for free" when a job arrives. To be fair, this assumption is not unusual in the area of algorithms with predictions more generally as the prediction cost may be small in the context of the algorithm or the focus may be on a different performance metric. For scheduling in particular, however, assuming that prediction costs can be ignored may not be realistic as the resources devoted to calculating predictions might be more effectively used to directly serve the jobs themselves. This perspective challenges the potential effectiveness of integrating predictions into real-world queueing systems.

This point is considered in Shahout and Mitzenmacher (2024), which incorporates costs into the analysis of $M/G/1$ queues with predictions and considers novel scheduling approaches that take these costs into account. In that work, they consider two models of costs. In the first model, referred to as the external cost model, predictions are provided by some external process and do not affect job service time, but there is a fixed cost for predictions. The expected cost per job in this model would naturally be taken as the sum of the job's expected response time within the system and the prediction costs. With this model, one might consider only using predictions for some jobs but not others. In the second model, referred to as the server time cost model, predictions themselves require a fixed time from the same server that is servicing the jobs, and hence, a scheduling policy involves also scheduling the predictions. The expected cost per job in this model is just the expected response time. Note that because the predictions require work from the server, there are more complex interactions; in particular, for heavily loaded systems, the time used for jobs to obtain predictions could lead to an overloaded, unstable system.

As a starting point, Shahout and Mitzenmacher (2024) derives the formulas for SPRPT and one-bit predictions under both cost models, which can again be done using an SOAP analysis (Scully et al. 2018). However, Shahout and Mitzenmacher (2024) argues that the introduction of costs allows for more interesting models and scheduling strategies. They focus on a setting where one-bit predictions are cheap compared with a prediction of the service time. In such a setting, it may make sense to use the cheap one-bit prediction for all jobs but only use the

Figure 2. SkipPredict Framework Under the Server Cost Model (Left Panel) and the External Cost Model (Right Panel)



more expensive prediction for long jobs. Predicted short jobs are scheduled by FIFO and predicted long jobs are scheduled after short jobs using SPRPT. They call this scheduling algorithm SkipPredict (Figure 2) and show that it also can be analyzed using SOAP analysis.

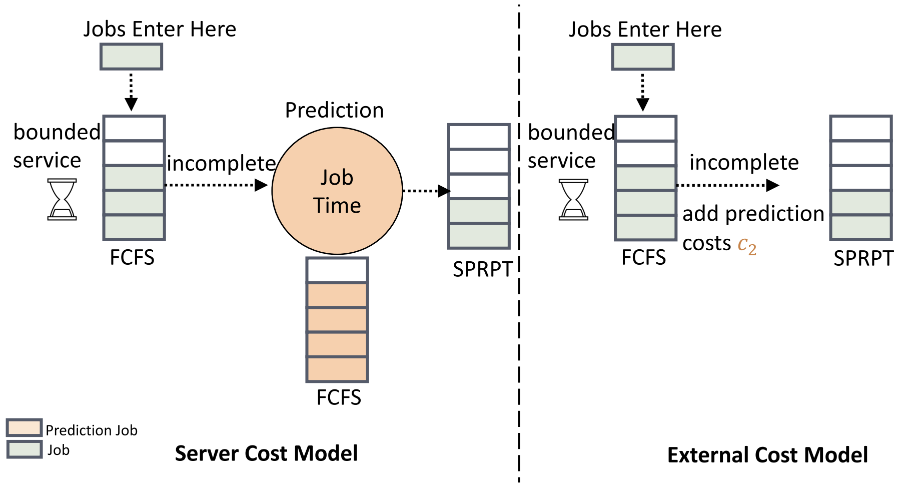
The SOAP analyses provided are somewhat complex as they utilize two-dimensional ranking functions to provide priorities. For example, for SkipPredict in the server cost model, short jobs always have highest priority followed by one-bit predictions for jobs that have entered and not received such predictions. Then, priority goes to predictions for long jobs, and long jobs themselves have the lowest priority. This prevents known short jobs from being delayed by other jobs and ensures that long jobs are handled by SPRPT when long jobs are being serviced.

As another alternative, they analyze a scheduling algorithm, DelayPredict, that avoids cheap predictions entirely but still limits the jobs that undergo more expensive predictions of the service time. DelayPredict initially schedules all jobs in an FIFO manner, but instead of using cheap predictions, it limits each job to a limit L of time, at which point the job is preempted and treated as a long job. At that point, the job will be deprioritized and queued for prediction; longer jobs are then served by SPRPT. DelayPredict provides an alternative to SPRPT that avoids the cost of predictions for every job and can lead to improvements if one-bit predictions are either not much cheaper than full predictions or not available. The DelayPredict framework is shown in Figure 3.

2.4.1. Open Questions

- Are there other natural models of prediction costs for queueing systems? For example, rather than having a separate prediction stage, one could obtain a prediction as the job runs at some cost (slowdown) of the job for some initial time period.

Figure 3. DelayPredict Framework Under the Server Cost Model (Left Panel) and the External Cost Model (Right Panel)



- When predictions have cost, they may not be worthwhile when the system is heavily loaded. Can we design and analyze scheduling policies that choose when to use predictions based on the current load or otherwise, respond dynamically in choosing when to use predictions?
- How can selective prediction strategies be integrated into scheduling to balance prediction cost and performance? Specifically, can we design a scheme where only a subset of jobs is predicted (e.g., with some probability), whereas jobs without predictions are handled via FIFO?
- Can we design systems that use dedicated prediction servers effectively?
- In Grosof and Mitzenmacher (2022), a different notion similar to cost is considered; jobs with incorrect predictions are possibly punished by being assigned lower priority. This model is used to examine incentive compatibility in the context of self-reported predictions. Are there other ways of adding economic considerations to refine prediction models?

2.5. Multiple-Server Systems

The above work has focused on using predictions in the setting of the $M/G/1$ queue. Using predictions in larger systems or networks of queues remains relatively open for further study. Most previous work has been empirical (Dell'Amico et al. 2015, Mailach and Down 2017, Dell'Amico 2019), such as the work by Mitzenmacher and Dell'Amico (2022), which looks at the power of two choice paradigm (Vvedenskaya and Suhov 1997, Mitzenmacher 2001) when using predictions. However, recent work by Dong and Ibrahim (2024) has examined the asymptotic performance of shortest predicted job first with noisy estimates of job sizes for $M/GI/s{+}GI$ systems, showing that it asymptotically maximizes the throughput among all nonpreemptive scheduling disciplines that use that noisy service-time information. The lack of theoretical work thus far is arguably not surprising; multiple server systems resist analysis even without predictions, and there are many open questions for systems with multiple servers that become only more challenging when adding predictions. (For recent work on multiple server systems generally, see, for example, Grosof (2024).)

Another small step forward appears in the work on one-bit predictions (Mitzenmacher 2021), where a variant of the power of two choices is considered in the setting of one-bit predictions by deriving the fluid limit differential equations. In this setting, each job chooses d queues uniformly at random, and the job chooses to wait at the best of the d choices for some defined notion of best. (In Mitzenmacher (2021), the decision is based on the number of jobs of the same predicted type that are queued.) The fluid limit system corresponds to the number of queues going to infinity. This analysis requires Poisson arrivals and exponentially distributed service times for the long and short jobs. The key to this analysis is that the set of queue states has a short description; the state can be represented by the number of queued jobs that are predicted to be short and long and whether the current running job is short or long. Mitzenmacher (2021) also provides an interesting example where the prediction error rate is greater than 50% over all jobs, but using predictions still performs better than not using predictions in the fluid limit. As one might expect, in this example, the prediction error rate is made large for short jobs and smaller for long jobs, showing the importance of predicting long jobs accurately.

2.5.1. Open Questions

- Can we generalize any existing theoretical work on multiple server systems to systems with predictions naturally?
- In particular, can we develop fluid limit models to analyze the power of two choices when using (more than one-bit) predictions?
- Another area where predictions may be useful is in multiserver-job systems, where jobs may run concurrently on many servers. As a challenging example, consider a setting where jobs may use a variable number of servers with the running time dependent on the number of servers used, and there are predicted times associated with each possible number of servers that the job could use. That is, one is predicting the speedup obtainable for a job by using more servers, which may vary depending on the job type.

3. Large Language Model Systems

LLMs have revolutionized artificial intelligence by moving beyond predicting tokens to solving adaptive problems. Their impact spans across professional sectors, from healthcare and finance to education and customer support, where they enable decision support and personalized interactions. In the creative and technical domains, LLMs can generate artistic content (Ramesh et al. 2021), automate code development (Chen et al. 2021), and accelerate scientific research through data analysis and literature synthesis (Jo 2023). ChatGPT (Achiam et al. 2023) exemplifies how LLMs can enhance AI capabilities through natural dialogue, enabling users to engage with AI systems for tasks ranging from simple queries to complex problem-solving.

In high-concurrency environments, users expect real-time responses, which make minimizing latency essential for a seamless experience. Scheduling can address this latency optimization challenge by minimizing the average user response time (the time from when a request first arrives until it completes service) or by affecting the tail of the response time distribution. The scheduler decides which requests to queue or serve and how to order the requests within each queue. (Because many applications rely on pretrained models for inference instead of training their own, we focus on scheduling during inference, although scheduling considerations may also apply to training LLMs.) With respect to minimizing use response time, scheduling in LLMs feels similar to queueing theory models, like those we have discussed. However, with LLMs, the scheduler might have additional goals to optimize for, such as throughput and cost, introducing further challenges.

Predictions can naturally help inform scheduling decisions for LLMs as many job details may be unknown on arrival. Ideally, these predictions are made before running a request, but they can also be refined during execution as the system gathers insights from the LLM itself. Also, as we discuss further later, predictions in the LLM setting can extend beyond estimating request sizes.

To explain how queueing models apply to LLM systems, we first provide background on LLM inference and hardware execution. We then discuss how LLM systems differ from traditional queueing systems and the challenges that these differences introduce. We believe both that the research community needs to develop new queueing theory models tailored to LLM systems and that existing queueing theory can provide both analysis methods and heuristic insights that can improve LLM scheduling. In the following sections, we present the inference process in detail. Our goal is to provide a faithful description of the system to facilitate future modeling efforts by the queueing theory community.

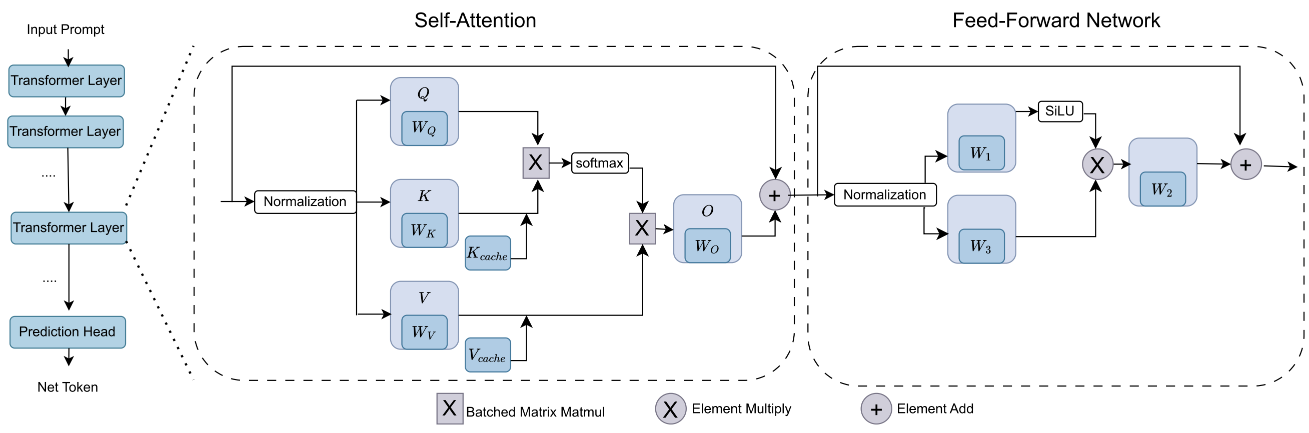
3.1. LLM Inference

LLM models follow an autoregressive pattern, where text is generated one token (i.e., one or more words or parts of words) at a time based on the calculated probability distribution for the next token given the preceding context. This process, referred to as inference, consists of sequential iterations where each iteration generates a token and appends it to the existing input prompt. The generation continues until a termination condition, such as a predefined maximum output length or an end-of-sequence token, is met.

3.1.1. Transformer Architecture. Most of today's LLMs adopt a decoder-only transformer architecture (Radford et al. 2019, Brown et al. 2020). The input to a transformer layer is an embedding of the tokenized input sequence. Each transformer model consists of a stack of sequential layers, where each layer applies self-attention mechanisms to capture contextual relationships between tokens and feed-forward networks (FFNs) to refine the representation of the input as shown in Figure 4.

3.1.2. Self-Attention Mechanism. The self-attention mechanism enables the model to weigh the importance of different tokens in the input sequence when making predictions. For a given input $X_{\text{pre}} \in \mathbb{R}^{n \times d}$, where n is the

Figure 4. Transformer Architecture



Note. SiLU, sigmoid linear unit.

sequence length and d is the hidden dimension (the embedding size of each token), the model applies learned linear transformations to produce the query ($\boldsymbol{Q}_{\text{pre}}$), key ($\boldsymbol{K}_{\text{pre}}$), and value ($\boldsymbol{V}_{\text{pre}}$) matrices:

$$\boldsymbol{Q}_{\text{pre}} = \boldsymbol{X}_{\text{pre}}\boldsymbol{W}_q, \quad \boldsymbol{K}_{\text{pre}} = \boldsymbol{X}_{\text{pre}}\boldsymbol{W}_k, \quad \boldsymbol{V}_{\text{pre}} = \boldsymbol{X}_{\text{pre}}\boldsymbol{W}_v,$$

where $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v \in \mathbb{R}^{d\times d_k}$ are learnable weight matrices that project the input embeddings into a lower-dimensional space of size d_k that represents the number of columns in $\boldsymbol{K}_{\text{pre}}$ and determines the size of the query-key dot product.

These queries, keys, and values are used to compute the attention output through the following formula:

$$\boldsymbol{O}_{\text{pre}} = \text{softmax}\left(\frac{\boldsymbol{Q}_{\text{pre}}\boldsymbol{K}_{\text{pre}}^T}{\sqrt{d_k}}\right)\boldsymbol{V}_{\text{pre}}\boldsymbol{W}_o + \boldsymbol{X}_{\text{pre}},$$

where $\boldsymbol{W}_q$, $\boldsymbol{W}_k$, $\boldsymbol{W}_v$, and $\boldsymbol{W}_o$ are the learnable weight matrices. The softmax function ensures that attention weights sum to one across each row, allowing the model to assign different importance levels to tokens. It is defined as $\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$, where each z_i represents an element of the input matrix.

3.1.3. Feed-Forward Network.

The output of the self-attention module is sent to the feed-forward network, which refines the attention output. This network introduces nonlinearity, allowing the model to capture more complex patterns in the data:

$$\text{FFN}(\boldsymbol{x}) = (\text{SiLU}(\boldsymbol{x}\boldsymbol{W}_1) \times \boldsymbol{x}\boldsymbol{W}_3)\boldsymbol{W}_2,$$

where $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, and $\boldsymbol{W}_3$ are linear modules. The sigmoid linear unit (SiLU) activation function is defined as $\text{SiLU}(x) = x\cdot\sigma(x) = x\cdot\frac{1}{1+e^{-x}}$, where $\sigma(x)$ is the sigmoid function defined as $\frac{1}{1+e^{-x}}$.

3.1.4. Prefill and Decode Phases.

LLM inference is divided into two phases: *prefill* and *decode*. The prefill phase is the initial step, where the model processes the input prompt ($\boldsymbol{X}_{\text{pre}}$) and generates key-value pairs that are stored in the KV cache, which holds contextual information required for generating subsequent tokens. The design of the transformer allows for parallel processing of input tokens during the prefill phase. In the decode phase, new tokens are generated based on previous tokens step by step. The input of this phase is $\boldsymbol{X}_{\text{dec}} \in \mathbb{R}^{1\times d}$, and the model retrieves previously stored key-value pairs from the KV cache to continue generating tokens where after each generated token, new key-value pairs are computed and appended to the existing cache:

$$\boldsymbol{Q}_{\text{dec}} = \boldsymbol{X}_{\text{dec}}\boldsymbol{W}_q, \quad \boldsymbol{K}_{\text{cat}} = [\boldsymbol{K}_{\text{cache}}, \boldsymbol{X}_{\text{dec}}\boldsymbol{W}_k], \quad \boldsymbol{V}_{\text{cat}} = [\boldsymbol{V}_{\text{cache}}, \boldsymbol{X}_{\text{dec}}\boldsymbol{W}_v].$$

The attention output in the decode phase is computed as

$$\boldsymbol{O}_{\text{dec}} = \text{softmax}\left(\frac{\boldsymbol{Q}_{\text{dec}}\boldsymbol{K}_{\text{cat}}^T}{\sqrt{d_k}}\right)\boldsymbol{V}_{\text{cat}}\boldsymbol{W}_o + \boldsymbol{X}_{\text{dec}}.$$

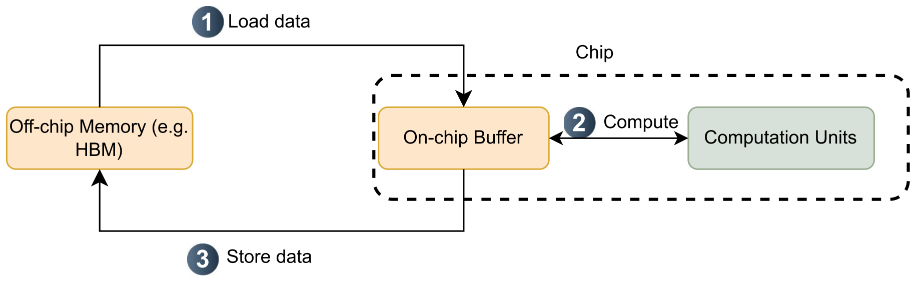
Storing and loading the intermediate attention matrix increase inference time in transformer architectures. FlashAttention (Dao et al. 2022, Dao 2023) and Flash-Decoding (Dao et al. 2023) combine matrix multiplications and the softmax operator in self-attention into a single operator. This integration eliminates the need to store and load the intermediate matrix, reducing both memory access overhead and inference time. The KV cache grows over the iterations for inference, introducing unique challenges in optimizing latency—a consideration specific to inference as the KV cache is not present during training.

3.1.5. Execution on Hardware.

The rapid progress in LLMs is closely tied to advancements in hardware accelerators, particularly GPUs. Unlike traditional CPUs, which execute tasks sequentially, GPUs are optimized for parallel processing. They consist of thousands of small cores capable of performing parallel computations, making GPUs particularly effective for matrix and vector operations that are fundamental to transformer-based models. A key feature that enhances GPU performance is the use of high-bandwidth memory (HBM), which enables faster data transfer between memory and processing units. As illustrated in Figure 5, executing a neural network layer on GPU involves three steps: transferring data (e.g., model weights and KV cache) from memory (such as HBM) to on-chip buffers, performing computations within the on-chip processing units, and writing the results back to memory. The efficiency of this process is influenced by both memory access speed and the computational capacity of the processing units.

The imbalance between computation and memory access may, however, lead to performance bottlenecks. When a layer involves significant computations but minimal memory access, it creates a *computation bound*, leaving memory units idle, whereas computations are processed. Alternatively, a *memory-bandwidth bound* arises

Figure 5. A Neural Network Layer Is Executed on Hardware Devices by Transferring Data from Memory (e.g., HBM) to On-Chip Buffers, Then Computing with the On-Chip Processing Units, and Eventually, Sending the Output Data Back to Memory



when a layer demands extensive memory access but performs fewer computational tasks, resulting in underutilized GPU processing cores.

The prefill and decoding phases differ in their use of computation and memory. The prefill phase efficiently uses GPU parallelism as each input is processed independently and all inputs are available up front. This makes the prefill phase compute bound. In contrast, the decode phase is memory-bandwidth bound. During decoding, significant GPU memory bandwidth is used to load model parameters, often making data transfers to the compute cores slower than the actual token processing. Batching multiple requests during the decode phase helps to reduce this bottleneck by loading model parameters once and reusing them for multiple inputs, which increases throughput and reduces inference costs. Thus, LLM inference throughput depends heavily on the number of requests that can be batched into the GPU's high-bandwidth memory.

3.2. Performance Metrics

LLM systems extend traditional performance metrics while introducing new considerations. Although model size strongly influences performance, it is not the only factor. Key metrics include the following.

- Computational cost. Depends on the model's size (its parameter count) and the length of the generated response, especially given the autoregressive nature of LLM inference.
- Latency (or response time). The time from request arrival until service completion. Latency can be measured in two ways: (a) overall latency, which is affected by model size as well as prompt and output lengths, and (b) time to first token (TTFT), which is the time from request arrival until the first token is produced, primarily influenced by model size and prompt length.
- Throughput. The number of tokens generated per second.
- Accuracy. Although larger models often produce higher-quality responses, the link between model size and accuracy varies with the request type.
- Energy (or carbon footprint). The energy consumed and the resulting carbon emissions during LLM inference, typically measured in kilowatt-hours for energy and grams of CO_2 per request or per token. Key factors include hardware specifications (e.g., GPU type), model size, batch size, and data center location. LLMCarbon (Faiz et al. 2023) offer estimates of LLM carbon footprints.

We focus on the question of reducing overall latency (which we refer to henceforth as just latency). Although using a smaller model can lower latency, it may compromise accuracy. Our aim is to develop scheduling policies that minimize response times without altering the chosen LLM model, which we take as a given. Problems where model selection or modification enables an accuracy-latency trade-off are an interesting direction mentioned in Section 5.

We define a request's latency as the time from when a user submits the request until the answer is returned. In an LLM system, two additional metrics are used to evaluate the system's performance in handling requests. TTFT measures how long it takes to generate the first token, which often depends on the prompt length (i.e., the prefill phase). Time per output token (TPOT) measures the time required to generate each subsequent token (i.e., one decode phase). For a request R, with an input size of n_{input} tokens, an output size of n_{output} tokens, and a waiting time t_{waiting}, the latency of R is defined as

$$t_{\text{response}} = t_{\text{waiting}} + \text{TTFT}(n_{\text{input}}) + n_{\text{output}} \cdot \text{TPOT}. \tag{1}$$

3.3. A Summary: The Job vs. System Perspective

Before going into details regarding LLM scheduling, we summarize the operations that we have described from two perspectives: the job perspective and the system perspective.

3.3.1. Job Perspective. This perspective examines individual jobs by analyzing their journey through processing phases, including the time spent in the queue. Each inference job consists of an input (prompt) and an output, both measured in tokens for convenience. A job undergoes two phases: prefill and decode. During the prefill phase, the model applies learned linear transformations to the input to produce information for the KV cache, with memory usage proportional to the input size. This phase executes as a single computational block because the entire prompt is available. In the decode phase, the model generates tokens sequentially in an autoregressive manner; with each iteration, a new token is produced, and an additional KV cache entry is added, causing the KV cache to grow in proportion to the combined input and output sizes. Preemption in LLM systems is at the token level (Yu et al. 2022); however, preemption poses challenges because the KV cache must be retained until the request is fully processed, or else, previously computed work is lost and must be recomputed. Alternatively, a job may be partially terminated by discarding the most recent (tail) portion of the KV cache.

3.3.2. System Perspective. This perspective considers the management of jobs at the system level through batching, where multiple jobs are served simultaneously. Batching allows the system to load model parameters once and reuse them for multiple jobs, optimizing resource utilization. With continuous batching introduced by Orca (Yu et al. 2022), where new requests can join an existing batch and completed requests are returned immediately at the iteration level rather than waiting for an entire batch to finish, a single batch may comprise jobs in different processing phases (prefill and decode). Batching is performed at the token level; at each token time unit, the system forms a batch of jobs, some in the prefill phase and others in the decode phase, and processes them concurrently via processor sharing using all available computational resources. After all jobs in a batch complete (so either the prefill completes or the decode generates a new token for the job), the system updates the batch by removing completed jobs, adding new ones and possibly preempting existing jobs, with the goal of ensuring continuous resource reallocation to maximize performance. Building on this, vLLM, an LLM serving system (Kwon et al. 2023) integrates paged attention, which allocates the KV cache gradually in blocks during inference instead of allocating for the maximum output length at the beginning, resulting in improved throughput and reduced operational costs. vLLM improves the throughput of popular LLMs by $2-4x$ with the same level of latency compared with Orca and FasterTrasnformer (NVIDIA 2024). vLLM has been widely adopted and established itself as the state-of-the-art framework for inference services.

Finally, we remark that in typical transformer architectures, processing occurs sequentially across neural network layers during both the prefill and decode phases. For scheduling purposes, we treat all layers as a single block and do not consider intralayer scheduling, although exploring this finer granularity remains an interesting possible direction.

3.4. How Do LLM Serving Systems Differ from Standard Queueing Systems?

Standard scheduling approaches often fall short for LLM serving because LLM systems present unique characteristics and challenges not found in typical systems. We summarize some of these key issues.

3.4.1. KV Cache During Inference. During inference, LLMs generate a KV cache that remains in memory until the request is completed. This contrasts with standard queueing systems, where jobs do not maintain a large memory footprint throughout processing and in particular, do not consider memory requirements that grow linearly with the request length. The KV cache reduces computation time but demands substantial memory, proportional to the model's number of layers and hidden dimensions. For instance, a single GPT-3 175B request with a sequence length of 512 tokens requires about 2.3 GB of memory for key-value pairs.

3.4.2. Preemption Overhead. Preemptive scheduling is essential in online settings, where new requests arrive during the execution of longer requests. From a job perspective, processing a single request simplifies KV cache management because there are no competing jobs for memory, even when preemption occurs. In contrast, from a system perspective, batch processing requires careful scheduling to manage the KV cache across multiple simultaneous jobs, and preemptions must be limited to ensure that each job completes without triggering memory overflow.

Given the large memory footprint of LLMs and the limited GPU capacity, this overhead can lead to memory exhaustion. Nonpreemptive policies, such as FCFS, avoid this overhead but often result in higher response times. One approach to mitigate this overhead suggests that inactive KV tensors could be offloaded to CPU memory and reloaded into GPU memory when needed. However, the overhead of offloading and reloading is nontrivial compared with token generation time. For example, deploying GPT-3 175B on NVIDIA A100 GPUs requires approximately 2.3 GB of memory per job for KV tensors. During decoding, token generation takes about 250 milliseconds (ms), whereas transferring KV tensors between host and GPU memory over PCIe 4.0 × 16 at full bandwidth takes about 36 ms. Existing approaches by Wu et al. (2023) and Abhyankar et al. (2024) attempt to optimize offloading and reloading by overlapping these operations with computation. However, the available memory budget for such overlap poses a fundamental constraint.

3.4.3. Multistage Processing. Requests in LLM systems can be viewed as multistage jobs in queueing theory, although they differ from standard queueing models. Mixed-phase batches, where some requests remain in the prefill phase, whereas others have advanced to decoding, can prolong the overall decode phase because the longer prefill phase may dominate the iteration time. Two strategies have been proposed to mitigate this imbalance. The first, *chunked prefill* (Agrawal et al. 2024), splits prompt tokens into smaller segments that are processed alongside decode requests in each batch iteration. In this abstraction, a request is treated as having two parts with distinct processing times. The second strategy, *split phases* (split wise) (Patel et al. 2024, Zhong et al. 2024), separates the prefill and decode phases across different machines, aligning processing resources with the computational demands of each phase. In split phases, different GPUs handle the prefill and decode phases, each operating with its own processing rate and memory constraints. The prefill machine transfers the KV cache to the decode machine, introducing a transfer delay that adds overhead. To mitigate this issue, Patel et al. (2024) optimizes KV cache transfers by leveraging high-speed Infiniband interconnects. This arrangement differs from assuming that all servers have uniform capabilities and can process any job interchangeably. Moreover, distributing these phases across multiple machines requires scheduling decisions regarding where to process each part of a request.

4. Scheduling in LLM Serving

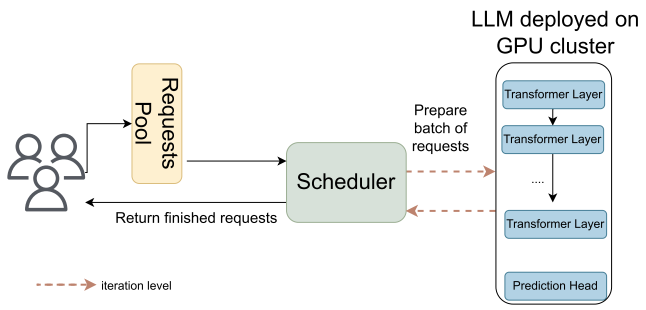
We consider in this section a single LLM deployment that spans one or more GPUs depending on the model's size. When the model's memory footprint requires multiple GPUs, we simplify the scheduling problem by treating the distributed model as a single unit, abstracting away the inter-GPU communication overhead.

4.1. Dynamic Batching and Preemption in LLM Inference

LLM inference systems commonly employ iteration-level scheduling (Yu et al. 2022) (Figure 6), also known as continuous or dynamic batching. Unlike traditional request-level scheduling, where a fixed batch runs to completion before processing a new batch, iteration-level scheduling operates token by token. This approach allows the scheduler to return finished requests to the user and to adjust the batch after each iteration and enables preemptive scheduling at the granularity of individual tokens. After each token is generated, the scheduler evaluates whether to continue processing the current request or switch to another pending request.

Request demands vary; some requests have a short yes/no output, whereas others seek lengthy explanations. This variability poses a challenge because short requests may wait behind larger ones, resulting in longer response times. From a mean response time perspective, it is more efficient to give short requests priority, as in policies like shortest remaining processing time. However, implementing SRPT in LLM systems poses unique challenges. We typically need accurate request sizes to use size-based scheduling, yet these sizes are often unknown. In LLM systems, a request's size depends on both the input size (in tokens) and the output size, and the latter is not known a priori. Predicting output size from a prompt is challenging because LLMs generate text autoregressively; each generated token is appended to the input, dynamically altering the context for subsequent token generation.

Figure 6. Iteration-Level Scheduling



Notes. The scheduler dynamically adjusts the batch and enables preemption decisions. After each token generation, it evaluates whether to continue with the current request or switch to another pending request. Finished outputs are returned to users as soon as they are ready.

Several works propose methods to predict request sizes. Zheng et al. (2024) use an auxiliary LLM model to predict response sizes and then, prioritize requests based on those predictions. Although this strategy reduces response time, it introduces additional computational overhead from the extra model used for size prediction. S^3 (Jin et al. 2023) fine-tunes a bidirectional encoder representations from transformers (BERT) model (Sanh et al. 2019) to predict output sequence sizes from input prompts. The studies of Jin et al. (2023), Cheng et al. (2024), and Stojkovic et al. (2024) address size prediction as a classification task, where predictions correspond to one of several buckets representing a range of sizes, whereas Qiu et al. (2024a, b) use regression-based approaches to estimate a specific size. Although these prediction models are relatively lightweight, their accuracy declines for requests with highly variable execution times. Learning to Rank (LTR) (Fu et al. 2024a) employs a learning-to-rank approach. Instead of predicting the absolute output size of a request, LTR ranks requests based on their output size, allowing the system to prioritize those with fewer remaining tokens. Although ranking is a simpler task than absolute size prediction, it requires training a ranking model in an offline phase. A limitation of LTR is that it ignores the size of the prompt when ranking, considering only the output size. This absence can lead to head-of-line blocking, particularly when a request with a short output is preceded by a lengthy prompt during the prefill phase. Given the difficulties in predicting output sizes for LLMs, FastServe (Wu et al. 2023) is based on multilevel feedback queue scheduling, where each job starts in the highest-priority queue and is downgraded if it exceeds a set threshold, resulting in frequent preemptions. To counter this, FastServe introduces a proactive GPU memory management mechanism that offloads and uploads intermediate states between GPU and CPU. Compared with Orca, FastServe improves the average and tail job completion times by up to 5.1× and 6.4×, respectively.

Preemptive scheduling can further reduce latency in online LLM services, where new requests may arrive during the execution of longer ones. By interrupting a long-running request to serve a shorter one, overall latency is reduced. Yet, a key complication arises from the need to retain the KV cache for any preempted request, consuming scarce memory resources (see Section 3.4).

Trail (Shahout et al. 2025a) addresses both output size prediction and the preemption overhead. For output size prediction, Trail uses the autoregressive nature of LLM output generation. It recycles embeddings from intermediate transformer layers and processes them with a lightweight linear classifier to estimate the remaining output size. This approach, known as *probing* (Hewitt and Liang 2019, Hewitt and Manning 2019, Belinkov 2022), combines the benefits of direct LLM-based predictions with computational efficiency, eliminating the need for a separate size-prediction model. To tackle the preemption challenge, Trail proposes a variant of SPRPT. In standard SPRPT, a newly arrived request with a shorter predicted remaining time preempts the currently running request. In contrast, Trail disables preemption once a request reaches a certain "age" (i.e., a fraction of its predicted total work). The intuition is that during the early or "young" phase of execution, the KV cache is small, making preemption relatively inexpensive in terms of memory overhead. Later, in the "old" phase, a significant amount of memory has been allocated for the KV cache, so it becomes more efficient to complete the request rather than preempt it and later, reallocate the required memory. Consequently, Trail enforces a global threshold of c·predicted request size (with $0 \le c \le 1$), disabling preemption once a request's age exceeds this threshold. Trail reduces mean latency by 1.66× to 2.01× on the Alpaca data set (Taori et al. 2023) and achieves 1.76× to 24.07× lower mean time to the first token compared with vLLM, evaluated on a server with a single NVIDIA A100 80 GB GPU. We note that a simplified $M/G/1$ policy that utilizes the idea of Trail, disabling preemptions at a threshold of $c \cdot$ predicted request size, can be analyzed using SOAP methods as shown in Shahout et al. (2025a). Such analyses can give insight into the trade-offs of SPRPT variants when memory or other issues need to be taken into account.

4.1.1. Open Questions

- How should we rank requests given the split between prefill and decode stages? As a challenging example, consider two requests with the same total size but different prefill and decode phase sizes (e.g., one with short prefill and long decode versus one with long prefill and short decode). The appropriate ranking may depend on the hardware environment and whether split-phase scheduling is used for the phases.
- How can preemption thresholds be dynamically adjusted? Because the effects of preemption depend on factors such as model size, batch size, available memory, and the distribution of incoming requests, what strategies can dynamically tune preemption thresholds based on the sizes (or expected sizes) of batched requests?
- Are there other interesting variants of SPRPT with limited preemptions worth studying, and how do they affect other performance measures (such as tail behavior)?

4.2. Adaptive Scheduling

Although job-level scheduling optimizes the execution of individual requests, many systems operate on usage-based billing models, making operating budgets a crucial design factor. From a system-level perspective, adaptive scheduling in LLM deployments must address cost constraints, heterogeneous hardware, and prompt sharing. Incorporating financial considerations into scheduling algorithms leads to multiobjective optimization formulations that balance low latency with cost efficiency, introducing trade-offs between latency and cost.

Further gains may be achieved by incorporating prompt sharing, which occurs when multiple requests contain overlapping input segments, enabling the system to reuse intermediate KV computations and reduce redundant processing during inference. In many LLM applications, prompts overlap across user requests and often share common prefixes. For example, Srivatsa et al. (2024) reports that 85%–97% of tokens in a prompt may be shared with other prompts. Such sharing occurs in settings such as conversational agents (Anthropic 2024), tool use (Qin et al. 2023, Schick et al. 2023), question answering (Li et al. 2024, Rawal et al. 2024, Wang et al. 2024b), complex reasoning (Wei et al. 2022, Besta et al. 2024), batch inference (Li et al. 2022), in-context learning (Dong et al. 2022), and agent systems (Chen et al. 2023b, Gao et al. 2024, Guo et al. 2024). Juravsky et al. (2024) and Srivatsa et al. (2024) explore methods that leverage shared prompts to improve online LLM inference efficiency. These methods reuse common prefixes to reduce redundant computation. However, these systems focus on reuse of KV cache entries and balance computational load across GPUs using data parallelism without explicitly optimizing for latency. SGLang (Zheng et al. 2025) introduces RadixAttention for efficient KV cache reuse across requests. Instead of discarding the KV cache after each request, SGLang stores both prompts and outputs in a radix tree that supports fast prefix search, insertion, and eviction using an Least Recently Used policy. They show settings where this approach yields up to five times higher throughput than vLLM.

Shared prompts can reduce the cost of the prefill phase when requests sharing the same context are batched; however, it remains unclear how best to order such requests. For instance, consider three requests R_1, R_2, and R_3, where R_3 is small and R_1 and R_2 share a long context. A naive strategy might always prioritize the smaller R_3, but that approach misses the opportunity to batch R_1 and R_2 together. Thus, there is a need to develop adaptive algorithms that continuously weigh prompt overlap, real-time GPU load, and queue dynamics to minimize both latency and resource underutilization. Theoretical modeling and trade-off analysis of these multifactor schedulers represent an important challenge in large-scale LLM deployments.

Another challenge is addressing the divergent latency requirements of interactive and batch requests. Interactive requests require near-immediate responses, whereas long-running or batch requests can tolerate delays but must avoid starvation. Prioritizing short, interactive queries improves responsiveness but risks indefinitely postponing larger tasks under high load. SAGESERVE (Jaiswal et al. 2025) presents a system for serving LLM inference requests with a wide range of service-level agreements (SLAs), which maintains better GPU utilization and reduces resource fragmentation that occurs in isolated resource pools (or silos). The goal is to develop scheduling algorithms that ensure low latency for interactive queries while maintaining the progress of noninteractive workloads. This challenge can be addressed within a multiobjective scheduling framework that balances latency and fairness using queueing theory to prevent starvation.

4.2.1. Open Questions

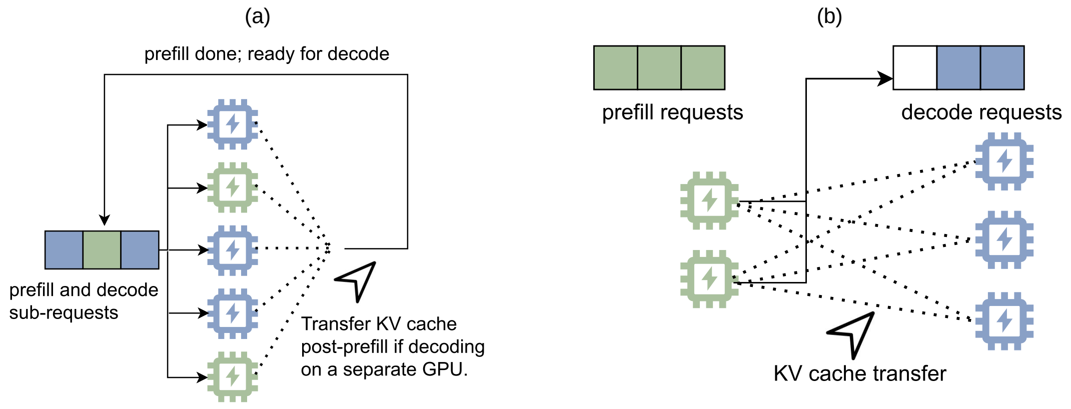
- How can we develop scheduling and resource-allocation strategies that meet a global cost target while ensuring acceptable response times? In particular, we may seek to design an adaptive approach that dynamically adjusts the prioritization of latency and cost based on workload characteristics and SLA requirements.
- How should scheduling policies handle interactive and noninteractive workloads?
- How should prompt sharing be incorporated into scheduling decisions? In scenarios where requests share similar prompts, what strategies can be employed to batch such requests effectively while avoiding delays for stand-alone small requests?

4.3. GPU Resource Allocation

Shifting to the system perspective, we now examine how to orchestrate GPU resources, each with varying computational and memory capabilities, across various architectures. For example, GPUs optimized for *decode* may favor larger memory capacities with relatively lower compute throughput, whereas those optimized for *prefill* may prioritize higher compute throughput with lower memory footprints.

Figure 7 shows two organizations. In the *pooled GPUs* approach (Figure 7(a)), each GPU handles both the prefill and decode phases. After completing the prefill phase, a request can either proceed directly to decoding on the same GPU or re-enter the queue to be processed by another available GPU. The latter option poses a challenge; transferring the KV cache from the prefill GPU to a decode GPU and storing it until the assignment introduce memory and communication overhead. This raises the question of whether to complete the request immediately or preempt it and rank it among waiting requests. The *dedicated GPUs* approach (Figure 7(b)) partitions the pool of GPUs into separate groups: one exclusively handling prefill and the other exclusively handling decode. This

Figure 7. GPU Organizations: Comparing Pooled GPUs with Dedicated GPUs for Prefill and Decode Phases


(a)
prefill done; ready for decode
prefill and decode sub-requests
Transfer KV cache post-prefill if decoding on a separate GPU.
(b)
prefill requests
decode requests
KV cache transfer

Notes. (a) Pooled GPUs. (b) Dedicated GPUs for prefill/decode.

arrangement is similar to tandem servers in queueing systems (Burke 1956, Reich 1957), but here, the GPUs can differ in speed and require KV cache transfer, complicating scheduling further. Although this approach can mitigate conflicts within each phase, it risks underutilization of one subset of GPUs if workload distributions are imbalanced (for example, when large prompt sizes prolong prefill, whereas decode resources remain idle). In heterogeneous GPU environments, mismatches between GPU capabilities and different phases (prefill versus decode) requirements can lead to inefficient resource usage and thus, higher operational costs. Effective scheduling must balance these phases to optimize both cost and latency.

Dynamic hardware availability further complicates scheduling decisions. In many practical settings, additional GPUs become available after an LLM service is already running. The challenge is to integrate these resources without disrupting ongoing workloads and to determine whether they should support the prefill phase to improve the handling of large prompts or the decode phase to reduce response times for interactive queries. Bottlenecks, workload composition, and cost constraints may change over time, making static allocations suboptimal.

4.3.1. Open Questions

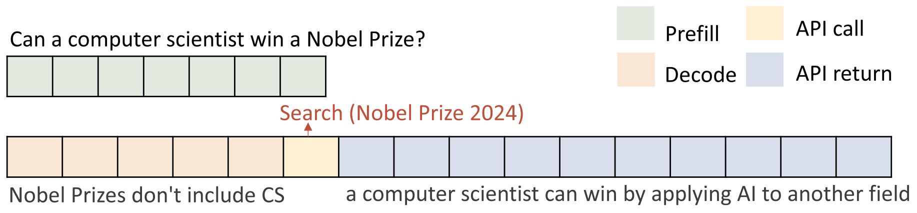
- How should GPU resources be orchestrated across pooled and dedicated organizations?
- How can scheduling policies be designed to balance prefill and decode phases when GPUs have heterogeneous compute and memory capabilities?
- Can results or insights from tandem queues give insight into optimizing the dedicated GPUs approach?
- Can we systematically estimate a number of lower-capacity GPUs, each with limited memory and throughput, required to match or exceed the performance of a single high-end GPU? We may seek to do this because of significant price differentials between low- and high-end GPUs. Beyond raw throughput, this evaluation must account for communication overhead between GPUs, the availability and cost of high-bandwidth interconnects, and cumulative power. Developing theoretical models and empirical studies of these factors will clarify when a single more capable GPU is preferable to a cluster of smaller GPUs, especially for the prefill and decode phases.
- How can we dynamically identify resource-constrained phases and design scheduling algorithms that adapt to changes in GPU availability? One natural approach is to use predictions of request arrival rates and prompt sizes to allocate resources to the most critical phase.

5. Scheduling in Compound AI Systems

In the previous section, we focused on a single LLM deployment. However, AI development is shifting toward compound systems that integrate multiple interacting components—such as external tools, model calls, and retrievers—rather than relying on monolithic models. LLMs are typically trained on general pretraining data sets consisting of short text, which do not provide high-quality examples for tasks requiring long contexts or frequent knowledge updates. Retrieval-augmented generation (RAG) addresses this limitation by incorporating an information retrieval step to enhance the generation process. In a typical RAG workflow, a retriever identifies relevant data sources for a given request, and the retrieved information is integrated with the input, leading to higher accuracy and better robustness. Surveys (Gao et al. 2023, Zhao et al. 2024) include details on RAG.

LLMs can orchestrate external application programming interface calls to fetch up-to-date information or perform computations, and agent-based approaches enable LLMs to autonomously plan and execute tasks across

Figure 8. Illustration of an Augmented-LLM Request


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

Notes. The API fetches detailed information about the 2024 Nobel Prize. CS, computer science.

various specialized modules. As these compound systems become more prevalent, improving their efficiency and adaptability becomes critical. This section presents the scheduling problem within such systems.

5.1. Augmented LLMs

Augmented language models (Mialon et al. 2023, Wang et al. 2024a) enhance traditional LLM capabilities by incorporating external tools or retrieval mechanisms. Unlike pure LLMs that rely solely on pretrained parameters to generate responses, augmented LLMs can query external data sources to expand their functionality. Figure 8 illustrates an example of an augmented LLM request. These augmentations, here referred to as *API*, fall into three main categories as described in Mialon et al. (2023): incorporating non-LLM tools during decoding (such as calculators (Wolfram 2024) and information retrieval systems (Baeza-Yates and Ribeiro-Neto 1999)); iterative self-calling of an LLM (like chatbots maintaining conversation history); and complex compositions involving multiple LLMs, models, and tools (exemplified by frameworks like LangChain (LangChain Team 2024), DSPy (Khattab et al. 2024), Gorilla (Patil et al. 2023), SGLang (Zheng et al. 2023), and AgentGraph (Chen et al. 2019)).

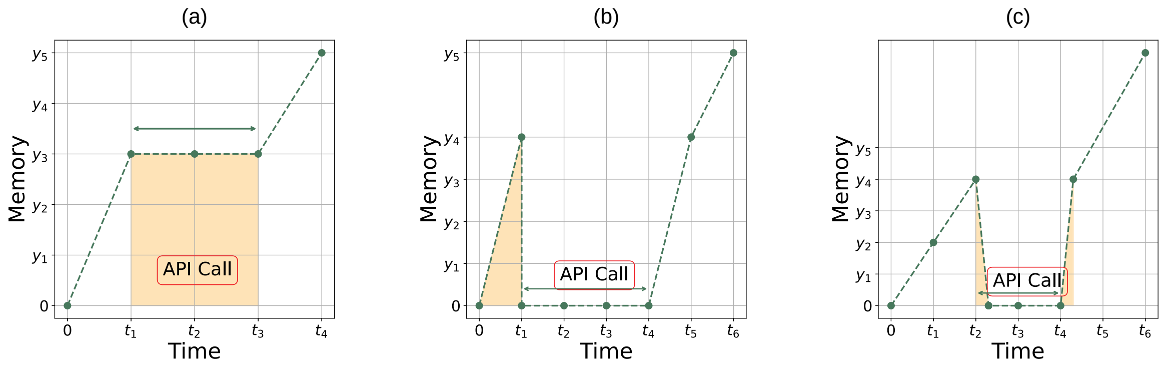
API call durations vary significantly between augmentation types, distinguishing short-running augmentations from long-running augmentations. Thus, API handling strategies should be tailored to the augmentation type rather than adopting a one-size-fits-all approach. During an API call, there are three primary strategies for handling the request.

- *Preserve*. Retain the KV cache in memory while waiting for the API response.
- *Discard and recompute*. Remove the KV cache, and rebuild it once the API returns.
- *Swap*. Offload the KV cache to CPU memory, and reload it when the API returns.

Each strategy has drawbacks. With *preserve*, the KV cache occupies memory even when the LLM is idle. The strategy *discard and recompute* wastes both memory and compute resources during cache reconstruction. *Swap* incurs data-transfer overhead and pauses request processing while transferring the KV cache. Figure 9 illustrates the memory-over-time function, with the highlighted areas in panels (a)–(c) of Figure 9 indicating the memory waste for a single request because of an API call.

API calls can range from milliseconds for simple calculations to several seconds for complex tasks, like image generation. API-augmented requests challenge the system by increasing KV cache memory demands during the memory-bound LLM decoding phase. INFERCEPT (Abhyankar et al. 2024) proposes a method to determine the

Figure 9. Memory Consumption over Time for a Request with One API Call Using Three Memory Management Strategies: (1) Preserve, (2) Discard and Recompute, and (3) Swap



Notes. The highlighted areas represent memory waste for one request. (a) Preserve. (b) Discard. (c) Swap.

handling strategy when a request reaches an API call; however, it lacks integrated scheduling policies to proactively minimize latency and relies on an FIFO approach, which may lead to head-of-line blocking.

Size-based scheduling methods can typically reduce request response times by utilizing known or predicted request sizes. Traditional scheduling, however, faces challenges with API-augmented requests because their memory requirements do not scale proportionally with execution time. In this context, it is unclear whether the API delay should be included in the size estimate. Even with known output sizes, techniques, such as shortest job first, may fail to perform effectively when requests involve API calls.

The work by Shahout et al. (2025b) is the first to propose scheduling policies beyond FIFO for augmented LLMs, presenting a system called LLM API- and Memory-based Predictive Scheduling (LMAPS). LAMPS employs a predictive, memory-aware scheduling approach that integrates API handling with request prioritization to reduce completion times. It operates in two steps. First, it assigns a handling strategy to API-augmented requests before scheduling based on predictions for expected output size and API call duration. Then, it schedules requests by ranking them according to their predicted total memory footprint across their life cycle, factoring in both request size and API interactions. LAMPS achieves end-to-end latency improvements of 27%–85% and reductions in TTFT of 4%–96% compared with INFERCEPT, with even greater gains over vLLM, an LLM serving system that applies paged attention to reduce memory overhead.

5.1.1. Open Questions

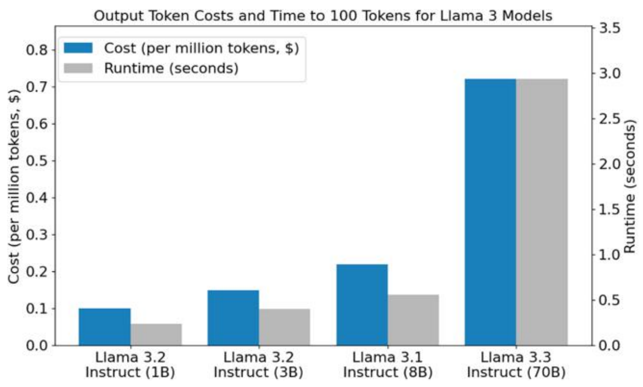
- What theoretical guarantees can scheduling algorithms offer for API-augmented requests? Few theoretical models address scheduling for the types of API calls examined here. Investigating algorithmic bounds, such as competitive ratios or approximation guarantees, for even simplified models may reveal new insights and guide the development of practical scheduling solutions.
- How can load balancing be implemented to route LLM inference requests among machines hosting API end points while considering real-time load with possible quality trade-offs?

5.2. Multiple LLMs Available

AI development is moving toward compound AI systems (Zaharia et al. 2024) that integrate multiple components, often including LLMs of varying sizes and capabilities. Model size significantly impacts run time, cost, and answer quality in transformer-based LLMs as shown in Figure 10. Smaller models may suffice for straightforward queries, whereas larger models provide deeper answers to complex prompts.

Speculative decoding (Chen et al. 2023a, Leviathan et al. 2023), also known as speculative sampling, employs two LLMs of different sizes (small and large) simultaneously to accelerate token generation while maintaining accuracy. This approach can reduce inference time by 2.5–3 times. The key insight is that the transformer architecture allows a single forward pass to generate one token or evaluate multiple tokens in parallel. Consequently, the computational cost of generating one token is equivalent to that of verifying an entire sequence of tokens concurrently. In this method, the smaller LLM generates preliminary tokens that the larger LLM verifies to determine which tokens to accept. When a token is rejected based on a comparison of the probability distributions from both models, the next token is sampled from the larger model. This process preserves overall accuracy (compared with using only the large model) through the speculative approach. An interpretation of this process is that the tokens generated by the smaller model are preliminary predictions that the larger model subsequently

Figure 10. Comparison of Output Token Costs (per Million Tokens) for the Llama 3 Family on Amazon Bedrock and the Time (in Seconds) Required to Generate 100 Tokens Measured on Two NVIDIA H100 GPUs



verifies and if necessary, corrects. This can be viewed through the analogy of a tandem model, where performance depends on the quality of the initial prediction. The time spent in the "first queue" (the small model) directly impacts the workload of the "second queue" (the large model). In other words, the accuracy of the small model's predictions determines the additional processing required by the larger model.

Routing all requests to a single large model can incur high costs, load, and latency, which degrade user experience and limit real-time applications. For example, on two NVIDIA H100 GPUs, Llama 3.2 Instruct (1B) generates 100 tokens in 0.2 seconds, whereas Llama 3.3 Instruct (70B) requires 3 seconds, reflecting greater computational complexity. Similarly, cost differences are significant; Amazon Bedrock charges $0.10 per million tokens for Llama 3.2 Instruct (1B) compared with $0.72 for Llama 3.3 Instruct (70B).[4] These observations motivate the need for advanced load-balancing and scheduling strategies in compound AI systems.

A key challenge is identifying the appropriate LLM to query based on the desired answer quality. Previous work (Ong et al. 2024) proposes four methods for predicting response quality in a two-model setting: similarity-weighted matching to training set labels, matrix factorization, a BERT-based classifier, or a call to Llama 3.1 Instruct (8B). These predictors are trained on human preference data augmented with LLM judge-labeled data sets.

5.2.1. Open Questions

- Can we model the speculative decoding process as a tandem queue and use it to gain insights into how to optimize the use of the smaller model?
- How can scheduling algorithms dynamically assign incoming requests to an appropriate LLM based on predicted response size and quality while meeting user-specified cost and latency constraints?
- Given that observed performance may differ from predictions under unpredictable workloads, how can scheduling algorithms adaptively balance predictions with observed performance (real-time feedback) under unpredictable workloads?
- Can queueing models give insight into optimizing the trade-offs between accuracy of results and the time spent in the system for LLM systems?

5.3. Multiple LLMs Needed

In the previous section, we discussed scenarios involving a single LLM selected from a pool of different LLM sizes to serve a request. Many compound AI systems, however, require sequential processing by multiple LLMs. We can view this setting from two primary perspectives. In the first, each LLM is responsible for a specific subtask, with a central coordinator aggregating the agents' outputs to form a final response. In the second, the processing follows a directed acyclic graph (DAG) of subjobs, where each stage processes and refines the previous stage's output, iteratively developing the response.

Both approaches present significant research challenges. Although aiming to optimize latency and reduce overhead, they differ fundamentally in task dependencies and scheduling complexities. In the agent-based model, LLMs operate as independent agents processing assigned subtasks. The scheduling challenge here centers on allocating incoming requests across heterogeneous GPU resources to maximize parallelism and throughput while minimizing coordination overhead during output aggregation. Alternatively, the DAG-based approach structures processing as a sequence of interdependent stages, with each stage's output serving as the subsequent stage's input. This model demands scheduling strategies that carefully manage stage dependencies, balancing pipeline parallelism with the synchronization needed for smooth interstage transitions. The objective is to minimize end-to-end latency despite variability in execution times and communication overhead. Although both settings share the overarching goals of reducing latency and overhead, they differ in fundamental approaches; the agent-based model emphasizes the parallel distribution of independent tasks, whereas the DAG-based model concentrates on orchestrating a chain of dependent operations.

5.3.1. Open Questions

- How can we design scheduling algorithms for multi-LLM systems both in the setting of independent, parallel tasks and in the setting of processing sequential, interdependent tasks? How can predictions aid scheduling decisions in these types of systems?
- What theoretical models from queueing theory and job-shop scheduling can be extended to provide either performance guarantees for (possibly simplified models of) multistage LLM inference systems or insights for scheduling approaches for such systems?

6. Scheduling in LLM Reasoning Systems

LLMs can now perform advanced reasoning tasks, such as solving mathematical problems, generating code, and analyzing legal documents. Inference-time reasoning algorithms play a key role by allowing LLMs to evaluate their outputs, explore alternative reasoning paths, and produce more reliable responses to complex questions. LLM reasoning algorithms have two fundamental phases: *expansion*, in which the model generates tokens to explore different solution paths where allocating more compute to expansion improves answer quality, and *aggregation*, where these candidate solutions are combined to produce a final answer.

LLM reasoning can be approached using the following distinct approaches. One popular approach is the majority (Wang et al. 2022) (or self-consistency), where a prompt is processed multiple times with different random seeds or temperature settings. The final output is determined by majority voting across these candidate responses, which enhances robustness by mitigating errors from any single inference run. The rebase (Wu et al. 2024) approach generates multiple intermediate reasoning steps from a given prompt. A reward model scores these steps, and the highest-scoring nodes are selected to guide further expansion. This iterative process constructs a reasoning tree, resulting in a final answer obtained either through weighted majority voting or by selecting the top-scored candidate. Another approach is Monte Carlo tree search (Feng et al. 2023, Hao et al. 2023), which builds a solution tree by iteratively expanding nodes. At each step, a continuation is sampled from the LLM until a leaf node with a candidate solution is reached; the candidate's score is then back propagated to update its ancestors, enabling effective exploration of both depth and breadth in the solution space. Finally, the internalized chain of thought leverages modern LLMs (OpenAI 2024, Qwen Team 2024, Guo et al. 2025) that have been trained to generate extended chain-of-thought (Wei et al. 2022) sequences in a single pass. These models internally refine their outputs by incorporating intermediate reasoning steps until reaching a final answer, eliminating the need for explicit multirun aggregation or tree search.

Prior works on serving systems (Kwon et al. 2023, Agrawal et al. 2024, Fu et al. 2024a, Patel et al. 2024, Shahout et al. 2025a) assume independent input/output requests and have extensively optimized LLM inference at the system level. These systems, however, overlook LLM reasoning programs that may submit interdependent inference requests. Parrot (Lin et al. 2024) and SGLang (Zheng et al. 2025) both target multirequest applications. Parrot offers an abstraction that allows users to specify dependencies between requests, enabling more effective crossrequest scheduling. SGLang introduces programming primitives tailored for multirequest workflows, optimizing execution by reusing intermediate KV cache memory across requests. However, these systems do not specifically address LLM reasoning programs. A recent work, Dynasor (Fu et al. 2024b), a reasoning-aware, end-to-end serving system for adaptive scheduling, efficiently targets reasoning systems by optimizing inference-time computing for LLM reasoning queries. It allocates additional compute resources to challenging queries, reduces compute for simpler ones, and terminates unpromising queries early. This balances accuracy, latency, and cost.

Reasoning systems demand adaptive scheduling that responds dynamically to the evolving state of the reasoning process. Although some queries converge after only a few steps, others require extensive exploration, leading to execution times that conventional scheduling algorithms are not designed to handle. In this context, the notion of request size goes beyond the number of output tokens to include the number of reasoning steps needed for convergence. This maps onto the job-size prediction framework of Section 2. The scheduler ranks each query by combining the number of reasoning chains (complexity) and the expected token length per chain. Complexity can be predicted with a one-bit prediction (easy/hard) or a multiclass classifier (e.g., easy, moderate, complex), whereas token count can be estimated via regression or by classifying into a small number of groups, each group corresponding to a range of tokens.

LLM reasoning systems thus provide a test bed for adaptive scheduling algorithms. A key research question is how to integrate complexity and size predictions into a unified ranking that guides scheduling decisions. Another direction is to refine predictions at run time as intermediate outputs become available, allowing the scheduler to update its ranking dynamically. Regarding resource allocation, when early reasoning outputs indicate a promising path, the scheduler can allocate additional compute to that query; when they signal divergence, it can terminate the task early to save resources and reassign them elsewhere. Parallel exploration of multiple reasoning branches can boost throughput but requires strategies for load balancing, avoiding redundant computation, and coordinating resource sharing.

6.1. Open Questions

- Because current reasoning algorithms share the core phases of expansion and aggregation, can we design a unified scheduling framework that dynamically allocates resources based on the current phase? In particular, how should scheduling priorities and resource management differ between the expansion phase and the aggregation phase?

• How can we develop a reasoning system that leverages KV cache sharing among different reasoning paths within a single request?

• How can prerun predictions of reasoning complexity (depth) be used to inform scheduling decisions, and how should we mitigate prediction errors?

7. Conclusion

Advances in algorithms with predictions have demonstrated the benefits of integrating machine learning with classical algorithms across various domains. Queueing systems are one such area, where recent works explore how predictions of service times can optimize scheduling. However, key questions remain regarding the limitations of existing theoretical frameworks and the robustness of queueing models under different predictive assumptions.

Interestingly, scheduling algorithms using predictions have already been shown to have significant potential for improving performance of LLM systems, particularly for inference. However, LLMs introduce particular scheduling challenges because of their memory-intensive inference processes, the need for dynamic batching, and the impact of preemption on KV cache management. Unlike traditional queueing systems, LLM systems must also account for factors such as cost and answer quality, and they involve multiple processing phases with distinct resource requirements that standard models do not capture. The growing complexity of AI deployments has further led to the emergence of LLM systems that extend beyond single-instance setups. These include compound AI platforms that integrate multiple LLMs with external tools as well as reasoning systems. Each of these architectures introduces additional challenges that provide new questions for queueing theory to consider.

Our goal in this paper is to highlight key challenges and open questions in algorithms with prediction in queueing in general and to describe particular new problems that arise in the context of LLMs that could potentially benefit from theoretical and algorithmic insights. Specifically, we highlight the need for new theoretical models that accommodate the characteristics of LLM inference. Addressing these challenges requires rethinking scheduling algorithms to better adapt to the growing complexity of modern AI systems.

Endnotes

[1] We note that this survey is written in conjunction with a tutorial that the authors will give at ACM SIGMETRICS 2025 and acknowledge that the survey includes discussion of work conducted by the authors alongside other contributions.

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

[3] We will use terms like job size, size, service time, and time interchangeably where the meaning is clear.

[4] Amazon Bedrock pricing is for the `us-east-1` region as of February 2025.

References

Abhyankar R, He Z, Srivatsa V, Zhang H, Zhang Y (2024) InferCept: Efficient intercept support for augmented large language model inference. Salakhutdinov R, Kolter Z, Heller K, Weller A, Oliver N, Scarlett J, Berkenkamp F, eds. *Proc. 41st Internat. Conf. Machine Learn. (ICML 2024)*, vol. 238 (PMLR, New York), 8056–8082.

Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, Aleman FL, Almeida D, et al. (2023) GPT-4 technical report. Preprint, submitted March 15, https://arxiv.org/abs/2303.08774.

Agrawal A, Kedia N, Panwar A, Mohan J, Kwatra N, Gulavani BS, Tumanov A, Ramjee R (2024) Taming throughput-latency tradeoff in LLM inference with Sarathi-serve. Preprint, submitted March 4, https://arxiv.org/abs/2403.02310.

Akbari-Moghaddam M, Down DG (2023) SEH: Size estimate hedging scheduling of queues. *ACM Trans. Model. Comput. Simulation* 33(4):14.

Algorithms with Predictions Project (2024) Algorithms with predictions: Paper list. Accessed July 2, 2025, https://algorithms-with-predictions.github.io.

Anthropic (2024) Prompt caching with Claude. Accessed July 2, 2025, https://www.anthropic.com/news/prompt-caching.

Azar Y, Leonardi S, Touitou N (2021) Flow time scheduling with uncertain processing time. Khuller S, Vassilievska Williams V, eds. *Proc. 53rd Annual ACM SIGACT Sympos. Theory Comput. (STOC)* (ACM, New York), 1070–1080.

Azar Y, Leonardi S, Touitou N (2022) Distortion-oblivious algorithms for minimizing flow time. Naor J (Seffi), Buchbinder N, eds. *Proc. 2022 ACM-SIAM Sympos. Discrete Algorithms (SODA)* (SIAM, Philadelphia), 252–274.

Baeza-Yates R, Ribeiro-Neto B (1999) *Modern Information Retrieval*, vol. 463 (ACM Press, New York).

Belinkov Y (2022) Probing classifiers: Promises, shortcomings, and advances. *Comput. Linguistics* 48(1):207–219.

Besta M, Blach N, Kubicek A, Gerstenberger R, Podstawski M, Gianinazzi L, Gajda J, et al. (2024) Graph of thoughts: Solving elaborate problems with large language models. Wooldridge M, Dy J, Natarajan S, eds. *Proc. Thirty Eighth AAAI Conf. Artificial Intelligence (AAAI-24)*, vol. 38(16) (AAAI Press, Palo Alto, CA), 17682–17690.

Blum A, Srinivas V (2025) Competitive strategies to use "warm start" algorithms with predictions. Azar Y, Panigrahi D, eds. *Proc. 2025 Annual ACM-SIAM Sympos. Discrete Algorithms (SODA)* (SIAM, Philadelphia), 3775–3801.

Boyar J, Favrholdt LM, Kudahl C, Larsen KS, Mikkelsen JW (2017) Online algorithms with advice: A survey. *ACM Comput. Surveys* 50(2): 93–129.

Brand J, Forster S, Nazari Y, Polak A (2024) On dynamic graph algorithms with predictions. Woodruff DP, ed. *Proc. 2024 Annual ACM-SIAM Sympos. Discrete Algorithms (SODA 2024)* (SIAM, Philadelphia), 3534–3557.

Brown TB, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, et al. (2020) Language models are few-shot learners. *Advances in Neural Information Processing Systems*, vol. 33 (Curran Associates, Inc., Red Hook, NY), 1877–1901.

Burke PJ (1956) The output of a queuing system. *Oper. Res.* 4(6):699–704.

Charlet N, Van Houdt B (2024) Tail optimality of the nudge-M scheduling algorithm. *Sigmetrics Performance Evaluation Rev.* 52(2):21–23.

Chen Y, Dong J (2021) Scheduling with service-time information: The power of two priority classes. Preprint, submitted February 16, https://arxiv.org/abs/2105.10499.

Chen JCY, Saha S, Bansal M (2023b) ReConcile: Round-table conference improves reasoning via consensus among diverse LLMs. Preprint, submitted September 22, https://arxiv.org/abs/2309.13007.

Chen C, Borgeaud S, Irving G, Lespiau JB, Sifre L, Jumper J (2023a) Accelerating large language model decoding with speculative sampling. Preprint, submitted February 3, https://arxiv.org/abs/2302.01318.

Chen L, Chen Z, Tan B, Long S, Gašić M, Yu K (2019) Agentgraph: Toward universal dialogue management with structured deep reinforcement learning. *IEEE/ACM Trans. Audio Speech Language Processing* 27(9):1378–1391.

Chen M, Tworek J, Jun H, Yuan Q, Pinto HPDO, Kaplan J, Edwards H, et al. (2021) Evaluating large language models trained on code. Preprint, submitted July 7, https://arxiv.org/abs/2107.03374.

Cheng K, Hu W, Wang Z, Du P, Li J, Zhang S (2024) Enabling efficient batch serving for LMaaA via generation length prediction. Preprint, submitted June 7, https://arxiv.org/abs/2406.04785.

Dao T (2023) Flashattention-2: Faster attention with better parallelism and work partitioning. Preprint, submitted July 17, https://arxiv.org/abs/2307.08691.

Dao T, Haziza D, Massa F, Sizov G (2023) Flash-Decoding for long-context inference. Accessed July 2, 2025, https://pytorch.org/blog/flash-decoding/.

Dao T, Fu D, Ermon S, Rudra A, Ré C (2022) Flashattention: Fast and memory-efficient exact attention with IO-awareness. *Adv. Neural Inform. Processing Systems* 35:16344–16359.

Dell'Amico M (2019) Scheduling with inexact job sizes: The merits of shortest processing time first. Preprint, submitted July 10, https://arxiv.org/abs/1907.04824.

Dell'Amico M, Carra D, Michiardi P (2015) PSBS: Practical size-based scheduling. *IEEE Trans. Comput.* 65(7):2199–2212.

Dell'Amico M, Carra D, Pastorelli M, Michiardi P (2014) Revisiting size-based scheduling with estimated job sizes. *Proc. 2014 IEEE 22nd Internat. Sympos. Model. Anal. Simulation Comput. Telecomm. Systems* (IEEE Computer Society, Washington, DC), 411–420.

Dinitz M, Im S, Lavastida T, Moseley B, Niaparast A, Vassilvitskii S (2024) Binary search with distributional predictions. Preprint, submitted November 25, https://arxiv.org/abs/2411.16030.

Dong J, Ibrahim R (2024) Shortest-job-first scheduling in many-server queues with impatient customers and noisy service-time estimates. *Oper. Res.*, ePub ahead of print December 14, https://doi.org/10.1287/opre.2022.310.

Dong Q, Li L, Dai D, Zheng C, Ma J, Li R, Xia H, et al. (2022) A survey on in-context learning. Preprint, submitted December 31, https://arxiv.org/abs/2301.00234.

Faiz A, Kaneda S, Wang R, Osi R, Sharma P, Chen F, Jiang L (2023) LLMcarbon: Modeling the end-to-end carbon footprint of large language models. Preprint, submitted September 25, https://arxiv.org/abs/2309.14393.

Feng X, Wan Z, Wen M, McAleer SM, Wen Y, Zhang W, Wang J (2023) Alphazero-like tree-search can guide large language model decoding and training. Preprint, submitted September 29, https://arxiv.org/abs/2309.17179.

Fu Y, Chen J, Zhu S, Fu Z, Dai Z, Qiao A, Zhang H (2024a) Efficient LLM scheduling by learning to rank. Preprint, submitted August 28, https://arxiv.org/abs/2408.15792.

Fu Y, Chen J, Zhu S, Fu Z, Dai Z, Qiao A, Zhang H (2024b) Efficiently serving LLM reasoning programs with certaindex. Preprint, submitted December 30, https://arxiv.org/html/2412.20993v1.

Gao S, Fang A, Huang Y, Giunchiglia V, Noori A, Schwarz JR, Ektefaie Y, Kondic J, Zitnik M (2024) Empowering biomedical discovery with AI agents. *Cell* 187(22):6125–6151.

Gao Y, Xiong Y, Gao X, Jia K, Pan J, Bi Y, Dai Y, Sun J, Wang M, Wang H (2023) Retrieval-augmented generation for large language models: A survey. Preprint, submitted December 18, https://arxiv.org/abs/2312.10997.

Grosof I (2024) Optimal scheduling in multiserver queues. *ACM SIGMETRICS Performance Evaluation Rev.* 51(3):29–32.

Grosof I, Mitzenmacher M (2022) Incentive compatible queues without money. Preprint, submitted February 11, https://arxiv.org/abs/2202.05747.

Guo T, Chen X, Wang Y, Chang R, Pei S, Chawla NV, Wiest O, Zhang X (2024) Large language model based multi-agents: A survey of progress and challenges. Preprint, submitted January 21, https://arxiv.org/abs/2402.01680.

Guo D, Yang D, Zhang H, Song J, Zhang R, Xu R, Zhu Q, et al. (2025) Deepseek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. Preprint, submitted January 22, https://arxiv.org/abs/2501.12948.

Hao S, Gu Y, Ma H, Hong JJ, Wang Z, Wang DZ, Hu Z (2023) Reasoning with language model is planning with world model. Preprint, submitted May 24, https://arxiv.org/abs/2305.14992.

Harchol-Balter M (2013) *Performance Modeling and Design of Computer Systems: Queueing Theory in Action* (Cambridge University Press, Cambridge, UK).

Harlev A, Yu G, Scully Z (2024) A Gittins policy for optimizing tail latency. *ACM SIGMETRICS Performance Evaluation Rev.* 52(2):15–17.

Hewitt J, Liang P (2019) Designing and interpreting probes with control tasks. Preprint, submitted September 8, https://arxiv.org/abs/1909.03368.

Hewitt J, Manning CD (2019) A structural probe for finding syntax in word representations. *Proc. North Amer. Chapter Assoc. Comput. Linguistics Human Language Tech.* (Association for Computational Linguistics, Stroudsburg, PA).

Jaiswal S, Jain K, Simmhan Y, Parayil A, Mallick A, Wang R, Amant RS, et al. (2025) Serving models, fast and slow: Optimizing heterogeneous LLM inferencing workloads at scale. Preprint, submitted February 20, https://arxiv.org/abs/2502.14617.

Jin Y, Wu CF, Brooks D, Wei GY (2023) S^3: Increasing GPU utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, vol. 36 (Curran Associates Inc., Red Hook, NY), 18015–18027.

Jo A (2023) The promise and peril of generative AI. *Nature* 614(7947):214–216.

Juravsky J, Brown B, Ehrlich R, Fu DY, Ré C, Mirhoseini A (2024) Hydragen: High-throughput LLM inference with shared prefixes. Preprint, submitted February 7, https://arxiv.org/abs/2402.05099.

Khattab O, Singhvi A, Maheshwari P, Zhang Z, Santhanam K, Haq S, Sharma A, et al. (2024) DSPY: Compiling declarative language model calls into state-of-the-art pipelines. *Proc. Twelfth Internat. Conf. Learn. Representations (ICLR 2024)* (ICLR, Appleton, WI).

Kwon W, Li Z, Zhuang S, Sheng Y, Zheng L, Yu CH, Gonzalez J, Zhang H, Stoica I (2023) Efficient memory management for large language model serving with pagedattention. *Proc. 29th Sympos. Operating Systems Principles* (ACM, New York), 611–626.

LangChain Team (2024) LangChain. Accessed June 16, 2025, https://github.com/langchain-ai/langchain.

Leviathan Y, Kalman M, Matias Y (2023) Fast inference from transformers via speculative decoding. Krause A, Brunskill E, Cho K, Engelhardt B, Sabato S, Scarlett J, eds. *Proc. 40th Internat. Conf. Machine Learn. (ICML 2023)*, vol. 202 (PMLR, New York), 19274–19286.

Li J, Wang M, Zheng Z, Zhang M (2024) LooGLE: Can long-context language models understand long contexts? Preprint, submitted November 8, https://arxiv.org/abs/2311.04939.

Li Y, Choi D, Chung J, Kushman N, Schrittwieser J, Leblond R, Eccles T, et al. (2022) Competition-level code generation with AlphaCode. Preprint, submitted February 8, https://arxiv.org/abs/2203.07814.

Lin C, Han Z, Zhang C, Yang Y, Yang F, Chen C, Qiu L (2024) Parrot: Efficient serving of LLM-based applications with semantic variable. *Proc. 18th USENIX Sympos. Operating Systems Design Implementation (OSDI '24)* (USENIX Association, Santa Clara, CA), 929–945.

Liu A, Feng B, Xue B, Wang B, Wu B, Lu C, Zhao C, et al. (2024) DeepSeek-V3 technical report. Preprint, submitted December 27, https://arxiv.org/abs/2412.19437.

Lykouris T, Vassilvitskii S (2021) Competitive caching with machine learned advice. *J. ACM* 68(4):24.

Mailach R, Down DG (2017) Scheduling jobs with estimation errors for multi-server systems. *Proc. 29th Internat. Teletraffic Congress (ITC 29)*, vol. 1 (IEEE, Piscataway, NJ), 10–18.

Mialon G, Dessì R, Lomeli M, Nalmpantis C, Pasunuru R, Raileanu R, Rozière B, et al. (2023) Augmented language models: A survey. Preprint, submitted February 15, https://arxiv.org/abs/2302.07842.

Mitzenmacher M (2001) The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distributed Systems* 12(10):1094–1104.

Mitzenmacher M (2020) Scheduling with predictions and the price of misprediction. Vidick T, ed. *Proc. 11th Innovations Theoret. Comput. Sci. Conf. (ITCS 2020)*, LIPIcs, vol. 151 (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, Germany), 14:1–14:18.

Mitzenmacher M (2021) Queues with small advice. Bender MA, Gilbert J, Hendrickson B, Sullivan BD, eds. *Proc. 2021 SIAM Conf. Appl. Comput. Discrete Algorithms (ACDA 2021)* (SIAM, Philadelphia), 1–12.

Mitzenmacher M, Dell'Amico M (2022) The supermarket model with known and predicted service times. *IEEE Trans. Parallel Distributed Systems* 33(11):2740–2751.

Mitzenmacher M, Vassilvitskii S (2020) Algorithms with predictions. Roughgarden T, ed. *Beyond the Worst-Case Analysis of Algorithms* (Cambridge University Press, Cambridge, UK), 646–662.

Mitzenmacher M, Vassilvitskii S (2022) Algorithms with predictions. *Comm. ACM* 65(7):33–35.

Moseley B, Newman H, Pruhs K, Zhou R (2025) Robust Gittins for stochastic scheduling. Preprint, submitted April 14, https://arxiv.org/abs/2504.10743.

NVIDIA (2024) FasterTransformer. Accessed July 2, 2025, https://github.com/NVIDIA/FasterTransformer.

Ong I, Almahairi A, Wu V, Chiang WL, Wu T, Gonzalez JE, Kadous MW, Stoica I (2024) RouteLLM: Learning to route LLMs with preference data. Preprint, submitted June 26, https://arxiv.org/abs/2406.18665.

OpenAI (2024) Learning to reason with LLMs. Accessed July 2, 2025, https://openai.com/index/learning-to-reason-with-llms/.

Patel P, Choukse E, Zhang C, Shah A, Goiri Í, Maleki S, Bianchini R (2024) Splitwise: Efficient generative LLM inference using phase splitting. *Proc. 51st Annual ACM/IEEE 51st Internat. Sympos. Comput. Architecture (ISCA 2024)* (IEEE, Piscataway, NJ), 3775–3801.

Patil SG, Zhang T, Wang X, Gonzalez JE (2023) Gorilla: Large language model connected with massive APIs. Preprint, submitted May 24, https://arxiv.org/abs/2305.15334.

Pope R, Douglas S, Chowdhery A, Devlin J, Bradbury J, Heek J, Xiao K, Agrawal S, Dean J (2023) Efficiently scaling transformer inference. *Proc. Machine Learn. Systems* 5:606–624.

Qin Y, Liang S, Ye Y, Zhu K, Yan L, Lu Y, Lin Y, et al. (2023) ToolLLM: Facilitating large language models to master 16000+ real-world APIs. Preprint, submitted July 31, https://arxiv.org/abs/2307.16789.

Qiu H, Mao W, Patke A, Cui S, Jha S, Wang C, Franke H, Kalbarczyk Z, Başar T, Iyer RK (2024a) Power-aware deep learning model serving with μ-serve. *Proc. 2024 USENIX Annual Tech. Conf. (USENIX ATC '24)* (USENIX Association, Santa Clara, CA), 75–93.

Qiu H, Mao W, Patke A, Cui S, Jha S, Wang C, Franke H, Kalbarczyk ZT, Başar T, Iyer RK (2024b) Efficient interactive LLM serving with proxy model-based sequence length prediction. Preprint, submitted April 12, https://arxiv.org/abs/2404.08509.

Qwen Team (2024) QwQ: Reflect deeply on the boundaries of the unknown. Accessed July 2, 2025, https://qwenlm.github.io/blog/qwq-32b-preview/.

Radford A, Wu J, Child R, Luan D, Amodei D, Sutskever I (2019) Language models are unsupervised multitask learners. *OpenAI Blog* 1(8):1–24.

Ramesh A, Pavlov M, Goh G, Gray S, Voss C, Radford A, Chen M, Sutskever I (2021) Zero-shot text-to-image generation. Meilă M, Zhang T, eds. *Proc. 38th Internat. Conf. Machine Learn. (ICML 2021)*, vol. 139 (PMLR, New York), 8821–8831.

Rawal R, Saifullah K, Farré M, Basri R, Jacobs D, Somepalli G, Goldstein T (2024) Cinepile: A long video question answering dataset and benchmark. Preprint, submitted May 14, https://arxiv.org/abs/2405.08813.

Reich E (1957) Waiting times when queues are in tandem. *Ann. Math. Statist.* 28(3):768–773.

Salman SM, Papadopoulos AV, Mubeen S, Nolte T (2023a) Evaluating dispatching and scheduling strategies for firm real-time jobs in edge computing. *Proc. 49th Annual Conf. IEEE Indust. Electronics Soc. (IECON 2023)* (IEEE, Piscataway, NJ), 1–6.

Salman SM, Dao VL, Papadopoulos AV, Mubeen S, Nolte T (2023b) Scheduling firm real-time applications on the edge with single-bit execution time prediction. *Proc. 2023 IEEE 25th Internat. Sympos. Real-Time Distributed Comput. (ISORC 2023)* (IEEE, Piscataway, NJ), 207–213.

Sanh V, Debut L, Chaumond J, Wolf T (2019) DistilBERT: A distilled version of BERT: Smaller, faster, cheaper and lighter. Preprint, submitted October 2, https://arxiv.org/abs/1910.01108.

Schick T, Dwivedi-Yu J, Dessi R, Raileanu R, Lomeli M, Hambro E, Zettlemoyer L, Cancedda N, Scialom T (2023) Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, vol. 36 (Curran Associates Inc., Red Hook, NY), 68539–68551.

Scully Z, Harchol-Balter M (2018) SOAP bubbles: Robust scheduling under adversarial noise. *Proc. 56th Annual Allerton Conf. Comm. Control Comput. (Allerton 2018)* (IEEE, Piscataway, NJ), 144–154.

Scully Z, Harchol-Balter M (2021) The Gittins policy in the M/G/1 queue. *Proc. 19th Internat. Sympos. Model. Optim. Mobile Ad Hoc Wireless Networks (WiOpt 2021)* (IEEE, Piscataway, NJ), 248–255.

Scully Z, Grosof I, Harchol-Balter M (2020a) The Gittins policy is nearly optimal in the M/G/k under extremely general conditions. *Proc. ACM Measurement Anal. Comput. Systems* 4(3):43.

Scully Z, Grosof I, Mitzenmacher M (2022) Uniform bounds for scheduling with job size estimates. Braverman M, ed. *13th Innovations Theoret. Comput. Sci. Conf. ITCS*, LIPIcs, vol. 215 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Wadern, Germany), 114:1–114:30.

Scully Z, Harchol-Balter M, Scheller-Wolf A (2018) SOAP: One clean analysis of all age-based scheduling policies. *Proc. ACM Measurement Anal. Comput. Systems* 2(1):16.

Scully Z, Van Kreveld L, Boxma O, Dorsman JP, Wierman A (2020b) Characterizing policies with optimal response time tails under heavy-tailed job sizes. *Proc. ACM Measurement Anal. Comput. Systems* 4(2):30.

Shahout R, Mitzenmacher M (2024) SkipPredict: When to invest in predictions for scheduling. Globerson A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak J, Zhang C, eds. *Advances in Neural Information Processing Systems*, vol. 37 (Curran Associates, Inc., Red Hook, NY).

Shahout R, Malach E, Liu C, Jiang W, Yu M, Mitzenmacher M (2025a) Don't stop me now: Embedding based scheduling for LLMs. *Internat. Conf. Learn. Representations* (ICLR, Appleton, WI).

Shahout R, Liang C, Xin S, Lao Q, Cui Y, Yu M, Mitzenmacher M (2025b) Fast inference for augmented large language models. Preprint, submitted October 25, https://arxiv.org/abs/2410.18248.

Srivatsa V, He Z, Abhyankar R, Li D, Zhang Y (2024) Preble: Efficient distributed prompt scheduling for LLM serving. Preprint, submitted May 8, https://arxiv.org/abs/2407.00023.

Stojkovic J, Zhang C, Goiri Í, Torrellas J, Choukse E (2024) DynamoLLM: Designing LLM inference clusters for performance and energy efficiency. Preprint, submitted August 1, https://arxiv.org/abs/2408.00741.

Taori R, Gulrajani I, Zhang T, Dubois Y, Li X, Guestrin C, Liang P, Hashimoto TB (2023) Stanford alpaca: An instruction-following LLaMA model. Accessed July 2, 2025, https://github.com/tatsu-lab/stanford_alpaca.

Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, et al. (2023) LLaMA: Open and efficient foundation language models. Preprint, submitted February 27, https://arxiv.org/abs/2302.13971.

Vvedenskaya ND, Suhov YM (1997) Dobrushin's mean-field approximation for a queue with dynamic routing. *Markov Processes Related Fields* 3(4):493–526.

Wang Y, Ma X, Chen W (2024b) Augmenting black-box LLMs with medical textbooks for biomedical question answering. *Findings 2024 Conf. Empirical Methods Natl. Language Processing (EMNLP 2024)* (Association for Computational Linguistics, Stroudsburg, PA), 1754–1770.

Wang X, Wei J, Schuurmans D, Le Q, Chi E, Narang S, Chowdhery A, Zhou D (2022) Self-consistency improves chain of thought reasoning in language models. Preprint, submitted March 21, https://arxiv.org/abs/2203.11171.

Wang L, Ma C, Feng X, Zhang Z, Yang H, Zhang J, Chen Z, et al. (2024a) A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18(6):186345.

Wei J, Wang X, Schuurmans D, Bosma M, Xia F, Chi E, Le QV, Zhou D, et al. (2022) Chain-of-thought prompting elicits reasoning in large language models. *Adv. Neural Inform. Processing Systems* 35:24824–24837.

Wierman A, Nuyens M (2008) Scheduling despite inexact job-size information. *Proc. 2008 ACM SIGMETRICS Internat. Conf. Measurement Model. Comput. Systems (SIGMETRICS '08)* (ACM, New York), 25–36.

Wolfram S (2024) ChatGPT gets its "Wolfram superpowers!" Accessed July 2, 2025, https://writings.stephenwolfram.com/2023/03/chatgpt-gets-its-wolfram-superpowers/.

Wu Y, Sun Z, Li S, Welleck S, Yang Y (2024) Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. Preprint, submitted August 1, https://arxiv.org/abs/2408.00724.

Wu B, Zhong Y, Zhang Z, Huang G, Liu X, Jin X (2023) Fast distributed inference serving for large language models. Preprint, submitted May 10, https://arxiv.org/abs/2305.05920.

Yu G, Scully Z (2024) Strongly tail-optimal scheduling in the light-tailed M/G/1. *Proc. ACM Measurement Anal. Comput. Systems* 8(2):27.

Yu GI, Jeong JS, Kim GW, Kim S, Chun BG (2022) Orca: A distributed serving system for transformer-based generative models. *Proc. 16th USENIX Sympos. Operating Systems Design Implementation (OSDI '22)* (USENIX Association, Berkeley, CA), 521–538.

Zaharia M, Khattab O, Chen L, Davis JQ, Miller H, Potts C, Zou J, et al. (2024) The shift from models to compound AI systems. Accessed July 2, 2025, https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems.

Zhao P, Zhang H, Yu Q, Wang Z, Geng Y, Fu F, Yang L, Zhang W, Jiang J, Cui B (2024) Retrieval-augmented generation for AI-generated content: A survey. Preprint, submitted February 29, https://arxiv.org/abs/2402.19473.

Zheng Z, Ren X, Xue F, Luo Y, Jiang X, You Y (2024) Response length perception and sequence scheduling: An LLM-empowered LLM inference pipeline. *Advances in Neural Information Processing Systems*, vol. 36 (Curran Associates Inc., Red Hook, NY), 65517–65530.

Zheng L, Yin L, Xie Z, Huang J, Sun C, Yu CH, Cao S, et al. (2023) Efficiently programming large language models using SGLang. Preprint, submitted December 12, https://arxiv.org/abs/2312.07104.

Zheng L, Yin L, Xie Z, Sun CL, Huang J, Yu CH, Cao S, et al. (2025) SGLang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, vol. 38 (Curran Associates Inc., Red Hook, NY), 62557–62583.

Zhong Y, Liu S, Chen J, Hu J, Zhu Y, Liu X, JX, Zhang H (2024) DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. Preprint, submitted January 18, https://arxiv.org/abs/2401.09670.