Algorithms with Predictions in Queueing: Challenges and Open Problems (Especially for LLMs)

Abstract

The research area of algorithms with predictions, also referred to as learning-augmented algorithms, considers the setting where an algorithm is given advice in the form of predictions, such as from a machine-learning model. For example, when scheduling jobs, one could obtain a prediction of each job's service time. Queueing systems present many opportunities for applying learning-augmented algorithms, raising numerous open questions about how predictions can be effectively leveraged to improve scheduling decisions. Several recent studies have started the exploration of queues with predicted service times instead of exact ones, typically aiming to minimize the average time a job spends in the system. We start by reviewing this recent work, highlighting the potential effectiveness of predictions and providing a collection of open questions regarding the performance of queues and queueing systems using predictions.

We then move to consider an important practical example, Large Language Model (LLM) systems, which presents novel scheduling challenges and highlights the need for learning-augmented algorithms to optimize performance. In particular, we consider LLMs performing inference, which refer to the model generating a response after receiving an input prompt, or request. Performance metrics in LLM systems have expanded beyond traditional measures to include multi-dimensional objectives such as runtime, latency, computational cost, and response quality. Inference requests (jobs) in LLM systems are inherently complex; they have variable inference times, dynamic memory footprints that are constrained by key-value (KV) 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.

1 Introduction

When considering 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 i.i.d service times – there are two common settings. First, it may be that no information about any job's service time is known, in which case First-In First-out (FIFO), also referred to as First-come First-Served (FCFS), is generally used. Second, it may be that every job's service time is known, in which case the Shortest Remaining Processing Time (SRPT) optimizes the expected response time.

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 [20, 21, 47, 48, 66, 82]. Recent work has also considered the setting of scheduling jobs with deadlines [61, 62], demonstrating that predictions can improve scheduling decisions.

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. While 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 [13] – 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 [1] on the subject and as discussed in the surveys [50, 51].

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: Large Language Model (LLM) systems. LLM systems have transformed many domains by enabling powerful 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 datasets, while inference, the focus of this work, is the real-time execution phase where pre-trained models generate responses to user requests. At inference, LLM models generate text token by token, 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 [75] and DeepSeek [43, 28], inference has become the most common mode of interaction with LLMs. Modeling LLM systems remains largely unexplored in the queueing community.

LLM systems present a wealth of new queueing and scheduling challenges that expand on problems from traditional queueing systems. While modeling LLM systems remains largely unexplored in the queueing community, we believe 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.

LLM systems 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. Also, LLM inference introduces challenges not present in standard queueing models, most notably the need for an ever-growing key-value (KV) cache. 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, each new token is conditioned on all previously generated tokens, so the cache accumulates more data as the response grows. However, its presence introduces memory constraints that complicate scheduling, as each request consumes GPU memory that cannot be freed until the request is completed. This issue becomes particularly problematic under high-load scenarios with limited GPU memory, especially when many inference requests are processed in parallel through batching to maximize GPU utilization.

Moreover, preemption, where a running job can be interrupted to prioritize a more urgent request, is non-trivial in LLM inference due to KV cache management. In LLM systems, preempting a request requires dealing with its allocated KV cache memory, which either must be kept in GPU memory (which uses space other jobs may need), deleted from memory (which requires recomputation), or transferred to the 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. In the prefill phase, the model processes all input tokens at once using matrix multiplication, making this phase compute-bound. The decode phase, in contrast, generates tokens sequentially in an autoregressive manner, and it is instead memory-bandwidth-bound and benefits from batching. The scheduler must efficiently balance these phases. Existing approaches explore prioritizing shorter requests, dynamically adjusting batch sizes, and using job size predictions, but fundamental trade-offs between latency, throughput, and memory efficiency remain open questions.

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 API calls for retrieval or computation, introducing new scheduling constraints. When an API call is made, the system must decide how to manage the KV cache: preserving it increases memory usage, discarding it leads to computation overhead, and swapping it incurs transfer delays. These decisions affect both latency and resource utilization. Another challenge arises when multiple LLMs of different sizes are available, requiring a routing strategy to balance cost, response time, and answer quality. Some compound AI systems involve multi-step pipelines, where requests pass through multiple LLMs in sequence, creating dependencies that influence scheduling.

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, while 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 first-in first-out (FIFO) scheduling when job sizes[1] are not known, and use shortest remaining processing time (SRPT) when job sizes are known. These scheduling policies are well understood, as are related policies that use known job sizes, such as shortest job first (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 are instead predicted.

Mitzenmacher [47] 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, for example, the expected response time for 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 [], 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 [47] 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 results in [47], shows the expected response time in equilibrium for SPJF and SPRPT and compares them to the expected response times for FIFO, SJF, and SRPT. The primary takeaway from this example is that while using predictions is naturally not as good as using exact information, it provides significant gains over not using FIFO, which does not use any information about job times.

Table 1: Results from equations (to four decimal places) for FIFO, Shortest Job First (SJF), Shortest Predicted Job First (SPJF), Preemptive Shortest Job First (SJF), Preemptive Shortest Predicted Job First (PSPJF) Shortest Remaining Processing Time (SRPT), and Shortest Predicted Remaining Processing Time (SPRPT), where the predicted time for a job with size x is exponentially distributed with mean x. Taken from [47].

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

While [47] derives equations for these prediction-based polices directly, following similar previous derivations for the standard variants without predictions (see, e.g. [31]), it should be noted that these particular prediction-based scheduling schemes are amenable to analysis using the SOAP methodology of [69]. (See also [67].) 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 it has been served. A job's type could correspond to a job class in a system with job classes, or 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 or time, it's rank is simply $y - a$ (the predicted remaining service time) as long as $a \leq x$

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

(since after being served for time x the job completes). The SOAP methodology provides a very general (albeit sometimes somewhat 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.

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 [68] 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., [22]). 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 1-bit predictions

Additional recent work continues to develop the use of predictions. Mitzenmacher [] studies the viability of using 1-bit predictions, where the bit corresponds to a prediction as to whether jobs have a size below or above some threshold T. The motivation for 1-bit prediction is that predicting jobs as being either short or long in this way is natural, likely to be less computationally expensive, and likely to be more accurate from a machine-learning standpoint. Additionally, the 1-bit prediction leads to a simple implementation: short jobs can be placed at the front of the queue ,while 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, 1-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, and 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:

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

and the response time with preemption is shown to be

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

where K_1 and K_2 are modified Bessel functions of the second kind (with different parameters, 1 and 2). In this particular setting, preemption is always helpful. It is perhaps surprising that such a model has a compact closed form, albeit in terms of the 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 [48], 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

Table 2: Table from [48]. 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

Table 3: Table from [48]. 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

distribution (with cumulative distribution function $F(x) = 1 - e^{-\sqrt{2x}}$). Perhaps not surprisingly, 1-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 1-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 [21] for discussion of this.) More generally, even with predictions of the actual service times, it is better to have good predictions for longer jobs.

Open Questions:

- What other natural models of prediction errors are there for 1-bit predictions?
- Can we formalize how to optimize the predictions we would like to obtain from machine-learned predictions, under some specific scheduling policy such as SPRPT or 1-bit predictions from thresholds?
- 1-bit prediction analysis can readily be generalized to k-bit prediction analysis (or from 2 classes to greater than 2 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.

2.3 Uniform Bounds

While 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, motivates 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., [1, 50]), and the two areas fit together quite naturally. Since 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 the ratio of the value of the solution obtained by the proposed online 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, 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 [44], we may say that an algorithm is α-consistent if its competitive ratio tends to α as the error in the predictions goes to 0, and β-robust if the competitive ratio is bounded by β even with arbitrarily bad predictions.

The work [66] 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 1, 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, 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.

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 0, 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, Grosof, and Harchol-Balter [65, 69]. 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, Leonardi, and Touitou 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 [8] 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 [9], 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 [66], without making stochastic assumptions on the job sizes, but with a more complex algorithm and a slightly larger-than-linear competitive ratio.

Open Questions:

- Can we tighten the various bounds of [66], in particular improving the constants C related to graceful degradation.

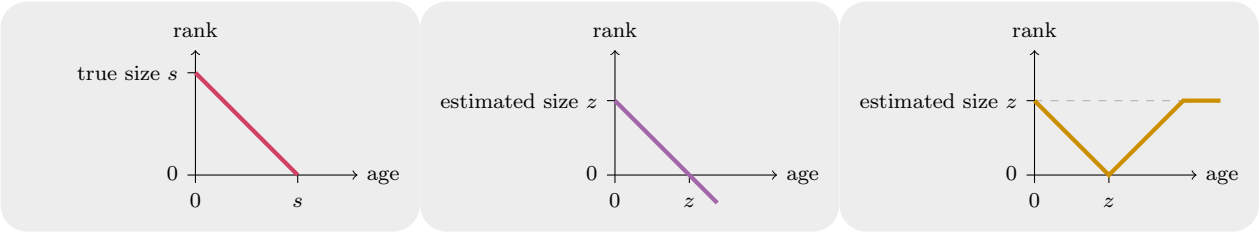


Figure 1: Rank functions of size-estimate-based policies. The rank function for SRPT is on the left; the rank decreases as $s - a$ where s is the true size and a is the age. The rank function for SPRPT is in the middle; it decreases as $z - a$ where z is now the estimated size. Note that a job can have negative rank, at which point it cannot be preempted. The SPRPT-with-bounce rank function from [66] is on the right; the rank decreases from the estimate z to 0 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.

- 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?

2.4 Accounting for Prediction Costs

The results presented above demonstrate the great potential for using predictions to improve scheduling performance. However, the models 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 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 [72], 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, [72] derives the formulas for SPRPT and 1-bit predictions under both cost models, which can again be done using a SOAP analysis [69]. However, [72] argues that the introduction of costs allows for more interesting models and scheduling strategies. They focus on a setting where 1-bit predictions are cheap compared to a prediction of the service time. In such a setting, it may make sense to use the cheap 1-bit prediction for all jobs, but only use the 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 it also can be analyzed using SOAP analysis.

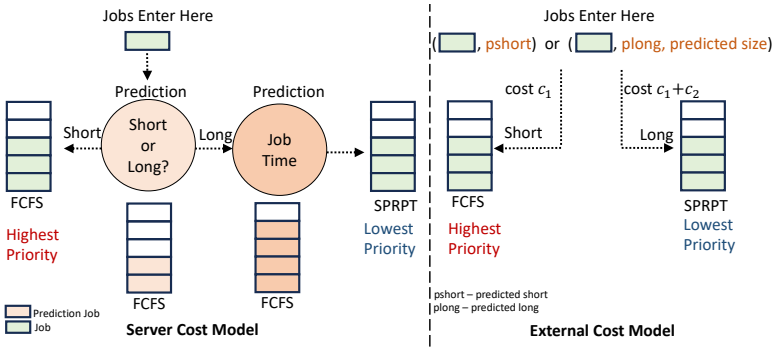


Figure 2: SkipPredict framework under the server cost model and external cost model.

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 1-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 a 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 1-bit predictions are either not much cheaper than full predictions, or not available. DelayPredict framework is shown in Figure 3.

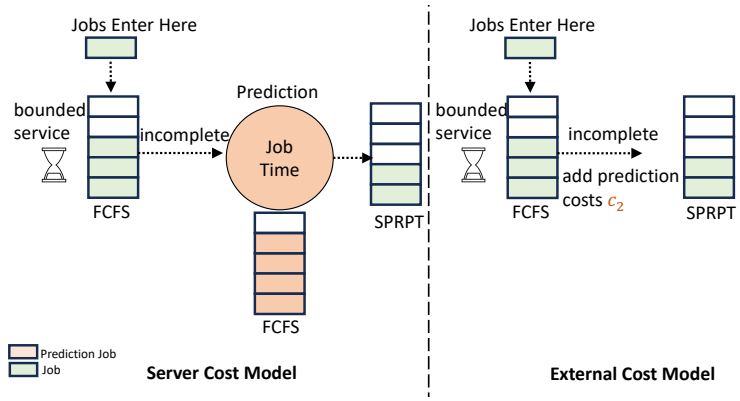


Figure 3: DelayPredict framework under the server cost model and external cost model.

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.
- 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), while jobs without predictions are handled via FCFS?
- Can we design systems that use dedicated prediction servers effectively?

2.5 Multiple server systems

The above work has focused on using predictions in the setting of the M/G/1 queue. The study of larger systems or networks of queues using predictions remains relatively open for further study. Most previous work has been empirical [20, 21], such as the work [49] which looks at the power of two choice paradigm [46, 76] when using predictions. 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.

A small step forward appears in the work on 1-bit predictions [48], where a variant of the power of two choices is considered in the setting of 1-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 [48], 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. [] 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.

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 1-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 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

Large Language Models (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 [59], automate code development [18], and accelerate scientific research through data analysis and literature synthesis [36]. ChatGPT [5] 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 makes 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. Since many applications rely on pre-trained models for inference instead of training their own, we focus on scheduling during inference, although scheduling considerations may also apply to training LLMs.

To explain how scheduling applies 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 these differences introduce.

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.

Transformer Architecture

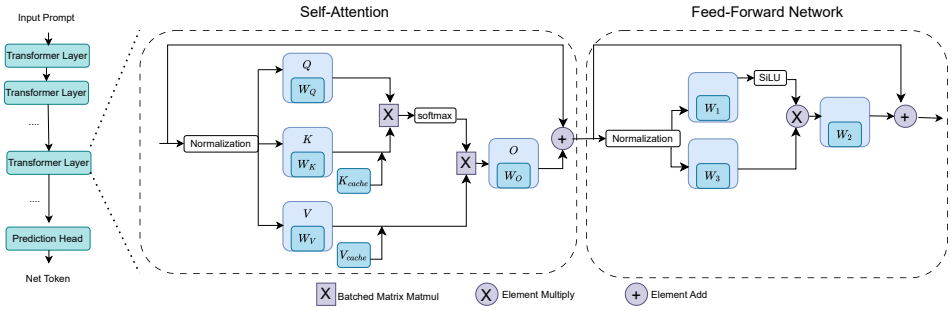


Figure 4: Transformer architecture

Most of today's LLMs adopt a decoder only transformer architecture [14, 58]. 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 to refine the representation of the input as shown in Figure 4.

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 sequence length and d is the hidden dimension (the embedding size of each token), the model applies learned linear transformations to produce the query (Q_{pre}), key (K_{pre}), and value (V_{pre}) matrices:

$$Q_{\text{pre}} = X_{\text{pre}} W_q, \quad K_{\text{pre}} = X_{\text{pre}} W_k, \quad V_{\text{pre}} = X_{\text{pre}} W_v$$

where $W_q, W_k, W_v \in \mathbb{R}^{d \times d_k}$ 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 K_{pre}, 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:

$$O_{\text{pre}} = \text{softmax}\left(\frac{Q_{\text{pre}} K_{\text{pre}}^T}{\sqrt{d_k}}\right) V_{\text{pre}} W_o + X_{\text{pre}} \tag{1}$$

where W_q, W_k, W_v, and 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.

Feed forward network. The output of the self-attention module is sent to the Feed-Forward Network (FFN), which refines the attention output. This network introduces non-linearity, allowing the model to capture more complex patterns in the data.

$$\text{FFN}(x) = (\text{SiLU}\,(xW_1) \times xW_3)\, W_2,$$

where W_1, W_2, W_3 are linear modules. The SiLU (Sigmoid Linear Unit) activation function is defined as: $\text{SiLU}(x) = x \cdot \sigma(x) = x \cdot \frac{1}{1+e^{-x}}$, where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, defined as:

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 (X_{pre}) and generates key-value pairs that are stored in the Key-Value (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 $X_{\text{dec}} \in \mathbb{R}^{1 \times d}$

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.

$$Q_{\text{dec}} = X_{\text{dec}} W_q, \quad K_{\text{cat}} = [K_{\text{cache}}, X_{\text{dec}} W_k], \quad V_{\text{cat}} = [V_{\text{cache}}, X_{\text{dec}} W_v]$$

The attention output in the decode phase is computed as:

$$O_{\text{dec}} = \text{softmax}\left(\frac{Q_{\text{dec}} K_{\text{cat}}^T}{\sqrt{d_k}}\right) V_{\text{cat}} W_o + X_{\text{dec}} \tag{2}$$

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.

Execution on Hardware

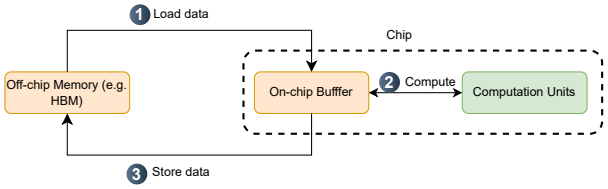


Figure 5: A neural network layer is executed on hardware devices by transferring data from memory (e.g., HBM) to on-chip buffers, then computing with the on-chip processing units, and eventually sending the output data back to memory.

The rapid progress in LLMs is closely tied to advancements in hardware accelerators, particularly GPUs (Graphics Processing Units). 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 balance 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 while computations are processed. Alternatively, a *memory-bandwidth bound* arises 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 upfront. 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:

- **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:** While larger models often produce higher-quality responses, the link between model size and accuracy varies with the request type.

We focus on the question of reducing overall latency (which we refer to henceforth or 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 tradeoff 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: Time To First Token (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:

$$t_{waiting} + \text{TTFT}(n_{\text{input}}) + n_{\text{output}} \cdot \text{TPOT}. \tag{3}$$

3.3 A summary: The job vs. system perspective

Before going into details regarding LLM scheduling, we summarize the operations we have described, from two perspectives: the job[2] perspective and the system perspective.

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 key-value (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 [86]; 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.

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 [86], 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.

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 intra-layer scheduling, although exploring this finer granularity remains an interesting possible direction.

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

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 also summarize some of these key issues.

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.

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. Non-preemptive policies, such as First-Come, First-Served (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 to 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 ms, whereas transferring KV tensors between host and GPU memory over PCIe 4.0×16 at full bandwidth takes about 36 ms. Existing approaches [4, 84] attempt to optimize offloading and reloading by overlapping these operations with computation. However, the available memory budget for such overlap poses a fundamental constraint.

Multi-stage processing. Requests in LLM systems can be viewed as multi-stage jobs in queueing theory, though they differ from standard queueing models. Mixed-phase batches, where some requests remain in the prefill phase while others have advanced to decoding, can prolong the overall decode phase since the longer prefill phase may dominate the iteration time. Two strategies have been proposed to mitigate this imbalance. The first, *chunked prefill* (Sarathi [6]), 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* (Splitwise [53, 91]), 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, [53] 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 [86] (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

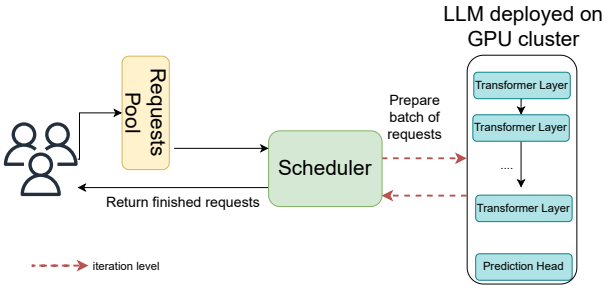


Figure 6: Iteration-level scheduling, dynamically adjusting the batch and enabling preemptive decisions. After each token generation, the scheduler evaluates whether to continue with the current request or switch to another pending request, in addition to returning finished outputs to users as soon as they are ready.

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, while 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 (SRPT). 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 prior. 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.

Several works propose methods to predict request sizes. Zhen et al. [90] use an auxiliary LLM model to predict response sizes and then prioritize requests based on those predictions. While this strategy reduces response time, it introduces additional computational overhead from the extra model used for size prediction. S^3 [35] fine-tunes a BERT model [63] to predict output sequence sizes from input prompts. The studies in [19, 35, 74] address size prediction as a classification task, where predictions correspond to one of several buckets representing a range of sizes, whereas [56, 57], 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. LTR [26] 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 [84] adopts a Multi-Level Feedback Queue (MLFQ) to avoid head-of-line blocking. However, frequent preemptions with MLFQ increase the cost of managing the KV cache and transferring it to the CPU, leading to additional memory and performance overhead.

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 [71] 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* [11, 32, 33], 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 $c \cdot$ predicted request size (with $0 \leq c \leq 1$), disabling preemption once a request's age exceeds this threshold.

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 a 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? Since 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?

4.2 Adaptive Scheduling

While 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 multi-objective 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, [73] reports that 85% to 97% of tokens in a prompt may be shared with other prompts. Such sharing occurs in settings such as conversational agents [7], tool use [55, 64], question answering [40, 60, 79], complex reasoning [12, 80], batch inference [41], in-context learning [23], and agent systems [16, 27, 29]. [37] and [73]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 KV cache entries reuse and balance computational load across GPUs using data parallelism without explicitly optimizing for latency.

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 multi-factor 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 [34] 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 non-interactive workloads. This challenge can be addressed within a multi-objective scheduling framework that balances latency and fairness, using queueing theory to prevent starvation.

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 non-interactive 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 standalone small requests?

4.3 GPU Resource Allocation

(a) Pooled GPUs

(b) Dedicated GPUs for prefill and decode

Figure 7: GPU organizations: comparing pooled GPUs with dedicated GPUs for prefill and decode phases.

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, while 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 introduces 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 arrangement is similar to tandem servers in queueing systems, 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 while decode resources remain idle). In heterogeneous GPU environments, mismatches between GPU capabilities and different phases (prefill vs. 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.

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 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. For example, LLMs can orchestrate external API calls to fetch up-to-date information or perform computations, and agent-based approaches enable LLMs to autonomously plan and execute tasks across 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

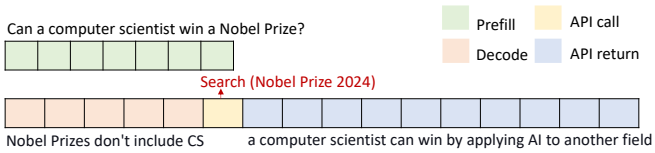


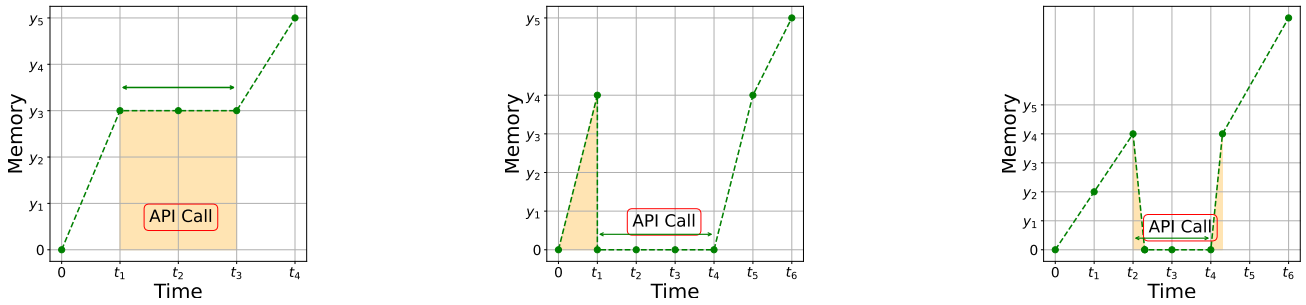
Figure 8: Illustration of an augmented-LLM request. The API fetches detailed information about the 2024 Nobel Prize.

Augmented Language Models [45, 77] enhance traditional LLM capabilities by incorporating external tools or retrieval mechanisms. Unlike pure LLMs that rely solely on pre-trained 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* (Application Programming Interfaces), fall into three main categories as described in [45]: incorporating non-LLM tools during decoding (such as calculators [83], information retrieval systems [10]), 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 [15], DSpy [38], Gorilla [54], SGLang [88], and AgentGraph [17]).

API call durations vary significantly between augmentation types, distinguishing short-running 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 while the LLM is idle. *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 Figures 9(a), 9(b), and 9(c) indicating the memory waste for a single request due to an API call.



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

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. The highlighted area represents memory waste for one request.

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 [4] proposes a method to determine the handling strategy when a request reaches an API call; however, it lacks integrated scheduling policies to proactively minimize latency and relies on a 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 in [70] is the first to propose scheduling policies beyond FIFO for augmented LLMs, presenting a system called MARS. MARS 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 lifecycle, factoring in both request size and API interactions. MARS achieves end-to-end latency improvements of 27%-85% and reductions in time-to-first-token (TTFT) of 4%-96% compared to INFERCEPT, with even greater gains over vLLM [39], an LLM serving system that applies paged attention to reduce memory overhead.

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 endpoints while considering real-time load with possible quality tradeoffs?

5.2 Multiple LLMs Avaiable

AI development is moving toward compound AI systems [87] that integrate multiple components, often including LLMs of varying sizes and capabilities. Model size significantly impacts runtime, cost, and answer quality in transformer-based LLMs, as shown in Figure 10. Smaller models may suffice for straightforward queries, while larger models provide deeper answers to complex prompts.

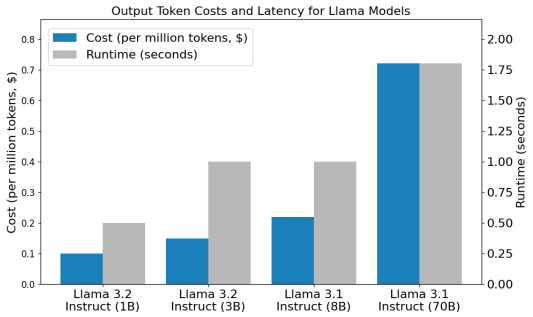


Figure 10: Comparison of output token costs (per million tokens) for the Llama 3 family on Amazon Bedrock and the time (in seconds) to generate 100 tokens, measured on an NVIDIA H100 GPU.

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 an NVIDIA H100 GPU, Llama 3.2 Instruct (1B) generates 100 tokens in 0.5 seconds, whereas Llama 3.1 Instruct (70B) requires 2 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 to $0.72 for Llama 3.1 Instruct (70B).[3] 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 [52] 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 datasets.

Open Questions:

- 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?

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 sub-jobs, where each stage processes and refines the previous stage's output, iteratively developing the response.

Both approaches present significant research challenges. While 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 inter-stage transitions. The objective is to minimize end-to-end latency despite variability in execution times and communication overhead. Though 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, while the DAG-based model concentrates on orchestrating a chain of dependent operations.

Open Questions:

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

- How can we design scheduling algorithms for multi-LLM systems, both in the setting of independent, parallel, and in the setting of processing sequential, interdependent tasks? Where can predictions aid in scheduling decisions for 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) multi-stage 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 [78] (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 [85] 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 [24, 30], 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 [2, 3, 28] that have been trained to generate extended chain-of-thought [81] 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 multi-run aggregation or tree search.

Prior works on serving systems [6, 53, 39, 71, 26] 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. ParrotServe [42] and SGLang [89] both target multi-request applications. ParrotServe offers an abstraction that allows users to specify dependencies between requests, enabling more effective cross-request scheduling. SGLang introduces programming primitives tailored for multi-request 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 [25] 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. While 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 additional information can be predicted and used to inform more effective scheduling decisions.

LLM reasoning systems thus present a rich ground for developing adaptive scheduling algorithms that allocate resources based on intermediate reasoning outputs. For example, if early reasoning steps indicate a clear and promising trajectory toward a correct answer, the scheduler might continue to invest in additional computational resources. Conversely, if early signals suggest divergence, the system could preempt the process and reallocate resources to other tasks. Moreover, there is potential for exploring parallel evaluation of multiple reasoning paths, which raises further challenges regarding load balancing, redundancy, and optimal resource sharing.

Open Questions:

- Since 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 pre-run 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 exploring 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 Large Language Model (LLM) systems, particularly for inference. However, LLMs introduce particular scheduling challenges due to 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 paricular new problems and issues 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.

References

[1] [n. d.]. Algorithms with Predictions Paper List. https://algorithms-with-predictions.github.io.

[2] [n. d.]. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/.

[3] [n. d.]. QwQ: Reflect Deeply on the Boundaries of the Unknown. https://qwenlm.github.io/blog/qwq-32b-preview/.

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

[5] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

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

[7] Anthropic. 2024. Prompt caching with Claude. https://www.anthropic.com/news/prompt-caching

[8] Yossi Azar, Stefano Leonardi, and Noam Touitou. 2021. Flow time scheduling with uncertain processing time. In *STOC*. 1070–1080. https://doi.org/10.1145/3406325.3451023

[9] Yossi Azar, Stefano Leonardi, and Noam Touitou. 2022. Distortion-Oblivious Algorithms for Minimizing Flow Time. In *ACM-SIAM*. 252–274. https://doi.org/10.1137/1.9781611977073.13

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

[11] Yonatan Belinkov. 2022. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics* 48, 1 (2022), 207–219.

[12] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 16 (March 2024), 17682–17690. https://doi.org/10.1609/aaai.v38i16.29720 arXiv:2308.09687 [cs].

[13] Joan Boyar, Lene M Favrholdt, Christian Kudahl, Kim S Larsen, and Jesper W Mikkelsen. 2017. Online algorithms with advice: A survey. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 1–34.

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

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

[16] Justin Chih-Yao Chen, Swarnadeep Saha, and Mohit Bansal. 2024. ReConcile: Round-Table Conference Improves Reasoning via Consensus among Diverse LLMs. https://doi.org/10.48550/arXiv.2309.13007 arXiv:2309.13007 [cs].

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

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

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

[20] Matteo Dell'Amico. 2019. Scheduling with inexact job sizes: The merits of shortest processing time first. *arXiv preprint arXiv:1907.04824* (2019).

[21] Matteo Dell'Amico, Damiano Carra, and Pietro Michiardi. 2015. PSBS: Practical size-based scheduling. *IEEE Trans. Comput.* 65, 7 (2015), 2199–2212.

[22] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, Aidin Niaparast, and Sergei Vassilvitskii. 2024. Binary Search with Distributional Predictions. *arXiv preprint arXiv:2411.16030* (2024).

[23] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A survey on in-context learning. `https://doi.org/10.48550/arXiv.2301.00234` arXiv:2301.00234 [cs].

[24] Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. 2023. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179* (2023).

[25] Yichao Fu, Junda Chen, Siqi Zhu, Zheyu Fu, Zhongdongming Dai, Aurick Qiao, and Hao Zhang. 2024. Efficiently Serving LLM Reasoning Programs with Certaindex. *arXiv preprint arXiv:2412.20993* (2024).

[26] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. *arXiv preprint arXiv:2408.15792* (2024).

[27] Shanghua Gao, Ada Fang, Yepeng Huang, Valentina Giunchiglia, Ayush Noori, Jonathan Richard Schwarz, Yasha Ektefaie, Jovana Kondic, and Marinka Zitnik. 2024. Empowering Biomedical Discovery with AI Agents. `https://doi.org/10.48550/arXiv.2404.02831` arXiv:2404.02831 [cs].

[28] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[29] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. `https://doi.org/10.48550/arXiv.2402.01680` arXiv:2402.01680 [cs].

[30] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992* (2023).

[31] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press.

[32] John Hewitt and Percy Liang. 2019. Designing and interpreting probes with control tasks. *arXiv preprint arXiv:1909.03368* (2019).

[33] John Hewitt and Christopher D Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* 4129–4138.

[34] Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St Amant, Chetan Bansal, Victor Rühle, Anoop Kulkarni, et al. 2025. Serving Models, Fast and Slow: Optimizing Heterogeneous LLM Inferencing Workloads at Scale. *arXiv preprint arXiv:2502.14617* (2025).

[35] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. *Advances in Neural Information Processing Systems* 36 (2023), 18015–18027.

[36] A Jo. 2023. The promise and peril of generative AI. *Nature* 614, 1 (2023), 214–216.

[37] Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y. Fu, Christopher Ré, and Azalia Mirhoseini. 2024. Hydragen: High-Throughput LLM Inference with Shared Prefixes. `https://doi.org/10.48550/arXiv.2402.05099` arXiv:2402.05099.

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

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

[40] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts? `https://doi.org/10.48550/arXiv.2311.04939` arXiv:2311.04939 [cs].

[41] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. `https://doi.org/10.48550/arXiv.2203.07814` arXiv:2203.07814.

[42] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of {LLM-based} Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 929–945.

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

[44] Thodoris Lykouris and Sergei Vassilvitskii. 2021. Competitive caching with machine learned advice. *Journal of the ACM (JACM)* 68, 4 (2021), 1–25.

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

[46] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[47] Michael Mitzenmacher. 2020. Scheduling with predictions and the price of misprediction. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA (LIPIcs, Vol. 151)*. 14:1–14:18.

[48] Michael Mitzenmacher. 2021. Queues with small advice. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM, 1–12.

[49] Michael Mitzenmacher and Matteo Dell'Amico. 2022. The supermarket model with known and predicted service times. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 2740–2751.

[50] Michael Mitzenmacher and Sergei Vassilvitskii. 2020. Algorithms with Predictions. In *Beyond the Worst-Case Analysis of Algorithms*, Tim Roughgarden (Ed.). Cambridge University Press, 646–662. `https://doi.org/10.1017/9781108637435.037`

[51] Michael Mitzenmacher and Sergei Vassilvitskii. 2022. Algorithms with predictions. *Commun. ACM* 65, 7 (2022), 33–35. `https://doi.org/10.1145/3528087`

[52] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. 2024. RouteLLM: Learning to Route LLMs with Preference Data. *arXiv preprint arXiv:2406.18665* (2024). `https://doi.org/10.48550/arXiv.2406.18665`

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

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

[55] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. `https://doi.org/10.48550/arXiv.2307.16789` arXiv:2307.16789 [cs].

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

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

[58] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[59] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. In *International conference on machine learning*. Pmlr, 8821–8831.

[60] Ruchit Rawal, Khalid Saifullah, Miquel Farré, Ronen Basri, David Jacobs, Gowthami Somepalli, and Tom Goldstein. 2024. CinePile: A Long Video Question Answering Dataset and Benchmark. `https://doi.org/10.48550/arXiv.2405.08813` arXiv:2405.08813 [cs].

[61] Shaik Mohammed Salman, Van-Lan Dao, Alessandro Vittorio Papadopoulos, Saad Mubeen, and Thomas Nolte. 2023. Scheduling firm real-time applications on the edge with single-bit execution time prediction. In *ISORC*. 207–213.

[62] Shaik Mohammed Salman, Alessandro Vittorio Papadopoulos, Saad Mubeen, and Thomas Nolte. 2023. Evaluating Dispatching and Scheduling Strategies for Firm Real-Time Jobs in Edge Computing. In *IECON*. 1–6.

[63] V Sanh. 2019. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108* (2019).

[64] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 68539–68551. `https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html`

[65] Ziv Scully, Isaac Grosof, and Mor Harchol-Balter. 2020. The Gittins policy is nearly optimal in the M/G/k under extremely general conditions. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[66] Ziv Scully, Isaac Grosof, and Michael Mitzenmacher. 2022. Uniform Bounds for Scheduling with Job Size Estimates. In *13th Innovations in Theoretical Computer Science Conference, ITCS (LIPIcs, Vol. 215)*, Mark Braverman (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 114:1–114:30.

[67] Ziv Scully and Mor Harchol-Balter. 2018. SOAP bubbles: Robust scheduling under adversarial noise. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 144–154.

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

[69] Ziv Scully, Mor Harchol-Balter, and Alan Scheller-Wolf. 2018. SOAP: One clean analysis of all age-based scheduling policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 1–30.

[70] Rana Shahout, Cong Liang, Shiji Xin, Qianru Lao, Yong Cui, Minlan Yu, and Michael Mitzenmacher. 2024. Fast Inference for Augmented Large Language Models. *arXiv preprint arXiv:2410.18248* (2024).

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

[72] Rana Shahout and Michael Mitzenmacher. 2024. SkipPredict: When to Invest in Predictions for Scheduling. In *Annual Conference on Neural Information Processing Systems (NeurIPS 2024)*. `http://papers.nips.cc/paper_files/paper/2024/hash/becd02b89259774da2ede23116a80648-Abstract-Conference.html`

[73] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2024. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. `https://openreview.net/forum?id=meKEKDhdnx`

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

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

[76] Nikita D. Vvedenskaya and Yuri M. Suhov. 1997. Dobrushin's mean-field approximation for a queue with dynamic routing. *Markov Processes and Related Fields* 3, 4 (1997), 493–526.

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

[78] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).

[79] Yubo Wang, Xueguang Ma, and Wenhu Chen. 2024. Augmenting Black-box LLMs with Medical Textbooks for Biomedical Question Answering. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 1754–1770. `https://doi.org/10.18653/v1/2024.findings-emnlp.95`

[80] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, 24824–24837.

[81] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[82] Adam Wierman and Misja Nuyens. 2008. Scheduling despite inexact job-size information. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 25–36.

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

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

[85] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724* (2024).

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

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

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

[89] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2025. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems* 37 (2025), 62557–62583.

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

[91] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670* (2024).