STAR: Decode-Phase Rescheduling for LLM Inference

Zhibin Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
wzbwangzhibin@gmail.com

Zetao Hong
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
mariohong128@gmail.com

Xue Li
Alibaba Group
Hangzhou, China
youli.lx@alibaba-inc.com

Zibo Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
wangzb@smail.nju.edu.cn

Shipeng Li
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
spli@smail.nju.edu.cn

Qingkai Meng
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
qkmeng@nju.edu.cn

Qing Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
wangqing.cs@nju.edu.cn

Chengying Huan
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
huanchengying@nju.edu.cn

Rong Gu*
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
gurong@nju.edu.cn

Sheng Zhong
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
sheng.zhong@gmail.com

Chen Tian
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
tianchen@nju.edu.cn

Abstract

Large Language Model (LLM) inference has emerged as a fundamental paradigm, however, variations in output length cause severe workload imbalance in the decode phase, particularly for long-output reasoning tasks. Existing systems, such as PD disaggregation architectures, rely on static prefill-to-decode scheduling, which often results in SLO violations and OOM failures under evolving decode workloads. In this paper, we propose STAR, a decode rescheduling system powered by length prediction to anticipate future workloads. Our core contributions include: (1) A lightweight and continuous LLM-native prediction method that leverages LLM hidden state to model remaining generation length with high precision (reducing MAE by 49.42%) and low overhead (cutting predictor parameters by 93.28%); (2) A rescheduling solution in decode phase with a dynamic balancing mechanism that integrates current and predicted workloads, reducing P99 TPOT by 75.1% and achieving 2.63× higher goodput.

*Corresponding author.

HPDC '26, Cleveland, OH, USA

ACM ISBN 979-8-4007-2640-8/2026/07
https://doi.org/10.1145/3806645.3807813

CCS Concepts

• **Software and its engineering** → **Scheduling**; • **Computer systems organization** → *Cloud computing*; • **Computing methodologies** → *Distributed algorithms.*

Keywords

LLM Serving; Decode Rescheduling; Generation Length Prediction; Load Balancing; Prefill-Decode Disaggregation

ACM Reference Format:
Zhibin Wang, Zetao Hong, Xue Li, Zibo Wang, Shipeng Li, Qingkai Meng, Qing Wang, Chengying Huan, Rong Gu, Sheng Zhong, and Chen Tian. 2026. STAR: Decode-Phase Rescheduling for LLM Inference. In *The 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26), July 13–16, 2026, Cleveland, OH, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3806645.3807813

1 Introduction

With the great success of Large Language Models (LLMs) such as ChatGPT [26], Claude [3], Qwen [2], Gemini [13], and DeepSeek [7], the demand for LLM services is surging. Despite their success, the deployment of LLMs imposes heavy requirements on hardware resources, leading to high serving costs. Consequently, vast systems and algorithms [5, 12, 25] are proposed to improve efficiency and reduce the cost of LLM serving.

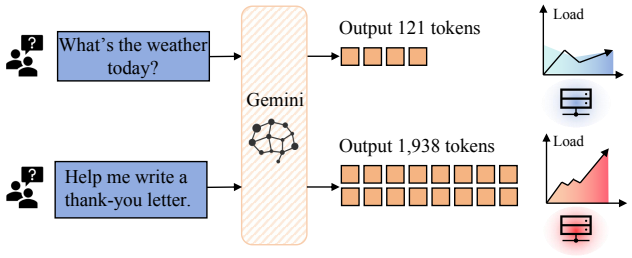


Figure 1: Different output lengths lead to significant load variations across decode instances.

The autoregressive inference process of LLMs results in significant workload variations, especially in reasoning tasks that require long chain-of-thought (CoT) outputs [35]. As illustrated in Figure 1, when using Gemini to process two different real-world inputs, the resulting output lengths differ by more than 16 times. The workload of LLM inference for a given model is determined by the length of the input and output, as inference of a request involves two stages: 1) input-related prefill processes the input prompt with a single forward pass to generate the initial output token and the key-value (KV) cache; 2) output-related decode generates output tokens auto-regressively, i.e., one token at a time, by leveraging the KV cache. As revealed in [9], the decode phase dominates the total cost in LLM inference, especially for long outputs. Therefore, the variation in output length leads to significant workload imbalance across requests during the decode phase.

Specifically, the variation of workload in the decode phase leads to two critical issues:

- **Issue 1: OOM of KV cache.** In the decode phase, uneven distribution of workload across instances leads to overloaded instances accumulating large numbers of tokens, which causes their KV cache memory usage to rapidly grow. Although techniques like PagedAttention [18] enable more flexible KV cache management, the total KV cache memory usage on overloaded instances can still exceed the hardware memory limit as long-output generations continue, resulting in "OOM errors". This risk also remains practical in real deployments, where concurrency is typically tuned for throughput rather than the worst-case KV-cache footprint of every request. For example, Anyscale explicitly warns that increasing the concurrent-sequence limit can cause OOM errors in long-context LLM serving [4]. Once OOM occurs, all affected requests on that instance are forced to recompute KV cache, greatly increasing their latency and reducing system throughput. Meanwhile, the overloaded (now OOM) instance's performance drops, its requests accumulate, and new requests are more likely to be assigned to other instances, causing further imbalance and overloading those as well. This chain reaction can produce cascading OOM errors and lead to overall system instability.
- **Issue 2: Violations of SLO.** Considering the distinct characteristics of prefill and decode, modern LLM serving systems [28, 39] adopt a prefill-decode (PD) disaggregation architecture that separates the single, but relatively long, prefill iteration from the multiple, but shorter, decode iterations to prevent phase interference. Therefore, the time-per-output-token (TPOT) of decode will not be affected by the prefill phase. However, as the time to read the KV cache will increase linearly as the output length increases, long-output requests may lead to a significant increase in TPOT, which may violate the SLO of TPOT.

However, existing LLM serving systems [1, 18, 28, 37, 39] are not prepared to handle workload imbalance posed by long output reasoning tasks. 1) Several works [28] consider the load of each prefill as well as the cache of KV cache to *assign the incoming request to a prefill instance*, but lack consideration of the long-term workload of decode instances. 2) Other works [20, 34] further consider the workload-balancing scheduling when *dispatching requests from prefill to decode*: Round-robin scheduling [34] assigns requests to decode instances in a round-robin manner, ensuring an even distribution of requests, however, it overlooks the varying workloads of different requests; Current-load-balancing scheduling [20] allocates requests based on the current load of each decode instance, i.e., size of KV cache, aiming to balance memory usage, but the evolving nature of requests results in imbalancing after a period of execution. Additionally, Llumnix [31] proposes dynamic runtime rescheduling via live migration to improve isolation, mitigate fragmentation, and support priorities with strong tail-latency gains; however, it does not consider the prefill-decode disaggregation architecture, the decode characteristics under long outputs, nor future execution states (e.g., remaining length), relying mainly on current-state decisions.

In this paper, we propose to address the above limitations with the rescheduling in decode phase. Specifically, we aim to enable the migration of requests across decode instances during the decode phase to resolve workload imbalance, thereby improving SLO compliance and preventing OOM errors. However, implementing decode rescheduling presents two key challenges:

Challenge 1: Accurate and efficient modeling of workload. In addition to the current workload of each decode instance, the scheduler must also consider the future decode workload contributed by currently active requests, which can be modeled by their remaining generation lengths. However, accurately predicting the remaining generation length is challenging due to the high variance and unpredictability of output lengths in real-world scenarios. Existing methods either 1) rely on auxiliary models [11, 16, 17, 29], which introduce additional computational overhead and limit by the ability of auxiliary models; or 2) inject tokens into the input prompt [38] through prompt engineering, which is intrusive and may affect the quality of the generated output. Worse still, for same input prompt, the output will vary due to the temperature sampling and nondeterminism of LLM system, making accurate prediction with only input prompt impossible.

Challenge 2: Effective rescheduling strategy in complex decision space. Unlike traditional task scheduling, decode rescheduling must consider both the current load of each instance and the remaining-token workload of active requests. It must also decide which request to migrate and when, since migrating a near-complete request may not amortize migration overhead. This enlarges the decision space relative to traditional prefill-to-decode scheduling.

In response to the above challenges, we introduce STAR, abbreviated for smart token-length aware rescheduling. As far as we know, STAR is the first system to consider decode rescheduling to resolve workload imbalance across decode instances. STAR achieves workload balancing through two key components:

Lightweight and continuous LLM-native predictor (Section 4) efficiently and accurately estimates the remaining generation length of each request. Specifically, instead of relying on auxiliary models or prompt engineering, we leverage the internal state of the original LLM to predict the remaining generation length. By feeding the hidden state of the last token from the final transformer layer into a lightweight MLP predictor, we achieve both lower prediction overhead and more accurate results. Compared to the SOTA, our method reduces prediction mean absolute error (MAE) by 49.42% on average while reducing overhead by 96.26% for output lengths up to 32K tokens. Moreover, we observe that by leveraging the additional generated tokens, we can further improve the prediction precision. With the low overhead of our LLM-native predictor, we can perform continuous prediction in an iterative manner, further enhancing precision in decision-making of decode rescheduling.

Multi-stage rescheduling strategy (Section 5) that effectively balances the workload across decode instances. We first align the execution time and memory usage of each decode instance through the number of tokens in the batch, facilitating the further modeling of workload. Then, we design a multi-stage rescheduling strategy that 1) identifies overloaded and underloaded decode instances considering both current and predicted workload; 2) enumerates requests for each overloaded-underloaded instance pair and filters requests that can benefit from migration; 3) simulates the migration for each request and selects the best feasible migration that maximizes workload variance reduction. By periodically executing this rescheduling strategy, we can achieve dynamic workload balancing across decode instances, thereby improving SLO compliance and preventing OOM errors.

Comprehensive evaluations (Section 6) demonstrate that STAR significantly outperforms SOTA LLM serving systems. Compared to PD disaggregation implemented in vLLM [18], STAR achieves up to 2.63× higher goodput, reduces P99 TPOT latency by 75.1%, and prevents OOM occurrences. Moreover, the simulation on larger-scale clusters shows that rescheduling effectively improves cluster load balancing, while prediction further reduces load fluctuations.

2 Background

2.1 Prefill and Decode Phases

Large language model inference has two phases: prefill and decode. Prefill processes the full input in one forward pass to generate the first token and build the KV cache; it is compute-bound and handled per request. Decode then generates tokens auto-regressively using the KV cache; it is memory-bound and typically batches requests for efficiency. Their SLOs differ: prefill minimizes Time-to-First-Token (TTFT), while decode reduces Time-per-Output-Token (TPOT).

Recognizing the distinct characteristics of these two phases, modern LLM serving systems (e.g., Mooncake [28], DistServe [39]) separate prefill and decode onto different hardware resources to satisfy their respective resource demands. Upon arrival, a request exclusively occupies a prefill instance, which queues inputs in FIFO order and selects instances based on load or KV cache reuse [28]. After the prefill phase, the request will be forwarded to a decode instance to generate tokens auto-regressively. Instead of queuing requests, decode instances batch multiple requests together as proposed in vLLM [18], Orca [37] to improve hardware utilization.

Particularly, taking deployment of DeepSeek-V3 [9] as an example, to fully utilize the hardware resources, a prefill pod leverages 32 H800 GPUs, while a decode pod utilizes 320 GPUs, which highlights that decode is the resource-consuming phase. In the subsequent sections, we will delve deeper into the decode phase.

2.2 Imbalance Workload in Decode Phase

Given the autoregressive nature of decode, the workload of each request is determined by the output length. Therefore, we first analyze the characteristics of output length in real-world scenarios.

Output length variation. Figure 2 illustrates the output length distribution in ShareGPT dataset when running DeepSeek-R1-Distill-Qwen-7B model with a maximum output length of 32K tokens.

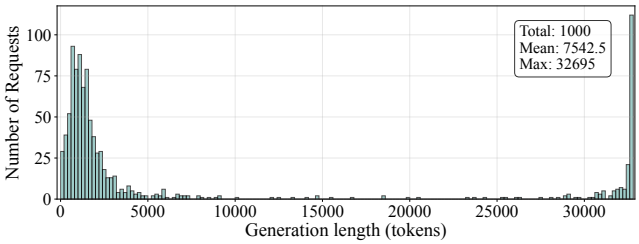
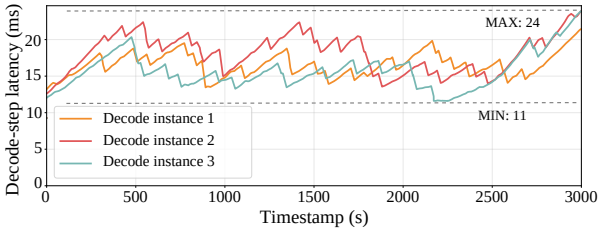
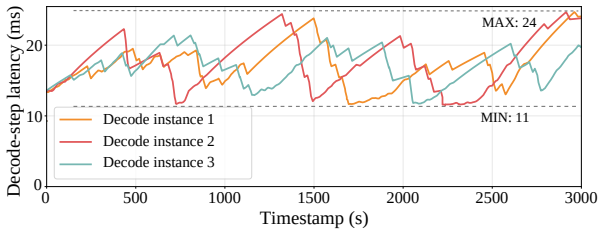


Figure 2: Output length distribution.



(a) Round-robin scheduling



(b) Current-load balancing

Figure 3: Per-instance decode-step latency over time across three decode instances under PD disaggregation (1 prefill + 3 decode instances).

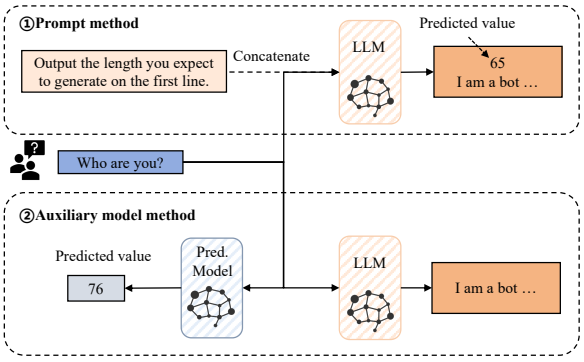


Figure 4: Current prediction methods.

Many requests present short outputs, with 29.2% requests generating fewer than 1K tokens, while a non-ignorable portion of requests produce very long outputs (17.3% with exceeding 30K tokens).

Existing prefill-to-decode scheduling. While PD disaggregation effectively eliminates phase interference, it introduces a critical limitation: it relies on statically assigning decode instances to achieve load balance. However, relying solely on prefill-to-decode scheduling is insufficient to address the workload imbalance within the decode phase, as it does not adapt to the varying computational and memory demands across decode instances.

- **Round-robin scheduling [34]:** This straightforward approach assigns requests to decode instances in a round-robin manner, ensuring an even distribution of requests. However, it overlooks the varying workloads of different requests, leading to potential load imbalances.
- **Current-load balancing [20]:** This method allocates requests based on the current KV cache size of each decode instance, aiming to balance memory usage. While it considers the memory footprint, it still fails to account for the actual computational load associated with generating output tokens, which can vary significantly between requests.

Figure 3 illustrates the TPOT variation across three decode instances under different prefill-to-decode scheduling strategies with the same workload in Figure 2 and request rate of 0.1 per second. Even with initial load balancing during prefill-to-decode handoff, significant TPOT divergence emerges with the progression of generation. Decode instances exhibit rapidly escalating performance disparities as output sequences lengthen—requests with prolonged residency on a single instance dominate its resources, causing cascading TPOT spikes for subsequent requests.

Summarization and Motivation. Existing prefill-to-decode scheduling strategies cannot effectively balance the workload across decode instances, especially in scenarios with long and variable output lengths. Rescheduling during the decode phase is necessary to dynamically adjust to workload variations.

2.3 Generation Length Prediction

Accurate prediction of remaining output length is critical for modeling future workload in LLM inference systems. There are two main approaches—prompt method and auxiliary model method—as well as an additional iterative refinement method.

- **Prompt-based methods** such as Perception in Advance (PiA) [38] modify user instructions to have the LLM first predict its output length before generating the response (Figure 4). While achieving reasonable prediction accuracy, this approach requires intrusive modifications to user prompts, potentially altering model behavior and output quality.[11] As LLM applications have matured, such user-facing interventions have become increasingly unacceptable, rendering this approach impractical for modern serving systems.
- **Auxiliary model methods** employing smaller models like opt or bert (Figure 4) with added prediction heads [17, 29, 30]. Sequence Scheduling [38] established that prediction accuracy correlates with model capability, yet these auxiliary models are orders of magnitude smaller than target LLMs, fundamentally limiting their contextual understanding and prediction accuracy. Our evaluation shows that MAE increases by 3.2× when context length grows from 4K to 32K tokens, making these methods particularly ineffective for long output sequences.

Summarization and Motivation. Existing methods still suffer from poor accuracy and high overhead. This is arising from lack of consideration of integrating the prediction model with the target LLM. How to leverage the target LLM's own capabilities to provide accurate predictions with minimal overhead remains unsolved.

2.4 Challenges

In summary, to balance the inference workload across decode instances in PD disaggregation, we identify two key challenges:

- **Accurate and efficient prediction of remaining output length:** Existing methods either require intrusive prompt modifications, rely on less capable auxiliary models with poor accuracy for long outputs, or incur prohibitive overheads that prevent iteration-level application. A new approach is needed that leverages the target LLM's own capabilities to provide precise predictions with minimal computational cost.
- **Complex dynamic rescheduling in decode phase:** Current PD disaggregation systems lack mechanisms for runtime migration between decode instances once requests are initially assigned. Extending traditional prefill-to-decode scheduling to enable such dynamic rescheduling is non-trivial, since decode-to-decode migration must: 1) account for the future system state, i.e., the predicted remaining output lengths of active requests, and 2) operate over a more complex decision space—not only deciding which decode instance to migrate to, as in prior work, but also when to migrate and which request to evict from a decode instance.

3 Overview

Figure 5 illustrates the overview of STAR built on the PD disaggregation architecture. When a request arrives, it is first sent to a prefill instance to process the input prompt and generate KV cache. After the prefill phase is completed, the request will be forwarded to a decode instance according to its input length, predicted output length, and the current load of each decode instance. During the

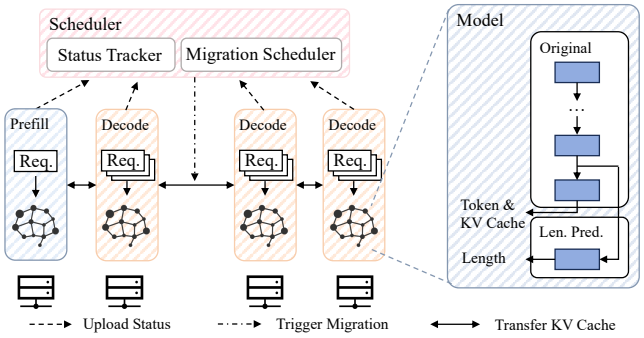


Figure 5: System overview.

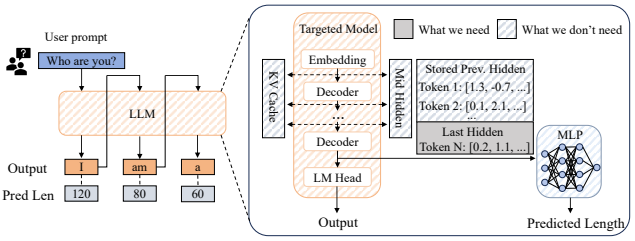


Figure 6: Our runtime prediction method: the MLP predictor consumes the hidden state vector of the last token from the final layer to estimate remaining output length.

decode phase, each request will be continuously predicted for its remaining output length at regular intervals, and the status of each decode instance will be reported to the scheduler. The scheduler will run the rescheduling algorithm to make migration decisions and trigger the migration of requests between decode instances. Particularly, STAR incorporates two novel components:

Length predictor (Section 4) is used to forecast the model's output length, providing a reference for future status in scheduling algorithms. Instead of using the suboptimal approaches discussed in Section 2.3, we leverage the model's internal state by feeding the hidden state of the last token from the final transformer layer into a lightweight MLP predictor. Moreover, we continuously predict the remaining output length of each request at regular intervals during the decode phase to further improve prediction accuracy.

Decode rescheduler (Section 5) not only plays a role in task distribution but also handles the rescheduling of decode tasks. To achieve this functionality, we introduce a multi-stage rescheduling approach. The process begins by evaluating both current and predicted workloads received from instances to identify instances that are either overloaded or underloaded. Next, we assess each instance pair and filter out requests that cannot benefit from migration. Finally, migration scenarios are simulated, and the best feasible migration is selected based on its ability to reduce workload variability. This rescheduling strategy is executed periodically, facilitating improved SLO adherence and minimizing the risk of OOM errors, ultimately enhancing system efficiency.

4 Generation Length Prediction

The generation length predictor is used to provide a reference for future status in decode rescheduling decisions to select requests for migration. Our analysis in Section2.3 reveals existing prediction methods suffer from a fundamental accuracy ceiling because they rely on auxiliary models (e.g., bert-base-uncased, opt-125m) significantly less capable than target LLMs. As Sequence Scheduling [38] established, prediction accuracy correlates with model capability.

4.1 Overview of Predictor

Contrary to prior works that perform generation length prediction only once after prefill, the prediction task in decode rescheduling presents following unique characteristics.

Tight overhead constraints: In contrast to the loose TTFT constraints in prefill (e.g., 4s for Chatbot OPT-175B in DistServe [39]), decode phase has stringent TPOT requirements (e.g., 0.2s for the same model). Heavy auxiliary models or prompt modifications that reprocess the full context are therefore impractical. As sequence length l grows, such methods incur an extra $O(l)$ cost compared to the target LLM. Unfortunately, as shown in Section 2.2, longer sequences are common in real-world scenarios and is the key reason for developing STAR.

Additional context from generated tokens: Rescheduling occurs during decode, after some output tokens have already been generated. These tokens provide additional context that can be leveraged to improve prediction accuracy. Existing methods are mismatched to this setting because they predict only once before decode and use only the prompt.

Figure 6 illustrates our proposed prediction method, which considers the above characteristics throughout two key design aspects: 1) leveraging the target LLM's own internal state, i.e., the hidden state of the last token from the final transformer layer, to achieve both low overhead and high accuracy (Section 4.2); and 2) continuously refining predictions according to the newly generated tokens during decode, thereby enhancing accuracy over time (Section 4.3).

4.2 Lightweight Target-LLM-Native Prediction

Prediction Formulation. The prediction task can be formulated as a regression task. Let ϑ be the parameters of the target LLM model, given an input sequence S including prompt and generated tokens, we aim to learn a prediction function f_θ to estimate the remaining output length,

$$f_\theta : (S) \mapsto p_\theta(Y \mid S), \tag{1}$$

where Y is the remaining output length and θ represents the model parameters of length predictor.

We observe that

> *Length predictor θ can leverage the partial knowledge of target LLM ϑ.*

Instead of using an auxiliary model, we propose a self-aware prediction method that leverages the target LLM's own hidden states to predict remaining output length.

Design Choices. However, there are several design choices to consider when selecting the internal state as input to the prediction model, we discuss the design space in detail below.

- *KV cache vs. Hidden states:* Both KV cache and hidden states are part of the model's internal state. While their size and structure are similar, the information they carry differs: a token's KV cache contains only the information specific to that token, while a layer's hidden state encapsulates information from both the current token and other tokens. As a result, we believe the hidden state carries more informative content than the KV cache. Therefore, we choose the hidden state as the input for our model, a choice also seen in EAGLE-style methods, although those works use hidden states for speculative decoding rather than remaining-length prediction [21–23].
- *All layers vs. last layer:* While we could choose to use the hidden state from all layers or just the last layer as the model input, aggregating hidden states from all layers increases both input dimensionality and computational cost. Since the last layer's hidden state already integrates information from all previous layers and captures the most abstract semantic features, using only the last layer strikes a good balance between information richness and efficiency.
- *All tokens vs. last token:* Using all token states would provide more information, but it makes the input grow with sequence length and would require storing or recomputing preceding states. In contrast, the last token's state is already available at each decode step and, after attending to the full sequence, still captures the overall context. We therefore use only the last token's state as an efficient approximation.

Our Solution: LLM-native Predictor. As demonstrated in Figure 6, our solution employs the hidden state of the last token from the last transformer layer as input to a lightweight MLP predictor. This design ensures a fixed-size input that captures the model's full contextual understanding without incurring the high cost of storing every token's hidden state or processing larger states like the full KV cache.

MLP Predictor. Formally, the predictor receives the last-layer hidden state of the last generated token as a fixed-size vector $h \in \mathbb{R}^d$ and outputs a scalar representing the remaining length through a 4-layer MLP:

$$\hat{y} = w_4\phi(\mathbf{W}_3\phi(\mathbf{W}_2\phi(\mathbf{W}_1 h))), \tag{2}$$

where $\mathbf{W}_1 \in \mathbb{R}^{m_1\times d}$, $\mathbf{W}_2 \in \mathbb{R}^{m_2\times m_1}$, $\mathbf{W}_3 \in \mathbb{R}^{m_3\times m_2}$, $w_4 \in \mathbb{R}^{1\times m_3}$ are learnable parameters, and ϕ is an element-wise nonlinearity (ReLU). In the evaluated DeepSeek-R1-Distill-Qwen-7B model with $d = 3584$, we set the intermediate dimensions to $m_1 = 2048$, $m_2 = 512$, and $m_3 = 64$.

4.3 Enhancing Precision with Continuous Prediction

Though our LLM-native prediction method achieves high accuracy compared to prior approaches, the absolute MAE of 3,873 tokens remains significant for long-output requests. It is worth noting that prediction based solely on the prompt cannot achieve high accuracy, as the same input prompt can lead to highly variable output lengths due to the inherent randomness in LLM generation, including sampling strategies [15], temperature settings [19], and system non-determinism [14]. To further enhance prediction precision, we propose to continuously predict the remaining output length with the integration of additional context from generated tokens. This design is arised from the following observation:

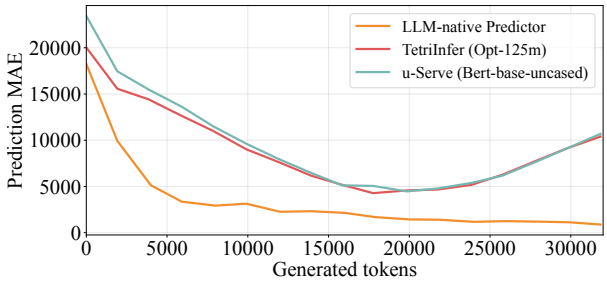


Figure 7: MAE of each prediction model for requests with 30-32K output tokens at different generated tokens.

> *Additional context from generated tokens can be employed to improve prediction accuracy.*

These additional contexts are particularly valuable for two reasons: 1) they provide direct evidence of the model's current trajectory, allowing the predictor to adjust its estimate based on the actual content being generated; 2) richer context enables the predictor to capture nuanced patterns and dependencies that may not be evident from the prompt alone, leading to more informed and accurate predictions.

Empirical Evidence. Figure 7 shows the prediction errors for requests that actually generate 30-32K tokens, measured at different generation stages (i.e., when different numbers of tokens have been generated). The detailed experimental setup is described in Section 4.4. With the increase of generated tokens, the amount of information fed into the prediction model increases, leading to precision of the prediction increasing. Specifically, with only input prompt, the MAE of our method is 18256 tokens, while after generating 8000 tokens, the MAE drops to 2929 tokens.

We notice that the existing auxiliary model methods, i.e., opt and bert, present an increasing trend of MAE as the number of generated tokens increases from 20000 to 30000. This is because these methods only support limited input lengths (1024 tokens for opt and 512 tokens for bert), thereby truncating long inputs, which results in a loss of information and a significant decrease in precision.

4.4 Implementation and Empirical Evaluation

Dataset Construction. We construct a supervised dataset by running the vLLM engine on ShareGPT requests with the DeepSeek-R1-Distill-Qwen-7B model. For each request, we record the last-layer, last-token hidden state $\mathbf{h}_t$ together with the ground-truth remaining length y_t at fixed decode intervals (e.g., every 20 tokens). This yields 100k samples $\mathcal{D} = \{(\mathbf{h}_t, y_t)\}$ across the generation trajectory of each request. To ensure proper evaluation, we split the data at the request level rather than the sample level. Specifically, we randomly partition the original ShareGPT requests into training (70%), validation (15%), and test (15%) sets. This ensures that samples from

Table 1: Comparison of various generation length prediction methods.

Methods	PiA	μ-Serve	TetriInfer	LLM-native
Parameters	7 B	110 M	125 M	8.4 M
Training time	0	$\tilde{1}$ day	$\tilde{1}$ day	$\tilde{1}$ hour
Average MAE	14169.3	8165.8	7658.1	3873.2
Latency (batch:1)	2.2 s	6.0 ms	10.3 ms	1.33 ms
Latency (batch:10)	15.3 s	30.0 ms	65.3 ms	2.4 ms

the same request (at different generation timesteps) never appear in different splits, preventing data leakage and ensuring that the model is evaluated on truly unseen requests.

Training Details. Both our LLM-native predictor and auxiliary models, i.e., μ-Serve (bert-base-uncased) [29] and TetriInfer (opt-125m) [16], are fine-tuned on the same training data on a single H800 GPU for up to 100 epochs (with early stopping patience of 10), minimizing a robust regression L1 loss using the AdamW optimizer [24] and employing early stopping [27] based on validation MAE. Regarding MLP in our LLM-native predictor, we perform hyperparameter search over network depth (2-6 layers), hidden dimensions (256-1024), learning rate (1e-4 to 1e-3), and batch size (16-128), selecting the configuration with the best validation MAE.

Result. Table 1 summarizes the prediction accuracy and overhead of different methods, including 1) PiA [38], a prompt-based method that modifies user instructions to have the LLM first predict its output length before generating the response; 2) μ-Serve [29], which uses bert-base-uncased as an auxiliary model; 3) TetriInfer [16], which uses opt-125m as an auxiliary model and 4) our LLM-native predictor. Regarding the training time, our LLM-native predictor presents a significant advantage compared to auxiliary models, while prompt-based methods PiA is training-free. Therefore, though prompt-based methods have large model size, lack of fine-tuning makes them poor in precision compared to trained models, while our LLM-native predictor achieves the best MAE. Regarding the inference latency, our LLM-native predictor is significantly smaller than auxiliary models and original LLM used in prompt-based methods, leading to the lowest latency.

5 Decode Rescheduling

With the help of LLM-native predictor described in Section 4, we can model the future state of the system and proactively schedule migrations between decode instances. This enables us to redistribute requests in a way that balances resource usage and minimizes latency spikes, especially under heavy or skewed workloads.

5.1 From Prefill-to-Decode Scheduling to Decode-to-Decode Rescheduling

In traditional prefill to decode scheduling, once a request finishes prefill, the scheduler assigns it to a decode instance based on system state and request characteristics at that moment. However, the rescheduling in decode phase presents distinct decision-making space as follows.

Whether to migrate: Decode rescheduling makes migration optional. The scheduler can skip migration when the system is already balanced, when the benefit is smaller than the migration overhead, or when the overload is likely to disappear soon.

Which request to migrate out: Decode rescheduling must explicitly choose a request to move out of an overloaded instance. This is a tradeoff: long requests relieve more load but incur larger KV-cache transfer and may overload the target, while short requests are cheaper to move but may not reduce enough load, especially if they are close to completion.

Which instance to migrate in: The scheduler must also choose the destination instance for the migrated request. This decision depends on the load and available resources of candidate decode instances, together with the demand of the migrated request.

However, recognizing the evolving workload imbalance during decode phase, the scheduler needs to further consider the *future state* of the system when making migration decisions. Here, the future state is derived from the predicted remaining lengths of currently active decode requests only, rather than from future arrivals or future requests for the same query.

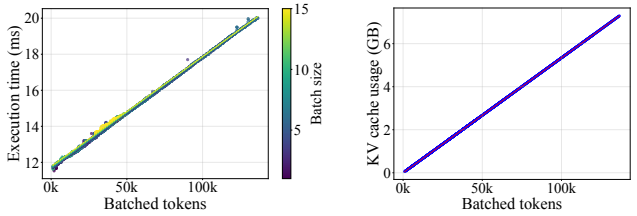
5.2 Migration Algorithm

To address workload imbalance described in Section 2.2 to ensure SLO compliance in decode instances, we design a migration algorithm that leverages execution time modeling and runtime prediction to make informed, low-overhead migration decisions. Before delving into the algorithm, we first review the workload characteristics in decode instances.

Aligning Execution Time and Token load. Recently, researchers have conducted extensive studies on modeling the decode execution time of LLMs [6], either through analytical models or empirical benchmarking. Despite the complexity of these models, one critical impact factor is the number of tokens in the running batch, which directly affects the time to read the KV cache during attention computation. As shown in Figure 8, the decode execution time per iteration is linearly correlated with the number of batched tokens (left panel). The key reason is that the attention computation time is dominated by the KV cache read time [37], which grows linearly with the number of tokens in the batch. Interestingly, another important factor, memory usage, is also linearly correlated with the number of tokens in the batch, as each token contributes a fixed-size KV cache (right panel). Therefore, in this paper, we employ the *number of tokens* in the running batch to unify the modeling of both execution time and memory usage, i.e., workload, simplifying our migration algorithm design.

Problem Formulation. Let $\mathcal{I}$ denote the set of decode instances, and for each instance $i \in \mathcal{I}$, let B_i denote the set of requests assigned to instance i. For a request $r \in B_i$, $N(r)$ denotes the current number of tokens in request r, and $\hat{N}(r)$ denotes the predicted remaining generation length for request r. The current token load of instance i is $N_i(B_i) = \sum_{r \in B_i} N(r)$.

We notice that for scheduling purposes of workload balancing, the absolute execution time is less important than the relative differences between instances. Therefore, our migration objective is to minimize the variance of token loads across all instances, which



(a) Decode execution time. (b) KV cache usage.

Figure 8: Relationship between cost metrics and number of batched tokens.

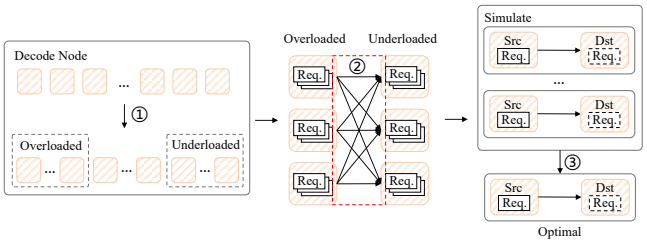


Figure 9: Workflow of the scheduler.

directly correlates with balancing execution time and memory usage. Variance is practical here because it supports efficient online updates and places a stronger penalty on cross-instance imbalance.

The current variance of token loads across all instances is:

$$\sigma_0^2 = \mathrm{Var}(\{N_i(B_i)\}) \tag{3}$$

To capture the dynamic nature of workloads where requests continuously generate new tokens or complete, we formulate our objective as minimizing the expected variance over a prediction horizon. Let $\hat{N}_i(B_{i,t})$ denote the predicted token load of instance i at future time step t, computed using our generation length predictor. The expected variance combines both current and predicted states:

$$\hat{\sigma}^2 = \sigma_0^2 + \sum_{t=1}^{\infty} \beta_t \cdot \mathrm{Var}(\{\hat{N}_i(B_{i,t})\}), \tag{4}$$

where β_t is a time-dependent weighting factor that balances the importance of current state versus predicted future states at time step t. This formulation ensures that migration decisions consider both immediate and long-term load balancing effects.

Algorithm Design. The algorithm consists of two main components: *scheduler-side* and *worker-side* functions. The scheduler periodically collects pre-simulated state reports from all workers, classifies instances, enumerates candidates, and completes the remaining simulation to select the best feasible migration. Workers continuously monitor their local state, retrieve prediction results, perform local future state simulation, and proactively report this pre-computed information to the scheduler. STAR is a periodic online heuristic that balances execution imbalance, memory safety, and migration overhead.

Figure 9 illustrates scheduler side workflow for our decode rescheduling architecture, which operates through three distinct phases implemented in Algorithm 1:

- **Phase 1: Instance Classification (Lines 11-16).** The scheduler identifies overloaded and underloaded instances by computing the weighted workload w_i for each instance and comparing it against the average workload $\bar{w}$. This classification, implemented in Lines 14-16, focuses the algorithm on a small subset of instances that require attention.
- **Phase 2: Candidate Enumeration (Lines 17-23).** For each source-target instance pair, the algorithm enumerates migration candidates by filtering requests that satisfy two constraints: (1) sufficient remaining tokens exceeding migration overhead (Line 20), and (2) memory safety ensuring no OOM on target instance in the near future (Line 21).
- **Phase 3: Best-Feasible Selection (Lines 24-34).** The algorithm evaluates each candidate by aggregating pre-computed worker simulations (Line 29) and computing the incremental variance reduction (Line 30). When prediction is enabled, the algorithm simulates future variance based on predicted token loads; otherwise, it evaluates candidates based on the current variance. The best feasible migration is selected as the one that maximizes variance reduction, leveraging the fact that workers have already performed local simulations to minimize scheduler computation overhead. Choose the migration that yields the greatest reduction in time-weighted token load variance and then execute migration.

Complexity Analysis. Let $n = |\mathcal{I}|$ be the number of decode instances, let $\mathcal{O}$ and $\mathcal{U}$ be the overloaded and underloaded instance sets, let $R_{\max}$ be the maximum number of active requests on any instance, and let H be the prediction horizon. In the native implementation of Algorithm 1, instance classification costs $O(n)$, and candidate enumeration costs $O(|\mathcal{O}| \cdot |\mathcal{U}| \cdot R_{\max})$. For each source-target pair, there are at most $R_{\max}$ candidate requests, and evaluating one candidate requires recomputing an H-step token-load trace over up to $R_{\max}$ requests, which costs $O(R_{\max}H)$. Therefore, the native scheduler complexity per interval is $O(n + |\mathcal{O}| \cdot |\mathcal{U}| \cdot R_{\max}^2 \cdot H)$. This cost can be reduced with worker-side pre-aggregation and scheduler-side incremental updates. Each worker first computes its H-step future-load summary once in $O(R_{\max}H)$ time. Then each candidate evaluation only updates the source and target summaries, reducing the cost to $O(H)$. The optimized scheduler complexity becomes $O(n + |\mathcal{O}| \cdot |\mathcal{U}| \cdot R_{\max} \cdot H)$. In Section 6.3, these computations remain below 300 ms even for 256 instances and can overlap with decode computation.

5.3 Trade-off between Frequency and Overhead

We notice that frequent reprediction and rescheduling, e.g., at every decode iteration, can improve the precision of the predicted remaining generation length as well as the workload balancing effectiveness. However, this comes at the cost of increased computational overhead incurred by the prediction model.

Specifically, consider the model DeepSeek-R1-Distill-Qwen-7B tested on RTX 4090D, our prediction model takes 1.40 ms to infer a batch of 10 requests, while each decode iteration takes about 18.23 ms under 50% KV cache memory occupancy. Therefore, injecting prediction at every decode iteration would incur a prohibitive

Algorithm 1 Decode Rescheduling

Scheduler loop
while serving requests **do**
  Wait for the next scheduling interval
  Collect worker states and update the average workload $\bar{w}$
  $(\mathcal{O}, \mathcal{U}) \leftarrow$ InstanceClassification$(\bar{w})$
  if $\mathcal{O} \neq \emptyset$ **then**
    $\mathcal{C} \leftarrow$ CandidateEnumeration$(\mathcal{O}, \mathcal{U}, \mathcal{S})$
    $m^* \leftarrow$ BestFeasibleSelection$(\mathcal{C}, \sigma_0^2)$
    if $m^* \neq \emptyset$ **then**
      ExecuteMigration(m^*)
Function InstanceClassification$(\bar{w})$
for each instance $i \in \mathcal{I}$ **do**
  $w_i = \sum_{t=1}^{H} \beta_t \cdot \hat{N}_i(B_{i,t})$
$\mathcal{O} \leftarrow \{i \in \mathcal{I} \mid w_i > (1+\theta) \cdot \bar{w}\}$
$\mathcal{U} \leftarrow \{i \in \mathcal{I} \mid N_i(B_{i,0}) < (1+\theta) \cdot \bar{w}\}$
return $(\mathcal{O}, \mathcal{U})$
Function CandidateEnumeration$(\mathcal{O}, \mathcal{U}, \mathcal{S})$
$\mathcal{C} \leftarrow \emptyset$
for each $(s,t) \in \mathcal{O} \times \mathcal{U}$ **do**
  $\mathcal{R}_s \leftarrow \{r \in B_s \mid \hat{N}(r) > \frac{C_{\text{mig}}}{\bar{t}_{\text{exec}}}\}$
  $\mathcal{R}_s \leftarrow \mathcal{R}_s \cap \{r \mid N_t(B_{t,0}) + \hat{N}(r) \leq C_{\text{mem}}\}$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{(r,s,t) \mid r \in \mathcal{R}_s\}$
return $\mathcal{C}$
Function BestFeasibleSelection$(\mathcal{C}, \sigma_0^2)$
$m^* \leftarrow$ null
$\sigma^2_{\max} \leftarrow 0$
for each candidate $(r,s,t) \in \mathcal{C}$ **do**
  if usePrediction **then**
    $\hat{\mathcal{S}} \leftarrow$ AggregateSimulations$(\mathcal{S}, r, s, t)$
    $\sigma^2 \leftarrow$ SimulateFutureVariance$(\hat{\mathcal{S}}, H)$
  else
    $\sigma^2 \leftarrow$ CurrentVariance$(\mathcal{S}, r, s, t)$
  $m^* \leftarrow$ UpdateBestCandidate$(m^*, \sigma^2, \sigma^2_{\max})$
return m^*

overhead of 7.68%, which is unacceptable in practice. Formally, setting prediction interval to every k decode iterations results in a prediction overhead of $\frac{1.40}{18.23 \times k}$. Regarding workload characteristics, k iterations at most increases the workload by $\frac{k}{l}$, where l represents the average length of requests in decode instances (e.g., 2000 tokens in our settings). Considering both prediction accuracy and computational overhead, we can set k to 20 as the prediction interval, which only incurs a low overhead of 0.38% while maintaining high prediction accuracy with less than 1% error.

5.4 Overlap Migration with Decode Computation

We notice that the migration of decode requests inevitably incurs overhead due to KV cache transfer, which can overlap with decode computation to minimize its impact on overall system performance. Specifically, migration process leverages NVIDIA Inference Transfer Library for asynchronous KV cache transfer. The paused request's KV cache is transferred to the target instance without blocking the execution of other requests in the same batch. This asynchronous design ensures that migration overhead does not impact the performance of non-migrating requests. To ensure seamless user experience, we implement a proxy-based architecture where users establish connections with proxy that is decoupled from the processing instances. The proxy maintains a persistent stream connection with the client, enabling continuous response delivery during request migration, ensuring that users remain unaware of migration events.

Table 2: Workload Statistics

Workload	Metric	Mean	Std	P50	P90	P95
ShareGPT	Input	305	1053	36	920	1609
	Output	7542	12008	1536	32670	32679
Alpaca	Input	11	4	10	15	18
	Output	8596	13354	987	32690	32691

6 Evaluation

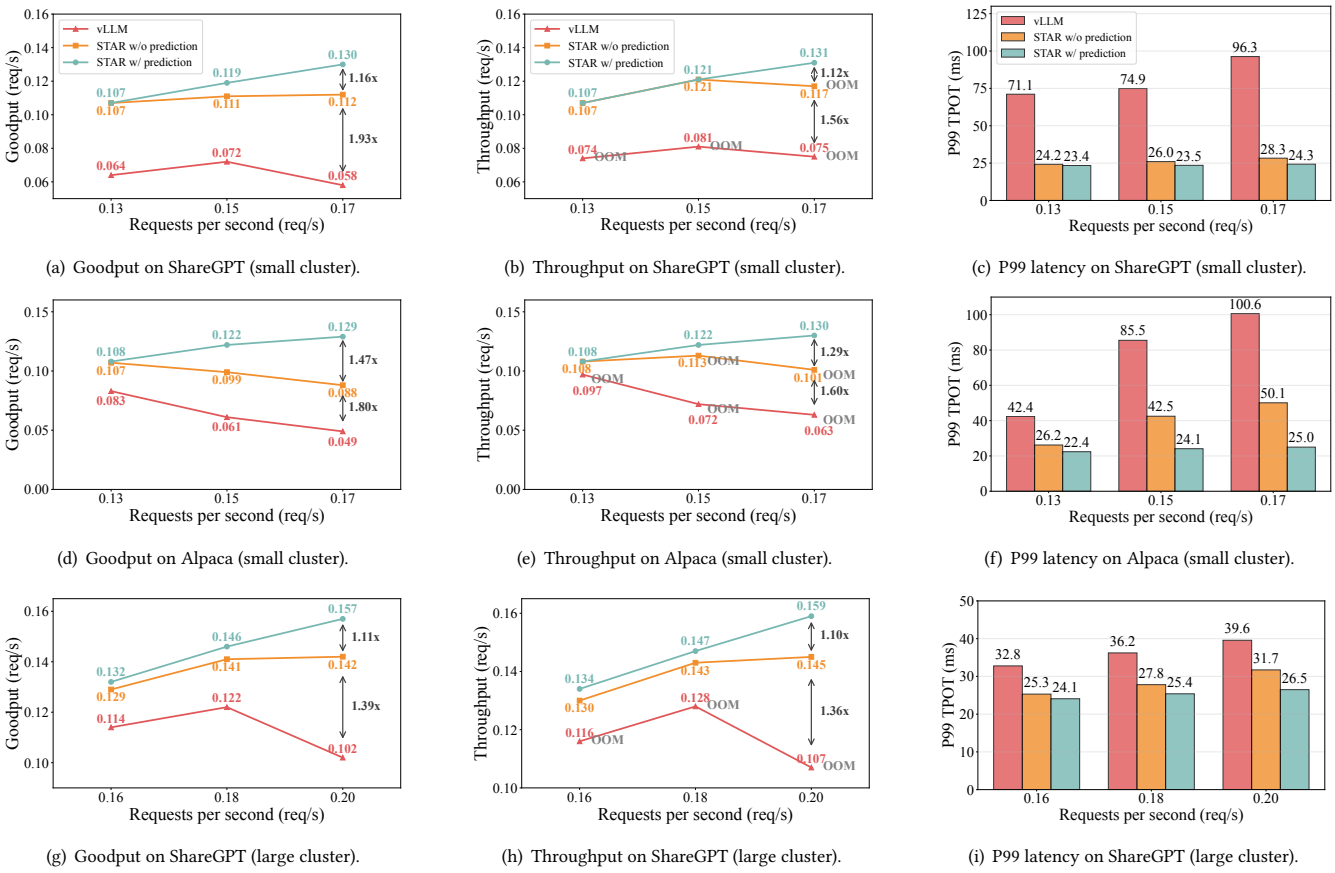
6.1 Experimental Setup

Implementation. We implement STAR in approximately 6,000 lines of code (LoC) on top of vLLM 0.9.2, which supports PD disaggregation with the official NIXL (NVIDIA Inference Transfer Library) component for KV cache transfers. STAR includes three main components: 1) the generation length predictor (Section 4), 2) the rescheduling algorithm (Section 5), and 3) the migration proxy that orchestrates migrations between decode instances. For the predictor, we implement a custom MLP architecture and develop training and evaluation scripts that integrate seamlessly into the vLLM engine. The rescheduling algorithm is implemented as a standalone module that monitors decode instances and executes migration decision process. The migration proxy extends the vLLM engine with decode-to-decode KV cache transfer capabilities and runs rescheduling algorithm. Additionally, we develop a dedicated simulator to validate the theoretical effectiveness of our approach in large-scale systems, which will be detailed in Section 6.3.

Environment and models. We validate STAR across multiple hardware configurations as below:

- small cluster: A server equipped with 4 NVIDIA RTX 4090D GPUs (1 for prefill and 3 for decode), a 40-core Intel Xeon Gold 5418Y CPU and 120GB memory, running Ubuntu 22.04.4 LTS and CUDA 12.2.
- large cluster: A server equipped with 8 NVIDIA H800 GPUs (2 for prefill and 6 for decode), a 40-core Intel Xeon Platinum 8458P CPU and 400GB memory, running Ubuntu 22.04.4 LTS and CUDA 12.8.

In addition, we also develop a simulation framework to evaluate larger scale clusters as detailed in Section 6.3. In order to verify the effect under long output requests, we use the reasoning model DeepSeek-R1-Distill-Qwen-7B [8] and Qwen3-14B [36] as base model and adopt W8A8 quantization (only in small cluster) to support longer output tokens [10], which supports up to 32K tokens for both prompt and output.



(a) Goodput on ShareGPT (small cluster).
(b) Throughput on ShareGPT (small cluster).
(c) P99 latency on ShareGPT (small cluster).
(d) Goodput on Alpaca (small cluster).
(e) Throughput on Alpaca (small cluster).
(f) P99 latency on Alpaca (small cluster).
(g) Goodput on ShareGPT (large cluster).
(h) Throughput on ShareGPT (large cluster).
(i) P99 latency on ShareGPT (large cluster).

Figure 10: Overall performance on ShareGPT and Alpaca datasets.

Workload. We utilize two widely adopted real-world datasets, ShareGPT [33] and Alpaca [32], to evaluate the performance of STAR. By default, we present results using the ShareGPT dataset. The prompt and generation length distribution of the dataset using DeepSeek-R1-Distill-Qwen-7B are shown in Table 2, where P50, P90, and P95 represent the 50th, 90th, and 95th percentiles of the length distribution starting from the shortest request. Notably, over 15% of requests generate over 25K tokens, highlighting the challenge of load balancing for decode instances.

Baselines. We evaluate the following migration strategies with *real-time KV cache load balancing* for the prefill-to-decode transfer (serving as the common foundation):

- *vLLM [18]*: We employ vLLM's built-in prefill-decode disaggregation architecture as the baseline, which adopts the idea from DistServe [39]. As the industry-standard serving framework, *vLLM* already supports PD disaggregation.
- *STAR w/o prediction*: We implement a rescheduling algorithm without prediction upon the vLLM framework, which periodically evaluates the workload of decode instances and migrates requests from overloaded to underloaded instances based on current state only.
- *STAR w/ prediction*: We integrate the generation length predictor and the decode rescheduling algorithm.
- *STAR Oracle*: An oracle version of *STAR w/ prediction* that assumes exact remaining generation lengths for all active requests, providing an upper bound on prediction-aware rescheduling.

6.2 End-to-End Performance

Figure 10 illustrates the three critical metrics: throughput, and goodput, and tail latency, across different scheduling strategies under various requests per second (RPS). Subsequently, we analyze each metric in detail.

Throughput. As shown in Figure 10(b), Figure 10(e) and Figure 10(h), both rescheduling and prediction improve throughput, with the largest gain under high load. On large cluster at 0.20 req/s, rescheduling raises throughput from 0.107 to 0.145 req/s (35.5%), and prediction further raises it to 0.159 req/s (an additional 9.7%). This gain comes from alleviating decode-instance saturation and OOM caused by workload imbalance.

Goodput. We use goodput to measure the requests that satisfy the SLO. The SLO is 1s TTFT, with TPOT targets of 25 ms for DeepSeek-R1-Distill-Qwen-7B and 50 ms for Qwen3-14B. Because workload

imbalance directly increases tail latency and SLO violations, goodput shows the benefit of rescheduling and prediction more clearly than throughput. Figure 10(g) shows an even larger gain in goodput. On large cluster at 0.20 req/s, rescheduling improves goodput from 0.102 to 0.142 req/s (39.2%), and prediction further improves it to 0.157 req/s (an additional 10.6%). This indicates fewer SLO violations under better-balanced decode load.

Latency. We report P99 TPOT to capture tail latency. Figure 10(c) and Figure 10(f) show large reductions across both datasets. On ShareGPT at RPS=0.17, rescheduling reduces P99 TPOT from 96.3 ms to 28.3 ms, and prediction further reduces it to 24.3 ms. Figure 10(i) shows a similar trend. At 0.20 req/s, rescheduling reduces P99 TPOT from 39.57 ms to 31.72 ms, and prediction further reduces it to 26.49 ms. This reduction comes from preventing heavily overloaded decode instances and the resulting tail-latency inflation.

6.3 Scalability and Load-Balance Analysis

To demonstrate the performance upper bound of our prediction-based rescheduling approach, we introduce an oracle version *STAR Oracle*, which assumes perfect knowledge of the generation lengths for all requests. Subsequently, we evaluate the effectiveness of our approach in both small-scale real systems and large-scale simulated clusters by demonstrating the variation in execution time across decode instances for 2,000 seconds on shareGPT dataset.

Execution on small scale. Figure 11 illustrates the execution time variance across different scheduling strategies when running on 3 decode instances in small cluster. Our proposed prediction solution achieved an average execution time variance of 0.78 ms², which is close to that of oracle prediction. *vLLM shows* bursty variance under workload imbalance, rescheduling mitigates it, and prediction reduces it further.

Simulation on large scale. We conduct simulations using our dedicated simulator that models large-scale cluster behaviors across hundreds of instances in production scale deployments. The simulator follows the same scheduling and migration logic as the real system. Specifically, we use event-driven simulation to model request arrivals, decode execution, and migration events. The execution time of each decode iteration is derived from real system measurements, while the migration overhead is calculated based on KV cache size and network bandwidth. Regarding prediction, we leverage the actual remaining generation lengths to simulate an oracle predictor. We set the request rate to 0.3 RPS for an 8-instance cluster and scale it linearly with cluster size. This configuration ensures that the system reaches a dynamic equilibrium where request arrival and completion rates are balanced, preventing unbounded queue growth while maintaining sufficient workload. Using ShareGPT and a 25 Gbps transfer speed (following DistServe's cross-node interconnection setting [39]), we evaluate cluster sizes from 8 to 256 instances. Figure 13 shows that rescheduling improves load balance. *STAR w/ prediction* achieves load balancing close to the oracle, and this advantage persists as the cluster scales up.

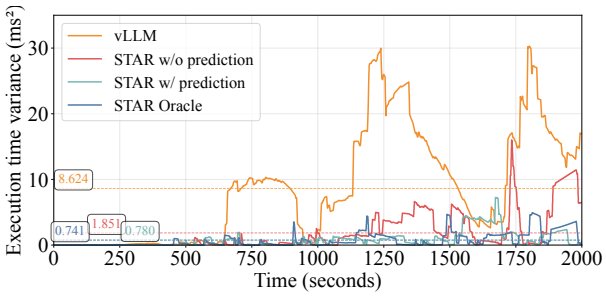


Figure 11: Execution time variance across different scheduling algorithms on high-load dataset.

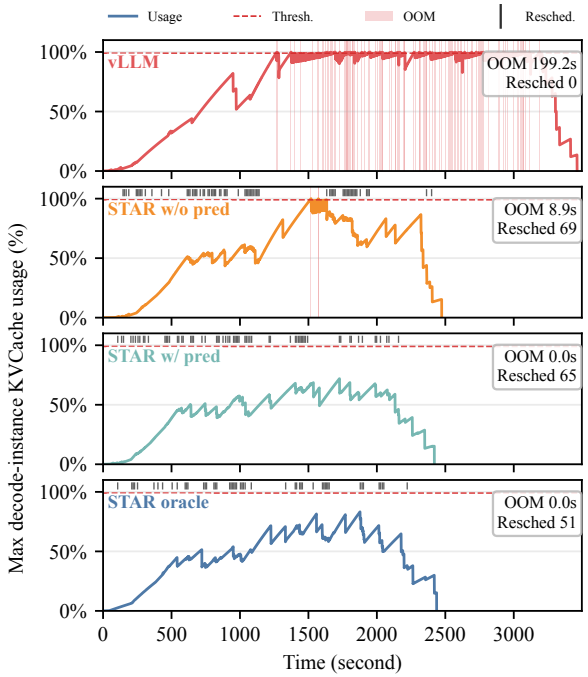
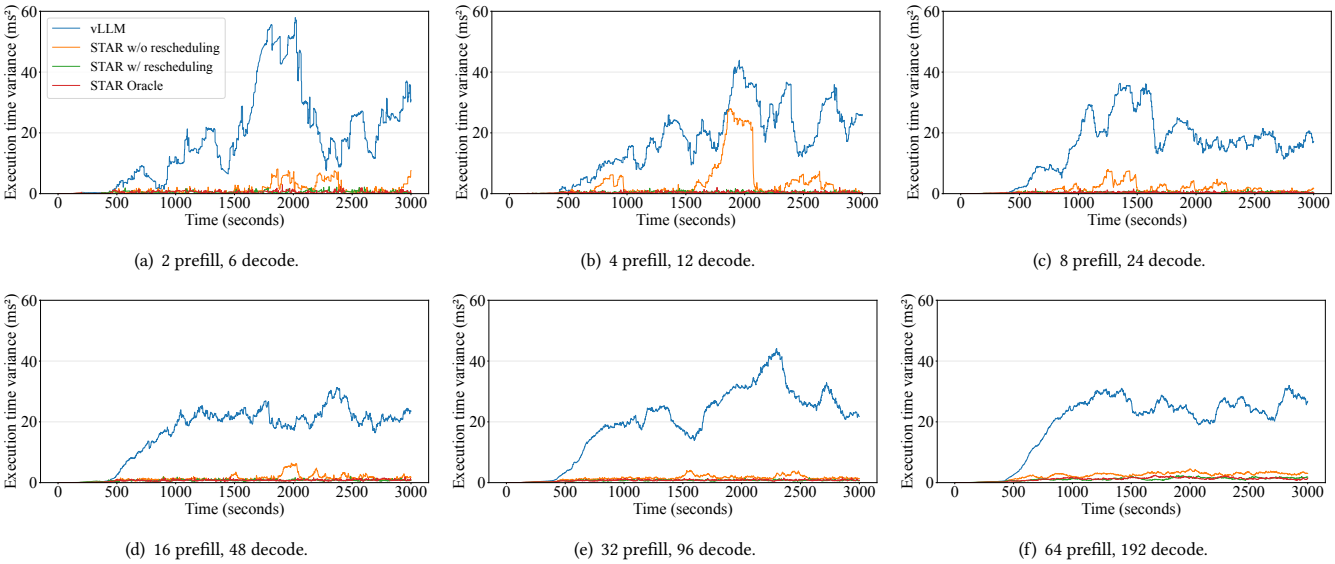


Figure 12: Runtime traces on the small cluster. The curve shows the maximum decode-instance KVCache usage; the dashed line marks the 99% threshold; shaded regions indicate when OOM occurs; vertical ticks denote rescheduling events.

6.4 Runtime Trace Analysis

Figure 12 presents runtime traces extracted from the same small-cluster experiment as the execution-on-small-scale study in Section 6.3. *vLLM* remains near saturation for extended periods and repeatedly experiences OOM. *STAR w/o prediction* substantially reduces OOM occurrences, whereas *STAR w/ prediction* and *STAR Oracle* stay below the 99% threshold throughout the trace. These traces show that STAR improves goodput and P99 TPOT by better balancing decode load. Rescheduling suppresses severe imbalance, while prediction enables more forward-looking migrations and achieves a better balance with fewer unnecessary migrations.



(a) 2 prefill, 6 decode. (b) 4 prefill, 12 decode. (c) 8 prefill, 24 decode.

(d) 16 prefill, 48 decode. (e) 32 prefill, 96 decode. (f) 64 prefill, 192 decode.

Figure 13: Execution time variance comparison across different cluster sizes under 25 Gbps transfer speed.

Table 3: Prediction accuracy sensitivity on the large cluster.

Setting	Exec. Var.	P99 TPOT	Goodput	Goodput Gain
Full	0.163	26.49	0.157	10.56%
6-bin	0.188	26.91	0.155	9.15%
4-bin	0.220	27.70	0.148	4.23%
2-bin	0.302	31.47	0.142	0.00%
No pred.	0.322	31.72	0.142	0.00%

Table 4: Prediction-interval tradeoff on the large cluster.

Interval	Exec. Var.	P99 TPOT	Goodput	Goodput Gain
1 iter	0.237	27.84	0.148	4.23%
20 iter	0.163	26.49	0.157	10.56%
100 iter	0.242	29.43	0.145	2.11%
No pred.	0.322	31.72	0.142	0.00%

6.5 Prediction Robustness Analysis

Prediction Accuracy Sensitivity. We evaluate prediction accuracy sensitivity on the same large cluster setting as Figure 10. Specifically, the 2-bin, 4-bin, and 6-bin settings partition the remaining-length target into [0, 8K), [8K, 32K]; [0, 4K), [4K, 8K), [8K, 16K), [16K, 32K]; and [0, 2K), [2K, 4K), [4K, 6K), [6K, 8K), [8K, 16K), [16K, 32K], respectively. We use these non-uniform bins to align with the scheduler's main decision boundary: whether a request is near completion and unlikely to benefit from migration, or still long enough that migration is worthwhile. Here, *Exec. Var.* denotes the execution-time variance across decode instances, consistent with Section 6.3. Table 3 shows that performance degrades gradually as the prediction granularity becomes coarser. The 6-bin predictor retains most of the benefit of the full predictor, with 0.155 vs. 0.157 goodput, only 0.42 ms higher P99 TPOT, and only slightly higher execution-time variance (0.188 vs. 0.163). In contrast, the 2-bin predictor becomes nearly indistinguishable from *No pred.*, with similarly high execution-time variance (0.302 vs. 0.322), indicating that STAR does not require exact token-count regression but does require sufficient granularity to separate substantially different remaining workloads.

Prediction-Interval Tradeoff. Table 4 reports the effect of the reprediction interval under the same large cluster setting. A moderate interval of 20 decode iterations gives the best overall result. Repredicting every iteration increases prediction overhead and can trigger unnecessary migrations, while predicting every 100 iterations makes the scheduling decisions stale.

7 Conclusion

In this paper, we propose STAR, a decode-phase rescheduling framework, which integrates 1) a lightweight and accurate generation length predictor and 2) a prediction-based rescheduling algorithm to proactively mitigate workload imbalance in decode instances. Evaluations under diverse workloads demonstrate that STAR significantly enhances load balance and achieves up to 2.63× higher goodput compared to existing systems. Furthermore, large-scale system simulations further validate the effectiveness of rescheduling and prediction for load balancing across clusters with up to 256 instances.

Acknowledgments

We sincerely thank the reviewers for their valuable comments and our shepherd Nikhil Jain for the helpful guidance. This work was supported by the National Natural Science Foundation of China under Grant No. 62572225, 62502198, 62325205, 62272215,

and U25B2035; the Natural Science Foundation of Jiangsu Province under Grant No. BK20251224; the Nanjing "U35" Talent Cultivation Program (No. U (2024) 001); and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118). Rong Gu is the corresponding author of this paper.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[2] Alibaba. 2025. Qwen. https://chat.qwen.ai/. Accessed: 2025-08-30.

[3] Anthropic. 2025. Claude. https://claude.ai/. Accessed: 2025-08-30.

[4] Anyscale. 2026. Tune Ray Serve LLM application parameters. https://docs.anyscale.com/llm/serving/parameter-tuning. Accessed: 2026-04-14.

[5] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. 2024. Efficient and Economic Large Language Model Inference with Attention Offloading. arXiv:2405.01814 [cs.LG] https://arxiv.org/abs/2405.01814

[6] Ke Cheng, Wen Hu, Zhi Wang, Hongen Peng, Jianguo Li, and Sheng Zhang. 2025. Slice-Level Scheduling for High Throughput and Load Balanced LLM Serving. arXiv:2406.13511 [cs.DC] https://arxiv.org/abs/2406.13511

[7] DeepSeek-AI. 2025. DeepSeek. https://chat.deepseek.com/. Accessed: 2025-08-30.

[8] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[9] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[10] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG] https://arxiv.org/abs/2210.17323

[11] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. arXiv:2408.15792 [cs.LG] https://arxiv.org/abs/2408.15792

[12] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 111–126. https://www.usenix.org/conference/atc24/presentation/gao-bin-cost

[13] Google-DeepMind. 2025. Gemini 2.5. https://gemini.google.com/app. Accessed: 2025-08-30.

[14] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. *Thinking Machines Lab: Connectionism* (2025). doi:10.64434/tml.20250910 https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[15] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019).

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

[17] 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.

[18] 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.

[19] Lujun Li, Lama Sleem, Niccolo' Gentile, Geoffrey Nichil, and Radu State. 2025. Exploring the Impact of Temperature on Large Language Models:Hot or Cold? arXiv:2506.07295 [cs.CL] https://arxiv.org/abs/2506.07295

[20] Weiqing Li, Guochao Jiang, Xiangyong Ding, Zhangcheng Tao, Chuzhan Hao, Chenfeng Xu, Yuewei Zhang, and Hao Wang. 2025. FlowKV: A Disaggregated Inference Framework with Low-Latency KV Cache Transfer and Load-Aware Scheduling. arXiv:2504.03775 [cs.DC] https://arxiv.org/abs/2504.03775

[21] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees. arXiv:2406.16858 [cs.CL] https://arxiv.org/abs/2406.16858

[22] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test. arXiv:2503.01840 [cs.CL] https://arxiv.org/abs/2503.01840

[23] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2025. EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty. arXiv:2401.15077 [cs.LG] https://arxiv.org/abs/2401.15077

[24] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG] https://arxiv.org/abs/1711.05101

[25] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 2024. SpotServe: Serving Generative Large Language Models on Preemptible Instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1112–1127. doi:10.1145/3620665.3640411

[26] OpenAI. 2025. ChatGPT. https://chat.openai.com. Accessed: 2025-08-30.

[27] Lutz Prechelt. 2002. Early stopping-but when? In *Neural Networks: Tricks of the trade*. Springer, 55–69.

[28] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[29] 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.

[30] 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).

[31] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/system/files/osdi24-sun-biao.pdf

[32] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[33] ShareGPT Teams. 2023. ShareGPT. https://sharegpt.com/. Accessed: 2025.

[34] vLLM Project. 2025. vLLM Disaggregated Prefill. https://docs.vllm.ai/en/latest/examples/online_serving/disaggregated_prefill.html. Accessed: 2025-09-09.

[35] 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.

[36] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[37] 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.

[38] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems* 36 (2023), 65517–65530.

[39] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin