# Adaptive Rescheduling in Prefill-Decode Disaggregated LLM Inference

Zhibin Wang[1], Zetao Hong[1], Xue Li[2], Zibo Wang[1], Shipeng Li[1], Qingkai Meng[1], Qing Wang[1], Chengying Huan[1], Rong Gu[1], Sheng Zhong[1], Chen Tian[1]
[1]*State Key Laboratory for Novel Software Technology, Nanjing University*
[2]*Alibaba Group*

## Abstract

Large Language Model (LLM) inference has emerged as a fundamental paradigm. In real-world scenarios, 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 ARES, an adaptive decoding 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 74.77% and achieving up to 2.24× higher goodput.

## 1 Introduction

With the great success of Large Language Models (LLMs) such as ChatGPT [24], Claude [3], Qwen [2], Gemini [11], and DeepSeek [6], 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 [4, 10, 23] are proposed to improve efficiency and reduce the cost of LLM serving.

The autoregressive inference process of LLMs results in significant workload variations, especially in reasoning tasks that require long chain-of-thought (CoT) outputs [32]. 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 [8], 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.

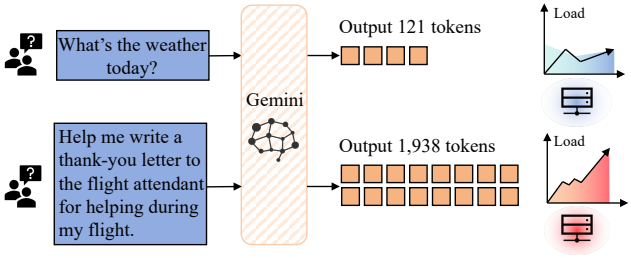



**Figure 1.** Different output lengths lead to significant load variations across decode instances.

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

- **Issue 1: OOM of KV cache.** Recently, to minimize the KV cache memory footprint of possible long outputs, PagedAttention [16] are proposed to allow on-demand KV cache management. However, the evolving KV cache memory footprint may still exceed the hardware memory capacity with the evolving of long-output generation, resulting in "OOM errors". This OOM issue results in recomputation of the KV cache for the affected requests, leading to increased latency and reduced throughput.
- **Issue 2: Violations of SLO.** Considering the distinct characteristics of prefill and decode, modern LLM serving systems [26, 35] 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, 16, 26, 33, 35] are not prepared to handle workload imbalance posed by long output reasoning tasks. 1) Several works [26] 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 [18, 31] further consider the workload-balancing scheduling when *dispatching requests from prefill to decode*: Round-robin scheduling [31] 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 [18] 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.

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 workload, which can be modeled by the remaining generation length of each request. 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 [14, 15, 27], which introduce additional computational overhead and limit by the ability of auxiliary models; or 2) inject tokens into the input prompt [34] 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.** Differing from traditional task scheduling, decode rescheduler must consider not only the current workload of each decode instance but also the predicted future workload based on the remaining generation length of each request. Moreover, instead of only considering the assignment of a single request, the scheduler must further consider which requests to migrate from a overloaded decode instance, and even when to trigger the migration, as if a request is close to completion, migrating it may introduce unnecessary overhead. Therefore, the decision space for decode rescheduling is significantly larger and more complex than traditional prefill-to-decode scheduling, making it challenging to design an effective rescheduling strategy.

In response to the above challenges, we introduce ARES, abbreviated for smart token-length aware rescheduling. As far as we know, ARES is the first system to consider decode rescheduling to resolve workload imbalance across decode instances. ARES 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 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 optimal one 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 ARES significantly outperforms SOTA LLM serving systems. Compared to PD disaggregation implemented in vLLM [16], ARES achieves up to 2.24× higher goodput, reduces P99 TPOT latency by 74.77%, 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

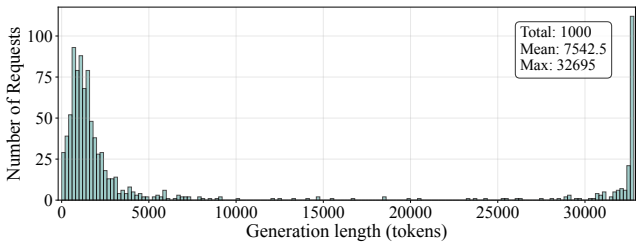



**Figure 2.** Output length distribution.

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 [26], DistServe [35]) 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 [26]. After the prefill phase is completed, 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 [16], Orca [33] to improve hardware utilization.

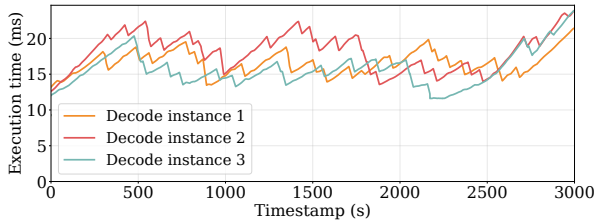
Particularly, taking deployment of DeepSeek-V3 [8] as an example, to fully utilize the hardware resources, a prefill pod leverages 32 H800 GPUs, while a decode pod utilizes 320 GPUs, which highlights that decode is the resource-consuming phase. In the subsequent sections, we will delve deeper into the decode phase.

### 2.2 Imbalance Workload in Decode Phase

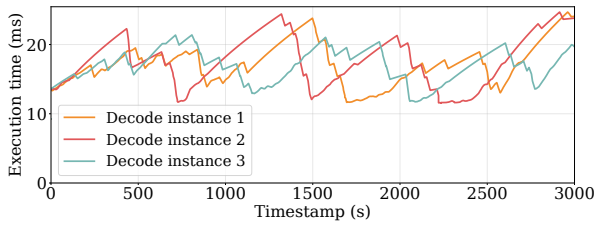
Given the autoregressive nature of decode, the workload of each request is determined by the output sequence length. Therefore, we first analyze the characteristics of output length in real-world scenarios.

**Output length variation.** Figure 2 illustrates the output length distribution in the ShareGPT dataset when running DeepSeek-R1-Distill-Qwen-7B model with a maximum output length of 32K tokens. Many requests present short outputs, with 29.2% requests generating fewer than 1,000 tokens, while a non-ignorable portion of requests produce very long outputs (17.3% with exceeding 30,000 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.




(a) Round-robin scheduling




(b) Current-load balancing

**Figure 3.** Per-step execution time (TPOT) across three decode instances under PD disaggregation (1 prefill + 3 decode instances).

- **Round-robin scheduling [31]:** 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 and suboptimal performance.
- **Current-load balancing [18]:** 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 (e.g., long-output sequences) dominate its resources, causing cascading TPOT spikes for subsequent requests.

**Summarization and Motivation.** Existing prefill-to-decode scheduling strategies can not effectively balance the workload across decode instances, especially in scenarios with long and variable output lengths. Rescheduling during the

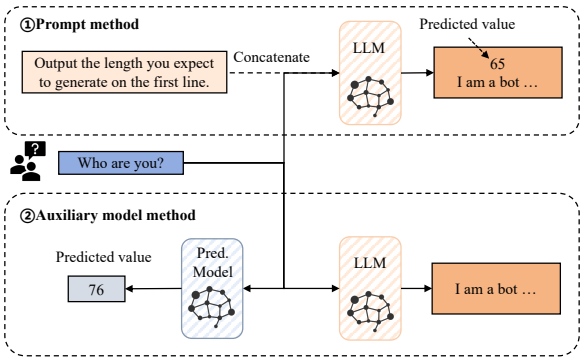



**Figure 4.** Current prediction methods.

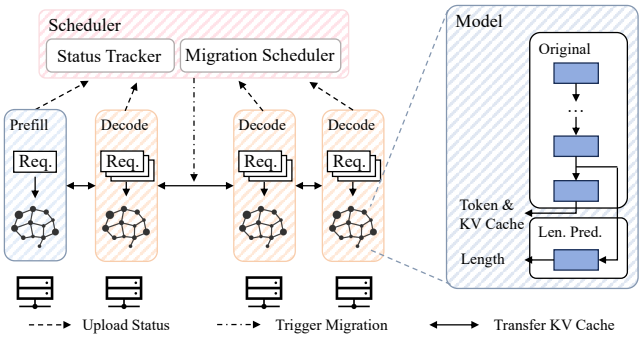



**Figure 5.** System overview.

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) [34] modify user instructions to have the LLM first predict its output length before generating the response (Figure 4). While achieving reasonable accuracy, this approach requires intrusive modifications to user prompts, potentially altering model behavior and output quality. As LLM applications have matured, such user-facing interventions have become increasingly unacceptable in production environments, rendering this approach impractical for modern serving systems.
- **Auxiliary model methods** employing smaller models like opt or bert (Figure 4) with added prediction heads [15, 27, 28]. Sequence Scheduling [34] 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 nontrivial, 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 ARES, which is built upon the PD disaggregation architecture. When a request arrives, it is first sent to a prefill instance to process the input prompt and generate the 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 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, ARES 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 most optimal 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 the best request for migration. Our analysis in Section2.3 reveals that 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 [34] establiszhed, prediction accuracy correlates with model capability.

### 4.1 Overview of Predictor

Contrary to prior works that treat generation length prediction as a one-off task performed only once after the prefill phase, 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 [35]), decode phase has stringent TPOT requirements (e.g., 0.2s for the same model). Therefore, a heavy auxiliary model or intrusive prompt modification requiring processing all preceding tokens is infeasible, as decode phase only need to process the current generating token. With the increasing length of the sequence including prompt and generated tokens denoted as $l$, the overhead of prediction methods increases by a factor of $O(l)$ 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 ARES.

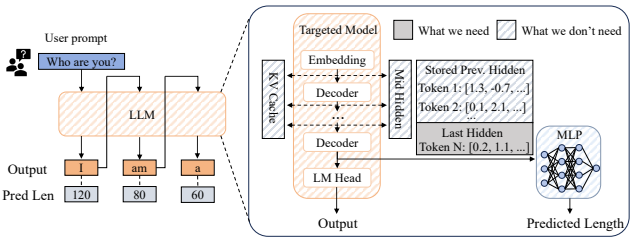



**Figure 6.** Our runtime prediction method: the MLP predictor consumes the hidden state vector of the last token from the final layer to estimate remaining output length.

**Additional context from generated tokens**: As rescheduling occurs during the decode phase, the model has already generated some output tokens. These tokens provide additional context that can be leveraged to improve prediction accuracy. Moreover, as more tokens are generated, the prediction can be refined iteratively, further enhancing accuracy. In contrast, existing prediction methods are only called once before decode, which makes them deviate from decode usage scenarios as 1) they only consider the prompt without any generated tokens, which limits their accuracy, and 2) they only predict once without iterative refinement, which makes their design incompatible with incremental prediction in decode.

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 $\vartheta$ be the parameters of the target LLM model, given a input sequence $S$ including prompt and generated tokens, we aim to learn a prediction function $f_\theta$ 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 $\theta$ represents the model parameters of length predictor.

We observe that

> *Length predictor $\theta$ can leverage the partial knowledge of target LLM $\vartheta$.*

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 other works that leverage the model's internal state for prediction tasks [19–21].
- *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 the states of all tokens provides the most comprehensive information, but the input size grows linearly with sequence length, resulting in increased computational and memory overhead as generation progresses. Moreover, since the internal state of preceding tokens is not cached during the decode phase, we would either need to store them in additional memory or recompute them, both of which incur extra cost. In contrast, the state of the last token—the current generating token—is the focus of the decode phase and is always available. While one might worry that using only the last token's state could lose information from preceding tokens, the last token's state, after attention over the entire sequence, already encapsulates rich semantic information about the full context, making it highly representative for prediction.

**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 $\phi$ 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 can not 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 [13], temperature settings [17], and system nondeterminism [12]. 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:

> *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

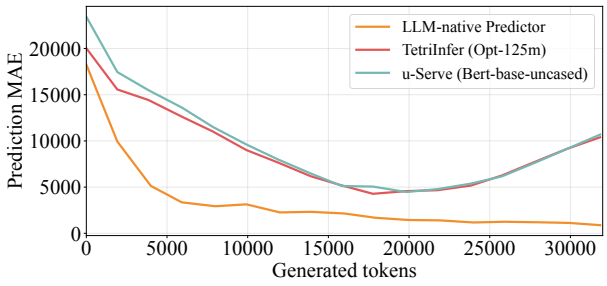



**Figure 7.** MAE of each prediction model for requests with 30-32K output tokens at different generated tokens.

$\mathcal{D} = \{(\mathbf{h}_t, y_t)\}$ 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 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., $\mu$-Serve (bert-base-uncased) [27] and TetriInfer (opt-125m) [14], 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 [22] and employing early stopping [25] 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 [34], a prompt-based method that modifies user instructions to have the LLM first predict its output length before generating the response; 2) $\mu$-Serve [27], which uses bert-base-uncased as an auxiliary model; 3) TetriInfer [14], 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, the cost is related to the model size, where our LLM-native predictor is significantly smaller than auxiliary models and original LLM used in prompt-based methods, leading to the lowest latency.

**Table 1.** Comparison of various generation length prediction methods.

| Methods | PiA | $\mu$-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 | 14169.29 | 8165.79 | 7658.14 | 3873.21 |
| Latency (batch:1) | 2.20 s | 6.00 ms | 10.29 ms | 1.33 ms |
| Latency (batch:10) | 15.25 s | 30.04 ms | 65.27 ms | 2.44 ms |

## 5 Decode Rescheduling

With the help of LLM-native predictor described in Section 4, we can model the future state of the entire 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.

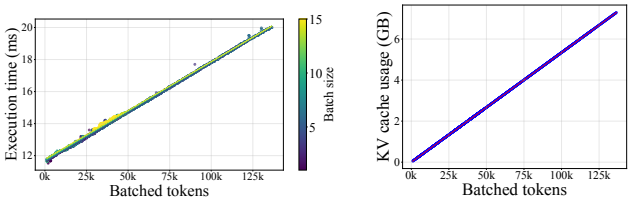
### 5.1 From Prefill-to-Decode Scheduling to Decode-to-Decode Rescheduling

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:** In prefill to decode scheduling, the scheduler must assign every incoming request to a decode instance, as each request must be processed in decode phase. In contrast, in decode rescheduling, the scheduler can choose not to migrate any requests when 1) the system is already balanced, or 2) the potential benefits of migration do not justify the overhead incurred, 3) one request in a high-load decode instance has high probability to finish soon, which can naturally relieve the load imbalance.

**Which request to migrate out:** We notice that in prefill to decode scheduling, the migration out decision is implicitly made when the request finishes prefill, as the scheduler must assign it to a decode instance. In contrast, in decode rescheduling, for a high-load decode instance, the scheduler must explicitly select one request to migrate out. The choice of which request to migrate out is critical, as migrating a long request can significantly reduce the load on the source instance, but requires transferring a large KV cache and potentially risks high-load on the target instance. Conversely, migrating a short request incurs lower overhead but may not sufficiently alleviate the source instance's load. Moreover, the remaining length of future tokens also impacts the scheduler's decision, as migrating a request that is about to finish soon may not be worthwhile.

**Which instance to migrate in:** In traditional prefill to decode scheduling, the scheduler must select a decode instance for each incoming request. Similarly, in decode rescheduling,




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

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

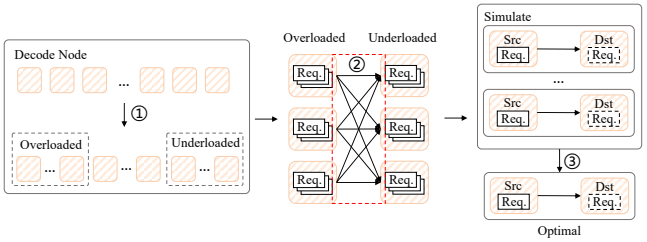



**Figure 9.** Workflow of the scheduler.

the scheduler must choose a target instance for each request being migrated out. This decision is influenced by the current load and resource availability of each decode instance, as well as the characteristics of the request being migrated.

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. Specifically, the scheduler need to consider the predicted remaining generation length of each request to make informed migration decisions that optimize system performance in the near future rather than just the current state.

### 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 [5], 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 [33], 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 directly correlates with balancing execution time and memory usage.

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 over the prediction horizon 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 $\beta_t$ 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 optimal 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.

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 11-17).** 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 18-24).** 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 21), and (2) memory safety ensuring no OOM on target instance in the near future (Line 22).
- **Phase 3: Optimal Selection (Lines 25-32).** The algorithm evaluates each candidate migration by aggregating pre-computed worker simulations (Line 29) and computing the incremental variance reduction (Line 30). The optimal 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.** We analyze the computational complexity from both the scheduler and worker perspectives: The scheduler performs three main operations: (1) instance classification with $O(n)$ complexity to identify overloaded instances $\mathcal{O}$ and underloaded instances $\mathcal{U}$; (2) migration candidate generation with $O(|\mathcal{O}| \cdot |\mathcal{U}|)$ complexity; (3) simulation-based evaluation with $O(|\mathcal{O}| \cdot R_{\max} \cdot |\mathcal{U}| \cdot H)$ complexity, where $R_{\max}$ is the maximum number of requests per instance and $H$ is the prediction horizon. The overall scheduler time complexity is $O(|\mathcal{O}| \cdot R_{\max} \cdot |\mathcal{U}| \cdot H)$. Each worker maintains local state information with $O(R_{\max})$ space complexity and performs prediction computations with $O(H)$ time complexity per request. The total worker-side space complexity is $O(H \cdot R_{\max})$ across all instances.

In Section 6.4, we simulate large-scale clusters with 256 instances, where the average completion time of these computations does not exceed 300ms, which can further 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 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.

**Algorithm 1** Decode Rescheduling

```
1:  // Main Entry - Scheduler Side
2:  while Serving requests do
3:      After a fixed interval (e.g., every 1 second)
4:      Collect States, update average workload w̄
5:      (O, U) ← InstanceClassification(w̄)
6:      if O ≠ ∅ then
7:          C ← CandidateEnumeration(O, U, S)
8:          m* ← OptimalSelection(C, σ_0^2)
9:          if m* ≠ ∅ then
10:             ExecuteMigration(m*)
11: function InstanceClassification(w̄)
12:     for each instance i ∈ I do
13:         w_i = Σ_{t=1}^{H} β_t · N̂_i(B_{i,t})
14:     O ← {i ∈ I | w_i > (1 + θ) · w̄}
15:     U ← {i ∈ I | N_i(B_{i,0}) < (1 + θ) · w̄}
16:     return (O, U)
17: function CandidateEnumeration(O, U, S)
18:     C ← ∅
19:     for each (s, t) ∈ O × U do
20:         R_s ← {r ∈ B_s | N̂(r) > C_mig / T_exec}
21:         R_s ← R_s ∩ {r | N_t(B_{t,0}) + N̂(r) ≤ C_mem}
22:         C ← C ∪ {(r, s, t) | r ∈ R_s}
23:     return C
24: function OptimalSelection(C, σ_0^2)
25:     m* ← null
26:     σ^2_max ← 0
27:     for each candidate (r, s, t) ∈ C do
28:         Ŝ ← AggregateSimulations(S, r, s, t)
29:         σ^2 ← SimulateFutureVariance(Ŝ, H)
30:         m* ← UpdateBestCandidate(m*, σ^2, σ^2_max)
31:     return m*
```

### 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 transfered 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.

# 6 Evaluation

## 6.1 Experimental Setup

**Implementation.** We implement ARES 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. ARES 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 the three-phase 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 6.4.

**Environment and models.** We validate ARES across multiple hardware configurations including NVIDIA H800, A100, and RTX 4090D GPUs. By default, we employ 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. H800 and A100 are further used to demonstrate the generality of our approach across different hardware platforms. In addition, we also develop a simulation framework to evaluate large-scale clusters as detailed in Section 6.4. In order to verify the effect under long output requests, we use the reasoning model DeepSeek-R1-Distill-Qwen-7B [7] as base model and adopt W8A8 quantization to support longer output lengths [9], which supports up to 32K tokens for both prompt and output.

**Workload.** We utilize two widely adopted real-world datasets, ShareGPT [30] and Alpaca [29], to evaluate the performance of ARES. By default, we present results using the ShareGPT dataset. The prompt and generation length distribution of the dataset 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.

**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 |

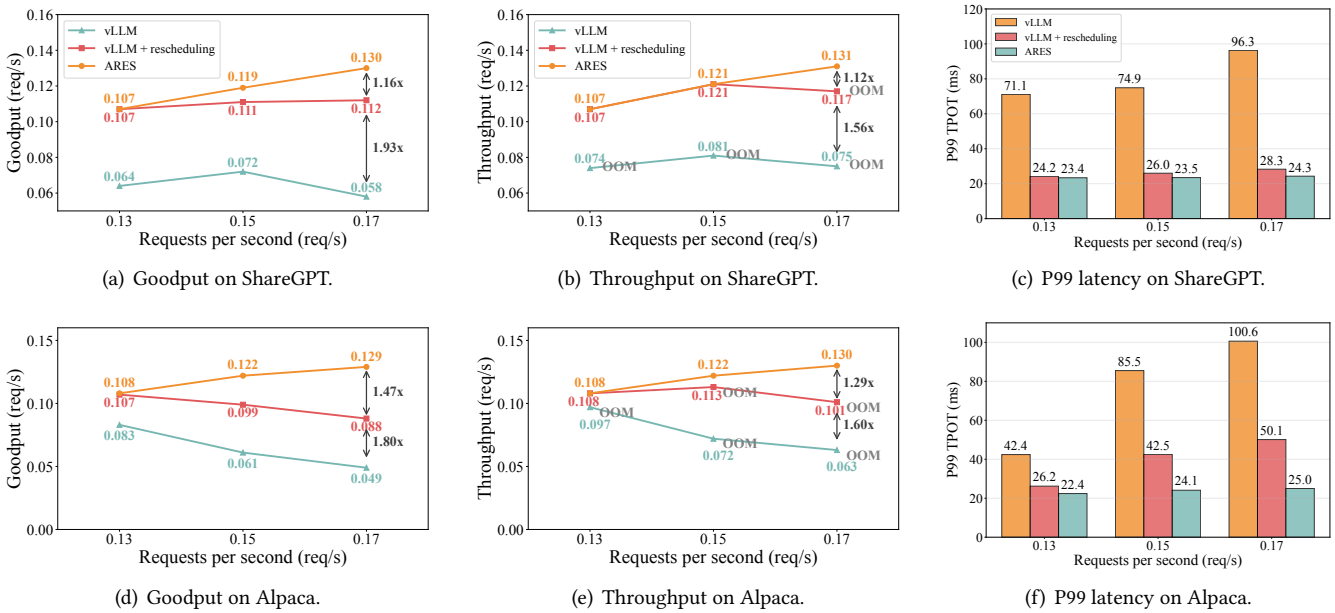
**Baselines.** We evaluate following migration strategies under identical *prefill-decode real-time KV cache load balancing* (serving as the common foundation):

- *vLLM [16]*: we employ vLLM's built-in prefill-decode disaggregation architecture as the baseline, which adopts the idea from DistServe [35]. Moreover, we change vLLM's default prefill-to-decode scheduling strategy, i.e., round-robin, to *prefill-decode real-time KV cache load balancing* to ensure a fair comparison.
- *vLLM + rescheduling*: 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.
- *ARES (Ours)*: This is our proposed approach that integrates the generation length predictor and the decode rescheduling algorithm.

## 6.2 Comparison with SOTA

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) and Figure 10(e), *ARES* achieves higher throughput compared to the baseline without migration, particularly under high load conditions (1.75× at RPS=0.17, ShareGPT dataset). Higher load may lead to OOM issues in decode instances due to workload imbalance, causing requests to recompute or offloading the KV cache, which significantly degrades throughput. Moreover, the involvement of prediction do not negatively impact throughput, as shown in lower workload conditions (at RPS=0.13, both *ARES* and *vLLM + rescheduling* achieve 0.107 req/s throughput), while effectively mitigating workload imbalance in high load scenarios, where vLLM with only rescheduling suffers from occasional OOMs.




(a) Goodput on ShareGPT. (b) Throughput on ShareGPT. (c) P99 latency on ShareGPT.

(d) Goodput on Alpaca. (e) Throughput on Alpaca. (f) P99 latency on Alpaca.

**Figure 10.** Overall performance under various RPS on ShareGPT and Alpaca datasets.

**Goodput.** In addition to throughput, we further employ goodput as a more comprehensive metric to incorporate SLO attainment, denoting the effective thoroughput, i.e., requests per second that meet their SLOs. Compared to throughput, *ARES* presents a more pronounced advantage in goodput, as the workload imbalance further leads to increased tail latency and SLO violations. Figure 10(a) demonstrates that *ARES* achieves 1.93× goodput over *vLLM* and 1.16× goodput over *vLLM + rescheduling*, which attributes to the effective workload balancing that reduces SLO violations and prevents OOM issues.

**Latency.** We also evaluate the tail latency, e.g., P99 TPOT of different methods, as it directly impacts user experience in real-time applications. As shown in Figure 10(c) and Figure 10(f), *ARES* significantly reduces P99 latency across both datasets. On ShareGPT at RPS=0.17, migration reduces P99 latency from 96.3ms to 28.3ms (70.6% improvement), while prediction further reduces it to 24.3ms (additional 14.1% improvement). On Alpaca at RPS=1.7, migration reduces P99 latency from 100.6ms to 80.8ms (19.7% improvement), with prediction achieving an additional 45.4% reduction. These results demonstrate the effectiveness of both migration and prediction components across different data characteristics.

### 6.3 Impact of Hardware Configuration

In LLM serving systems, two primary hardware factors influence the performance, namely GPU compute power and network bandwidth. GPU compute power directly affects the execution time of each token, while network bandwidth determines the KV cache transfer time during migration. Therefore, we extend our evaluation to include NVIDIA H800 and A100 GPUs, which have different compute capabilities, and we also vary the network bandwidth to assess its impact on migration overhead.

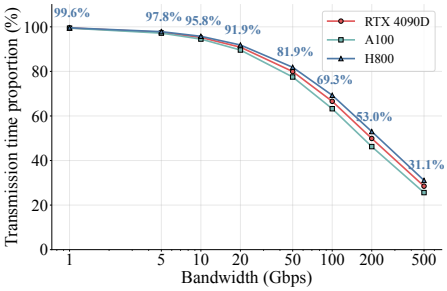
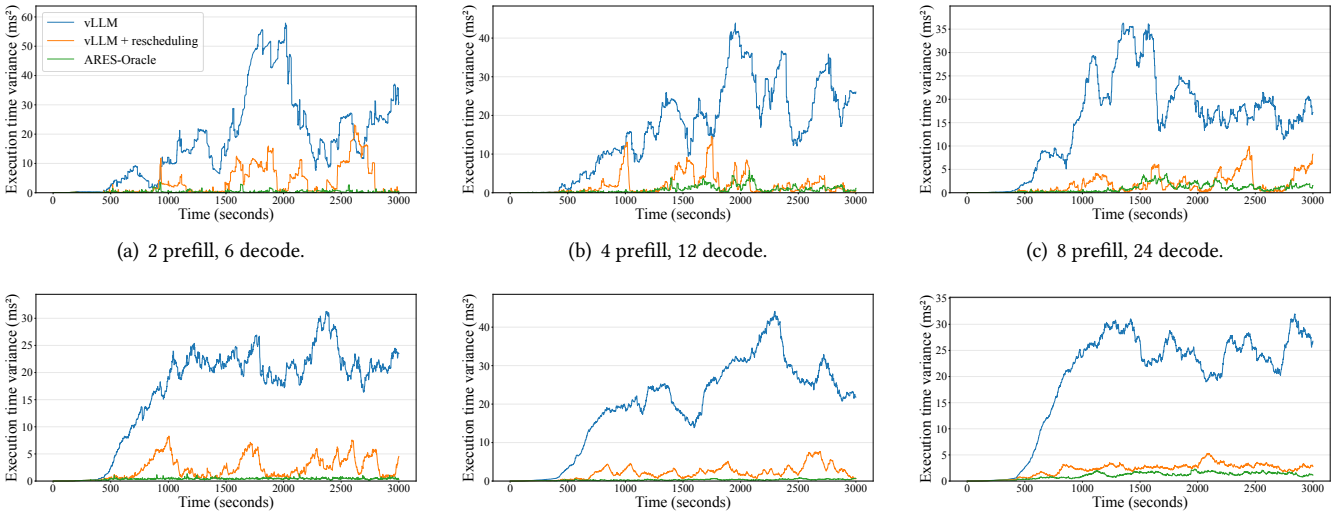



**Figure 11.** Transmission time proportion in TBT on various hardware.

Figure 11 illustrates the proportion of transmission time within a token's execution time when triggering migration under different network bandwidths and hardware configurations. Our evaluation shows that bandwidth limitations severely exacerbate TBT degradation, particularly in constrained network environments. For example, under 10 Gbps bandwidth, transmission overhead accounts for 95.5% of the total TBT for H800 hardware, causing significant user-perceived stuttering. This analysis underscores the necessity of our prediction-based method to make optimal migration decisions to reduce unnecessary overhead.

### 6.4 Balance Analysis

To demonstrate the performance upper bound of our prediction-based rescheduling approach, we introduce an oracle version




(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 12.** Execution time variance comparison across different cluster sizes under 25 Gbps transfer speed.

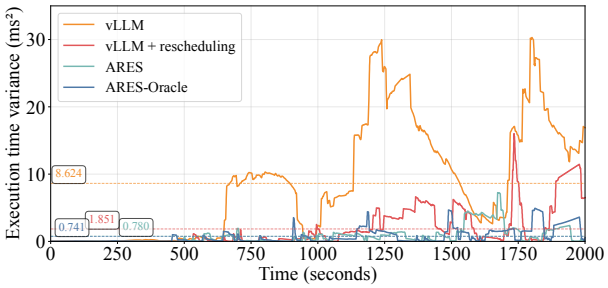



**Figure 13.** Execution time variance across different scheduling algorithms on high-load dataset.

*ARES-Oracle*, which assumes perfect knowledge of the remaining 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 13 illustrates the execution time variance across different scheduling strategies when running on 3 decode instances with RTX 4090D GPUs. Our proposed prediction solution achieved an average execution time variance of 0.78 ms², which is close to that of oracle prediction. We observe that vLLM presents a bursty execution time pattern due to workload imbalance, while rescheduling effectively mitigates this issue. Furthermore, prediction further reduces execution time variance by proactively balancing workloads, thereby enhancing system stability and user experience.

**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. Our simulator implementation mirrors our real system, following the same execution flow as our actual experiments. 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 configure the simulation with a request rate of 0.3 RPS for an 8-instance cluster, scaling proportionally with cluster size (e.g., 0.6 RPS for 16 instances). 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 the data from ShareGPT, we test three strategies: *vLLM*, *vLLM + rescheduling*, and *ARES-Oracle* under 25 Gbps transfer speed (following DistServe's cross-node interconnection speed setting [35]). We evaluate execution time variance across cluster sizes ranging from 8 to 256 instances, as shown in Figure 12. As the inference service continues to run, rescheduling effectively improves cluster load balancing, while prediction further reduces load fluctuations.

## 7 Conclusion

In this paper, we propose ARES, 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 ARES significantly enhances load balance and achieves up to 2.24× 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.

## References


[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

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

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

[4] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. Efficient and economic large language model inference with attention offloading, 2024.

[5] Ke Cheng, Wen Hu, Zhi Wang, Hongen Peng, Jianguo Li, and Sheng Zhang. Slice-level scheduling for high throughput and load balanced llm serving, 2025.

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

[7] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[8] 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. Deepseek-v3 technical report, 2024.

[9] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

[10] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 111–126, Santa Clara, CA, July 2024. USENIX Association.

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

[12] Horace He and Thinking Machines Lab. Defeating nondeterminism in llm inference. *Thinking Machines Lab: Connectionism*, 2025. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

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

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

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

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

[17] Lujun Li, Lama Sleem, Niccolo' Gentile, Geoffrey Nichil, and Radu State. Exploring the impact of temperature on large language models:hot or cold?, 2025.

[18] Weiqing Li, Guochao Jiang, Xiangyong Ding, Zhangcheng Tao, Chuzhan Hao, Chenfeng Xu, Yuewei Zhang, and Hao Wang. Flowkv: A disaggregated inference framework with low-latency kv cache transfer and load-aware scheduling, 2025.

[19] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees, 2024.

[20] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test, 2025.

[21] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty, 2025.

[22] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[23] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. 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*, ASPLOS '24, page 1112–1127, New York, NY, USA, 2024. Association for Computing Machinery.

[24] OpenAI. Chatgpt. https://chat.openai.com, 2025. Accessed: 2025-08-30.

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

[26] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 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)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[27] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Power-aware deep learning model serving with {$\mu$-Serve}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 75–93, 2024.

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

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

[30] ShareGPT Teams. Sharegpt. https://sharegpt.com/, 2023. Accessed: 2025.

[31] vLLM Project. vllm disaggregated prefill. https://docs.vllm.ai/en/latest/examples/online_serving/disaggregated_prefill.html, 2025. Accessed: 2025-09-09.

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

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

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

[35] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.