Arrow: Adaptive Scheduling Mechanisms for Disaggregated LLM Inference Architecture

Yu Wu
University of Science and Technology of China

Tongxuan Liu
University of Science and Technology of China

Yuting Zeng
University of Science and Technology of China

Siyu Wu
Beihang University

Jun Xiong
JD.com

Xianzhe Dong
University of Science and Technology of China

Hailong Yang
Beihang University

Ke Zhang
JD.com

Jing Li
University of Science and Technology of China

Abstract

Existing large language models (LLMs) serving systems typically employ Prefill-Decode disaggregated architecture to prevent computational interference between the prefill and decode phases. However, real-world LLM serving scenarios often exhibit significant fluctuations in request input/output lengths, causing traditional static prefill/decode node configuration ratio to result in imbalanced computational loads between these two nodes, consequently preventing efficient utilization of computing resources to improve the system's goodput. To address this challenge, we design and implement Arrow, an adaptive scheduler that leverages stateless instances and elastic instance pools to achieve efficient adaptive request and instance scheduling. Arrow dynamically adjusts the number of instances handling prefill and decode tasks based on real-time cluster performance metrics, significantly enhancing the system's capability to handle traffic spikes and load variations. Our evaluation under diverse real-world workloads shows that Arrow achieves up to 5.62× and 7.78× higher request serving rates compared to state-of-the-art PD-colocated and PD-disaggregated serving systems respectively.

Keywords: Inference Serving; Prefill-Decode Disaggregation; Request Scheduling

1 Introduction

Large language models (LLMs), such as Gemini [41, 42], GPT [28, 29], Llama [9, 43], Qwen [2, 34], and DeepSeek [5, 6], have achieved remarkable success across various domains in both industry and academia. Building upon these models, a series of innovative applications have emerged, including LLM-based search engines [16, 24], personal assistants [14, 15], and embodied intelligence systems [21, 45], highlighting the tremendous application potential of LLMs. However, deploying these LLMs' inference services in production environments presents numerous challenges. The massive parameters of LLMs and ultra-long input sequences impose tremendous computational and memory demands, making it difficult to consistently meet strict Service Level Objectives (SLOs) even when using high-performance GPU servers [50, 51, 53].

In LLM inference services, token generation proceeds iteratively in an autoregressive manner, where each iteration decodes the next token based on previous token sequence. The inference process is typically divided into two distinct phases: prefill and decode. During the prefill phase, the entire input token sequence undergoes forward propagation to generate the first output token. In the decode phase, each newly generated token is concatenated to the end of the current token sequence and used as input for generating the subsequent token. This process iterates continuously, meaning that for a request with output length N, the LLM must sequentially execute N iterations. Two key metrics are commonly used to evaluate the performance of LLM inference services: (1) Time-to-First-Token (TTFT), measuring the latency to generate the first token in the prefill phase; and (2) Time-per-Output-Token (TPOT), representing the average token generation latency in the decode phase. Efficient inference services must satisfy strict SLOs under limited hardware resources while maximizing the system's goodput.

Recently, researchers have proposed various optimization techniques to improve the overall throughput of serving systems and meet SLOs of diverse applications. A notable advancement is continuous batching [49], which implements iteration-level scheduling to dynamically add or remove request in the batch during each computation iteration. This approach provides greater flexibility compared to traditional request-level scheduling, significantly reducing queuing latency. Continuous batching has been integrated into inference frameworks like vLLM [20] and TensorRT-LLM [30], becoming a widely adopted technology for LLM serving. Subsequent studies introduced chunked prefill [1, 10], which divides input sequence into smaller chunks and batches them with decode tasks to mitigate the latency spikes caused by lengthy prompts, further enabling stall-free scheduling that

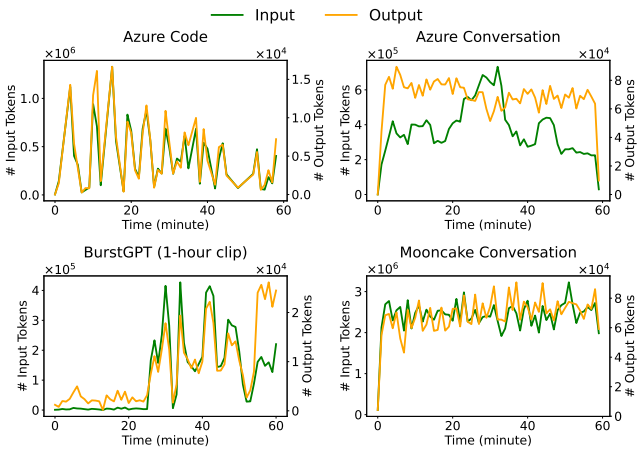

Figure 1. Total request input/output length per minute over time in different LLM serving systems.

allows new requests to be added without interrupting ongoing decode processing. However, recent work [12, 31, 52] have shown that prefill and decode phases exhibit fundamentally distinct computational characteristics and latency requirements. The colocating prefill and decode computation from different requests creates mutual interference, causing increased TTFT and TPOT that ultimately degrades the system's goodput.

To address the interference between prefill and decode phases, DistServe [52] assigns these phases to separate instances, eliminating phase interference while enabling independent optimization of resource allocation and parallelization strategies for each phase. Splitwise [31] further explores both homogeneous and heterogeneous cluster deployments to optimize cost-efficiency and throughput. To further improve scheduling in disaggregated architecture, TetriInfer [12] designs a two-level scheduling algorithm that performs request scheduling based on resource utilization prediction to prevent decode scheduling hotspots. Meanwhile, Mooncake [32] and MemServe [13] enhance serving performance through distributed KV cache storage pools, enabling cache reuse across requests. While disaggregation resolves phase interference, a key challenge remains: properly configuring the ratio between two distinct prefill workers and decode workers to maximize goodput. Improper configuration ratio can lead to severe performance degradation [32].

We observe substantial variability in input and output lengths across real-world LLM inference workloads. This observation is drawn from diverse production traces as illustrated in Figure 1, including Azure LLM inference services (Azure Code and Azure Conversation) [31], Azure OpenAI GPT service (BurstGPT) [46], and Kimi conversation service (Mooncake Conversation) [32]. The variation of input/output length directly impacts the workload distribution between prefill and decode nodes [7, 52], making the optimal prefill-decode (PD) ratio configuration highly sensitive to workload patterns. Consequently, static PD ratio fails to achieve optimal performance under such fluctuating conditions, necessitating adaptive resource allocation strategies.

To address the above challenge, we first conduct an in-depth analysis of workload variations in real-world inference services. Our study reveals that existing Prefill-Decode disaggregated systems exhibit lagging instance scheduling when handling dynamic workload changes (§3). Based on the request processing workflow of Prefill-Decode disaggregated systems, we derive crucial insights for request and instance scheduling (§4). Building upon these analyses and insights, we design **Arrow**, an adaptive request and instance scheduler that dynamically schedules requests and instances based on SLO settings and instance load (§5, §6). Arrow employs stateless instances where each instance can process both prefill and decode requests without dedicated roles. The system features an SLO-aware scheduling algorithm where a global scheduler dynamically adjusts request dispatching and instance allocation based on: (1) predicted TTFT for incoming requests, (2) real-time token generation intervals of ongoing requests, and (3) target SLO metrics. This design enables efficient adaptation to varying workloads through adaptive request and instance scheduling.

We implement Arrow based on vLLM [20] and evaluate its performance against both PD-colocated systems and PD-disaggregated systems using diverse production workloads (§7). Experimental results demonstrate that Arrow can significantly outperform existing approaches. It delivers 3.60× ∼ 5.62× higher request serving rates than state-of-the-art PD-colocated systems and 4.06× ∼ 7.78× improvements over PD-disaggregated systems under the given SLO constraints.

In summary, our main contributions are as follows:

- Identify that fluctuations in input/output lengths can lead to suboptimal goodput under traditional static PD configuration ratio and propose several key insights for more effective request and instance scheduling.
- Design a novel scheduler Arrow that enables adaptive request and instance scheduling through stateless instances and elastic instance pools.
- Conduct a comprehensive performance evaluation of Arrow using real-world workloads, demonstrating the effectiveness of its adaptive scheduling strategy.

2 Background

2.1 LLM Inference

Modern LLMs [5, 6, 9, 28, 34] mostly adopt the transformer architecture [44] and process input sequences through an autoregressive generation process. To avoid redundant computation, existing inference engines typically employ KV Cache [19] to cache intermediate results, thereby dividing

the computation for a single request into two phases: Prefill and Decode. During the prefill phase, the inference engine processes the user's input, generates KV Cache for all input tokens, and produces the first output token. In the decode phase, the engine computes KV Cache for each newly generated token in subsequent iterations and outputs the next token. Since both phases require shared access to KV Cache, existing serving systems typically colocates these two phases within the same instance. Techniques such as continuous batching [49] and chunked prefill [1] are employed to further optimize the system's overall throughput.

2.2 Prefill-Decode Disaggregation

Traditional LLM serving systems optimize computation by colocating prefill and decode phases on the same instance. However, recent studies have highlighted significant differences in computational characteristics between the prefill and decode phases [31, 52], which can lead to mutual interference between the two phases [12] and suboptimal hardware resource allocation [52]. To address these issues, researchers have proposed the Prefill-Decode disaggregated inference architecture [12, 31, 52], which divides compute instances into two types: prefill instances and decode instances, each dedicated to handling their respective phases. After completing the prefill computation for a request, the prefill instance transfers both the request and its corresponding KV Cache to a decode instance via high-speed interconnects such as NVLink or InfiniBand. The Decode instance then proceeds with the subsequent decode computations. This architecture eliminates computational interference between the two phases and enables independent optimization of parallelization strategies and resource allocation for each phase by decoupling prefill and decode, improving the flexibility in performance tuning and overall system goodput.

3 Motivation

The Prefill-Decode disaggregated inference architecture enables independent optimization for both prefill and decode phases. However, we observe that existing disaggregated systems employing static instance partitioning schemes suffer from low hardware resource utilization and inadequate responsiveness to traffic bursts. In this section, we will elaborate on this issue using production traces from real-world LLM serving systems and present our key insights for addressing it.

3.1 Diversity of Workloads

We conduct an in-depth analysis of the four traces from real-world LLM inference serving systems mentioned in Section 1. Each trace records request information processed by the inference serving system over a period, including arrival times and input/output lengths. Figure 1 shows the total input and output lengths of requests per minute over time,

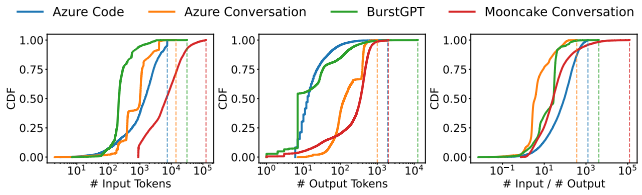

Figure 2. Input and output length distributions of different traces.

while Figure 2 presents the cumulative distribution functions (CDFs) of input and output lengths across these traces. We observe that these traces exhibit diversity in both vertical and horizontal dimensions:

- **Vertical diversity** refers to significant load variations in the same trace. Figure 2 reveals extreme differences in input length, output length, and input-to-output length ratios among requests within a single trace, with minimum and maximum values differing by hundreds to tens of thousands of times. Figure 1 demonstrates substantial temporal variations in request input and output lengths within traces. For instance, in the Azure Code trace, the lowest-load minute exhibits input/output lengths of 25.7K / 0.25K, while the peak-load minute reaches 1327.9K / 16.6K - representing a 50× difference in input length and 65× difference in output length.
- **Horizontal diversity** refers to significant variation in workload characteristics across traces. Figure 1 shows varying traffic burstiness patterns: Azure Code and BurstGPT exhibit frequent bursts (input length coefficient of variation $c_v = 0.80$ and 1.11 respectively), while Mooncake Conversation maintains relatively stable loads ($c_v = 0.16$). Load predictability also varies significantly, with Azure Code showing strong input-output length correlation (correlation coefficient $r = 0.95$) compared to Azure Conversation's weaker correlation ($r = 0.29$). Figure 2 highlights differing length distributions across scenarios - requests in Azure Code feature larger median input lengths but smaller median output lengths than Azure Conversation.

Prior works [7, 52] have conducted detailed analyses of the computational demands for the prefill and decode phases, showing that the load on prefill phase scales quadratically with the input length, while the load on decode phase grows linearly with the total number of tokens in the batch. This fundamental difference in scaling characteristics leads to distinct load growth rates between prefill and decode instances. The observed diversity in input/output lengths further exacerbates load volatility, causing significant and often load fluctuations in real-world serving scenarios. Consequently, inference serving systems employing the Prefill-Decode disaggregated architecture must dynamically adjust the ratio between Prefill and Decode nodes to efficiently accommodate varying workload patterns.

3.2 Lagging Instance Scheduling

To address vertical diversity, existing Prefill-Decode disaggregated systems typically adjust the types of instances dynamically. When significant workload changes occur, the flipping of instance is performed by converting prefill instances to decode instances or vice versa. However, the instance flipping strategies adopted by current systems generally suffer from long response times. DistServe [52] monitors workload characteristics and reruns the cluster deployment algorithm when substantial changes are detected, requiring each instance to reload model weights. Similarly, Splitwise [31] alters types of instances that have remained in a mixed state for an extended period, also necessitating model reloading during the flipping process. TetriInfer [12] flips instances with persistently low utilization, but the flipping process must wait until the instance finishes processing its current requests. These approaches typically involve multiple steps, including observation, waiting for flipping conditions, and restarting instances. The entire process often takes several minutes to finish, exhibiting significant scheduling latency. As a result, the serving system struggles to promptly adapt to workload fluctuations, particularly when handling sudden traffic bursts. Moreover, this approach introduces additional instance downtime, which further degrades the system's overall serving capacity.

3.3 Inflexibility of Workload Profiling

To address horizontal diversity, existing work typically determines the PD ratio based on profiling or simulator data. PD-Serve [17] proposes a set of constraints that the number of prefill and decode instances should satisfy and uses profiling to determine the optimal PD ratio. Splitwise [31] models request arrivals using a Poisson distribution and employs a simulator to search for PD ratio configuration that meets SLO settings. Mooncake [32] evaluates different PD ratios on trace data to select the best-performing configuration based on TTFT and TBT(Time-Between-Tokens) metrics. However, profiling-based methods are only effective when request arrival patterns and length distributions remain relatively stable. In scenarios with significant workload fluctuations, pre-collected profiling or simulator data may fail to accurately reflect real-time request characteristics, leading to mismatches between preset PD ratios and actual load demands. Consequently, these profiling and simulation approaches cannot consistently achieve optimal performance across diverse scenarios, lacking the flexibility and generality required for robust PD ratio configuration.

3.4 Opportunities

To address the above challenges, we propose an adaptive request and instance scheduling strategy for Prefill-Decode disaggregated serving systems. Our key insights are:

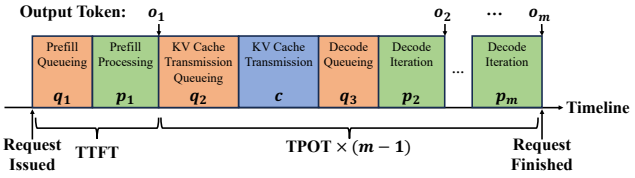

Figure 3. Request processing timeline in Prefill-Decode disaggregated inferenece system.

- Prefill and decode should be treated as properties of requests rather than attributes of instances. By employing stateless instances, the system can eliminate the overhead of flipping delays and instance restarts, thereby enabling real-time PD ratio adjustment.
- For PD-disaggregated systems that aims to satisfy strict SLO settings, scheduling decisions should be made directly based on real-time TTFT and TPOT metrics rather than relying on indirect indicators like request length distributions, arrival patterns, or instance utilization that require prolonged observation. This approach eliminates extensive monitoring and profiling overhead, allowing rapid response to traffic bursts and workload variations while maintaining generality and adaptability of the scheduling policy across diverse scenarios.

These key insights enable efficient handling of workload diversity while avoiding the latency and inflexibility issues in prior work. In the following sections, we first derive some key scheduling principles by analyzing request processing steps in PD-disaggregated systems (§4), and then elaborate on how to leverage these insights to design an Prefill-Decode disaggregated LLM serving system that performs real-time adaptive adjustments based on load (§5).

4 Analysis

In this section, we present several key insights for request and instance scheduling in PD-disaggregated systems.

4.1 Request Processing

Figure 3 illustrates the complete request processing workflow in a typical PD-disaggregated system [20, 52]. Consider a request r with output tokens $o_1 o_2 \cdots o_m$, the request is first dispatched to a prefill instance, experiencing prefill queuing delay q_1 before starting prefill computation (duration p_1). The system then waits (q_2) for a decode instance to fetch both the request and its corresponding KV Cache, with transfer time c. After decode queuing delay q_3, the decode instance begins iterative token generation, where each iteration produces one token with computation time p_2 through p_m. We assume that prefill instances process one request per batch, while decode instances maximizes batch size by grouping multiple decode requests within given batch size and GPU memory

constraints. This scheduling approach is commonly used in existing PD-disaggregated systems [20, 52].

4.2 TTFT

TTFT (Time-to-First-Token) is a key indicator of the processing capability of prefill instances. It is defined as the time from when the user issues a request until the first token is received, corresponding to the $q_1 + p_1$ duration illustrated in Figure 3. Suppose there are n prefill requests $r_1 \cdots r_n$ to be processed on the prefill instance. Let a_i denote the arrival time of the i-th request, e_i denote its computation completion time, $q_1^{(i)}$ denote the prefill queuing delay, and $p_1^{(i)}$ denote the prefill processing time. Then we have:

$$\mathrm{TTFT}_i = q_1^{(i)} + p_1^{(i)} = \max\{e_{i-1} - a_i, 0\} + p_1^{(i)} \tag{1}$$

$$e_i = a_i + \mathrm{TTFT}_i \tag{2}$$

Specifically, $\mathrm{TTFT}_1 = p_1$. We summarize three key characteristics of TTFT:

- **Strong Predictability**: Equations 1 and 2 indicate that the TTFT of the i-th request can be uniquely determined by the arrival times and prefill processing times of the first to i-th requests. Since the computation time of a single prefill request is proportional to the square of the input length [7, 32, 52], the precise functional relationship can be determined through profiling. Thus, the TTFT of each request in the queue can be accurately predicted.
- **Monotonicity**: Starting from the moment a user issues a request, the TTFT of the request can only increase monotonically with the processing time. If the current queuing delay and computation time exceed the TTFT SLO before the prefill phase completes, the request can no longer meet the SLO requirement.
- **Sensitivity to Burst Traffic**: Consider the case where n requests arrive as a burst, meaning their arrival times t_i fall within a narrow interval. In this case, t_i can be approximated as a constant, and the monotonic increase of e_i causes the TTFT of requests 1 through n to exhibit an increasing trend.

Insight 1: The strong predictability of TTFT enables the serving system to leverage queue information from prefill instances to accurately predict the TTFT of new requests, thereby anticipating potential violations of TTFT SLO.

Insight 2: The monotonicity of TTFT and its sensitivity to burst traffic imply that the serving system cannot rely on monitoring the TTFT metrics of completed requests to make instance scheduling decisions. Otherwise, this approach may lead to TTFT SLO violations for later-queued requests in bursty traffic scenarios, with no remedial actions available to bring these requests back into SLO compliance.

4.3 TPOT

The TPOT (Time-per-Output-Token) metric is a key indicator of the processing capability of decode instances. It represents the average waiting time between every two consecutive tokens received by the user. Let denote the time interval t_{j+1} between the output tokens o_j and o_{j+1} of request r. Then, TPOT can be expressed as:

$$\mathrm{TPOT} = \begin{cases} \frac{\sum_{j=2}^{m} t_j}{m-1} = \frac{\text{Decode Phase Time}}{m-1} & \text{if } m \geq 2 \\ 0 & \text{if } m = 1 \end{cases} \tag{3}$$

As shown in Figure 3:

$$t_j = \begin{cases} q_2 + c + q_3 + p_2 & \text{if } j = 2 \\ p_j & \text{if } j > 2 \end{cases} \tag{4}$$

We focus our analysis on the four components of t_2:

- KV Cache Transmission Queueing Delay q_2: The prerequisite for a decode instance to fetch the KV Cache of a decode request is having sufficient GPU memory. However, the iterative token generation process of LLMs makes it difficult to determine the output length of each request in advance, further complicating the estimation of the available GPU memory space of decode instances at any given moment. Consequently, KV Cache transmission queuing delays are highly unpredictable for new requests when the decode instance is under high load with limited available memory.
- KV Cache Transmission Time c: It is determined by the size of the KV Cache to be transmitted and the available bandwidth.
- Decode Queueing Delay q_3: The prerequisite for scheduling Decode requests is that neither the batch size limit nor memory capacity is reached. Similar to the analysis of KV Cache transmission queuing delay, since the completion time of ongoing decode requests cannot be determined, the queuing delay for new requests to be scheduled becomes unpredictable when the decode instance is under high load.
- Decode Iteration Time p_2: Previous works [7, 52] have shown that the computation in decode phase is proportional to the number of tokens in the batch. The relationship between processing time and token count can be determined through profiling.

Based on the above analysis, We summarize two key characteristics of TPOT:

- **Weak Predictability**: Equation 4 and the related analysis indicate that TPOT is influenced by multiple factors. The uncertainty in request output length makes several delays difficult to predict.
- **Non-monotonicity**: Equation 3 shows that TPOT is determined by the generation intervals of all tokens. Therefore, for the first a and b output tokens of a request, $\frac{\sum_{j=2}^{a} t_j}{a-1}$ and $\frac{\sum_{j=2}^{b} t_j}{b-1}$ can have an arbitrary relationship in magnitude. Thus, the TPOT of a request does not exhibit a definite monotonic relationship with processing time.

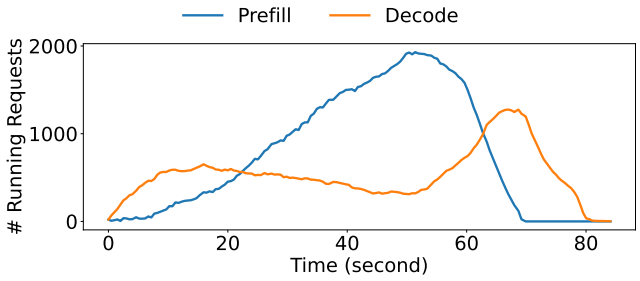

Figure 4. Prefill and Decode load over time when serving an LLM with 8B parameters under Azure Conversation workload using four prefill instances and four decode instances.

Insight 3. The weak predictability of TPOT makes it challenging for the serving system to accurately predict new requests' TPOT using input/output length characteristics and cluster load. Real-time monitoring of token generation intervals is required to detect TPOT SLO violations.

Insight 4. The non-monotonicity of TPOT allows decode instances to tolerate temporary workload spikes. Longer generation delays for some tokens do not always result in TPOT SLO violations.

4.4 Load Difference

We take a clip of the Azure Conversation trace from the 20th to the 40th minute as an example to analyze the load differences between prefill and decode instances under gradually increasing input load. This clip is characterized by a rising trend in request count per minute. As shown in Figure 1 showing a progressive increase in total input length during this period. Figure 4 illustrates the number of requests being processed by prefill and decode instances over time. The results reveal that under gradually increasing workload, since requests must be processed sequentially through prefill instance followed by decode instance, the prefill instances experience an earlier onset of load increase, peak load timing, and load decline compared to decode instances.

Insight 5. The mandatory Prefill → Decode computation order creates temporal misalignment in peak load patterns between prefill and decode instances, offers additional optimization opportunities for instance scheduling under bursty traffic: When prefill load increases, some decode instances with still-low load can be temporarily scheduled for prefill computation, until decode load begins to rise, at which point more instances should be reallocated to decode computation.

5 Design

5.1 Overview

Based on the analyses in Sections 3 and 4, we design Arrow, an adaptive scheduling engine for Prefill-Decode disaggregated inference architecture. As shown in Figure 5, Arrow adopts a stateless design for instances (III), allowing each instance to process both prefill and decode requests. These stateless instances are organized into elastic instance pools (V) for flexible resource management and scheduling. When a new request arrives, the global scheduler (II) uses the TTFT predictor (I) to estimate the expected TTFT and select an appropriate instance from the instance pool for request dispatching. The local scheduler (IV) queues both request computations and KV Cache transfers, deciding which requests and KV Cache transmissions to execute in each iteration. Each instance periodically reports performance data to the instance monitor (VI), which provides real-time metrics for the global scheduler to decide whether to move instances among instance pools.

5.2 Instance Management

Stateless Instance. Arrow designs instances to be stateless, allowing each instance to handle both prefill and decode requests. The request processing workflow in Arrow when using stateless instances is illustrated in Figure 6. When a new request arrives (a), the global scheduler selects instance A to process the prefill phase (b). Instance A notifies the global scheduler and returns the first output token o_1 to the user (c) after completing the prefill phase. The global scheduler then selects instance B to handle the decode phase (d). Upon receiving the Decode request, instance B pulls the KV Cache from instance A (e) and begins the iterative decode phase (f). This provides the global scheduler with greater flexibility in request and instance scheduling: (1) Each request is split into prefill and decode sub-requests, which can be scheduled independently. The global scheduler can use different scheduling strategies for these two types of requests, and may even assign both phases to the same instance if desired. (2) Prefill and decode are no longer treated as attributes of instances, but solely as attributes of requests, completely eliminating flip waiting time and instance restart time during instance scheduling.

Elastic Instance Pool. To facilitate the management of multiple instances, we use four elastic instance pools — Prefill, Decode, P→D, D→P to manage instances. These pools respectively represent instances handling prefill requests, instances handling decode requests, instances scheduled to handle decode requests but still processing unfinished prefill requests, and instances scheduled to handle prefill requests but still processing unfinished decode requests. When flipping an instance, Arrow simply removes the instance from its original pool and moves it into the new pool, achieving zero-wait-time instance scheduling. The global scheduler updates the instance's pool membership according to the instance transition diagram shown in Figure 5. For example, if the global scheduler decides to flip an instance from prefill pool and the instance still has ongoing prefill requests, it will be moved to the P→D pool. Once the instance completes all prefill requests, it is then moved to the decode pool according to the black transition edge.

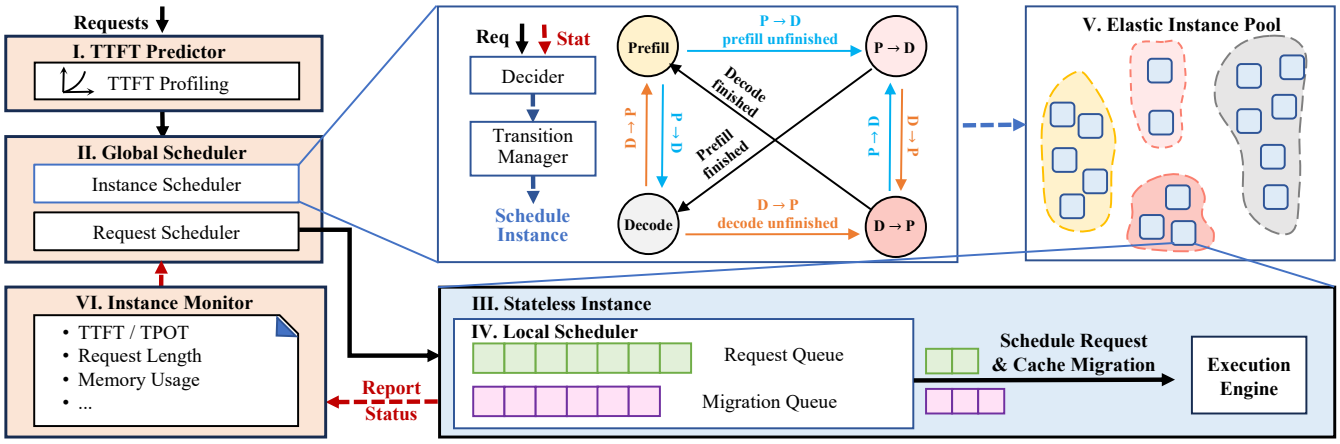

Figure 5. Arrow architecture overview.

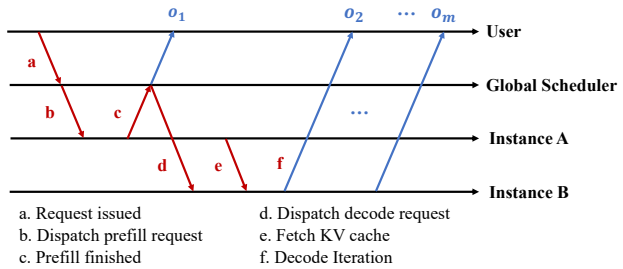

Figure 6. Request processing workflow in Arrow.

Instance Monitor. According to Insights 1 and 3, we deploy an instance monitor in the system, which periodically collects statistics such as the number and length of prefill and decode requests, GPU memory usage, TTFT, TPOT, and token generation intervals for each instance. The global scheduler makes scheduling decisions based on the performance data collected by the instance monitor.

5.3 Global Request Scheduling

Arrow adopts the minimum-load greedy scheduling strategy by default for request dispatching. The advantage of this greedy strategy is that the scheduling decision for each request only depends on historical requests, without requiring predictions or assumptions about the future request distribution, making it well-suited for online serving scenarios.

Instance Load Evaluation. To use the minimum-load scheduling strategy, the scheduler must be able to evaluate the real-time load of each instance:

- TTFT predictor profiles each instance's prefill processing capability when the cluster is first launched, which measures the TTFT for requests with different input lengths and fits a quadratic curve to model the relationship between TTFT and input length. The global scheduler uses this TTFT predictor to estimate the processing time for prefill requests on each instance during scheduling.
- As Section 4.3 shows that the processing time of the decode phase is generally difficult to predict. Considering that the processing time of each decode iteration is proportional to the total number of tokens in the batch, Arrow uses the total number of tokens of requests currently being processed on the instance as its decode load.

SLO-aware Request Scheduling. Building on the minimum-load scheduling strategy, Arrow further designs its request scheduling strategy to be SLO-aware, meaning that the scheduler considers the real-time TTFT and TPOT of existing requests and makes decisions in conjunction with the SLO targets:

- The scheduling strategy for prefill requests is shown in Algorithm 1. The global scheduler first selects the instance t_1 with the minimum prefill queueing delay from the prefill instance pool, and then checks whether assigning current request to this instance would violate TTFT SLO. If t_1 can satisfy the TTFT SLO, the request is dispatched to t_1. Otherwise, the scheduler repeats the above process in the D→P instance pool to find an instance t_2. If none of the instances can satisfy TTFT SLO, the scheduler will attempt to move some decode instances to the prefill pool. The instance scheduling strategy will be discussed in detail in Section 5.5. If all the above scheduling schemes fail, the scheduler falls back to assigning t_1 as the scheduling result for the request.
- The scheduling strategy for decode requests is shown in Algorithm 2. The global scheduler first checks whether the instance that handled the request's prefill phase has already been reassigned to participate in decode computation. If so, the decode request is dispatched directly to that instance. Otherwise, the scheduler selects the instance t_1 with the smallest total number of running tokens from the decode instance pool, then checks whether the total

Algorithm 1: SLO-aware Prefill Scheduling

Input: Prefill request r, Prefill instances I_P, $D \rightarrow P$ instances $I_{D \rightarrow P}$.
Output: Target instance t.

$t_1 \leftarrow$ Instance $i \in I_P$ with minimal prefill queueing delay;
$t_2 \leftarrow$ Instance $i \in I_{D \rightarrow P}$ with minimal prefill queueing delay;
if $t_1.prefill\_delay + r.prefill\_time \leq$ *TTFT threshold* **then**
  return t_1;
else if $t_2.prefill\_delay + r.prefill\_time \leq$ *TTFT threshold* **then**
  return t_2;
else if *Load of decode instances is low* **then**
  if $(t_3 \leftarrow$ try_move_decode_to_prefill()$) \neq$ *None* **then**
    return t_3;
return t_1;

Algorithm 2: SLO-aware Decode Scheduling

Input: Decode request r, Decode instances I_D, $P \rightarrow D$ instances $I_{P \rightarrow D}$.
Output: Target instance t.

if $r.prefill\_instance \in I_D \cup I_{P \rightarrow D}$ **then**
  return $r.prefill\_instance$;
$t_1 \leftarrow$ Instance $i \in I_D$ with minimal running tokens;
$t_2 \leftarrow$ Instance $i \in I_{P \rightarrow D}$ with minimal running tokens;
if $t_1.running\_tokens \leq$ *Max Running Tokens* **and** $t_1.average\_token\_interval \leq$ *TPOT threshold* **then**
  return t_1;
else if $t_2.running\_tokens \leq$ *Max Running Tokens* **and** $t_2.average\_token\_interval \leq$ *TPOT threshold* **then**
  return t_2;
else if $(t_3 \leftarrow$ try_move_prefill_to_decode()$) \neq$ *None* **then**
  return t_3;
if $t1.running\_tokens \leq t2.running\_tokens$ **then**
  return t_1;
return t_2;

number of tokens and the recent average token generation intervals exceed the predefined thresholds. If they do not, the request is dispatched to t_1. The subsequent steps are similar to those for prefill request scheduling.

There are several noteworthy design considerations in the scheduling strategies described above:

- In the prefill request scheduling strategy, the scheduler prioritizes dispatching requests to prefill instances over D→P instances. This is because ongoing decode requests on D→P instances can cause discrepancies between the predicted and actual TTFT values for prefill requests, and there can be interference between prefill and decode requests on the same instance. Therefore, the scheduler tries to avoid dispatching requests to D→P instances whenever possible. A similar design applies to the decode request scheduling strategy, where the scheduler gives higher priority to decode instances over P→D instances.
- If a prefill instance is reassigned as a decode instance, the scheduler will let this instance continue to process the decode phase of onging prefill requests, in order to eliminate the overhead of KV Cache transmission.
- The Max Running Tokens parameter in Algorithm 2 is determined at cluster startup through profiling. The profiler determines the maximum number of tokens each instance's KV Cache can store, and also tests the maximum number of tokens that can be processed in a single batch under the given TPOT SLO setting. This approach helps prevent unpredictable queuing delays for decode requests due to insufficient memory, thereby reducing the risk of TPOT SLO violations.

5.4 Local Request Scheduling

When a new request arrives, the local scheduler first checks whether KV Cache migration is required. If so, the request is placed in the migration queue and only moved to the request queue after migration completes. The local scheduler adopts a FCFS policy for KV Cache migration, and uses the chunked prefill scheduling strategy [1] for requests: Under a given batch size, decode requests in the queue are prioritized to be included in the running batch. If there is remaining space, chunked prefill requests are added. The chunked prefill strategy enables instances in the P→D or D→P pools to begin processing new type of requests as soon as possible, avoiding the situation where requests queued before instance flipping block the execution of new requests after flipping.

5.5 Instance Scheduling

Instance Scheduling Procedure. Benefiting from the stateless instance and elastic instance pool design, the global scheduler can move instances in or out of any instance pool without incurring additional overhead such as instance restart time. Algorithm 3 describes the process of reassigning a decode instance to a prefill instance: The scheduler first checks that there are at least two instances available to handle decode requests, ensuring that there will always be instances capable of handling decode requests after the adjustment. Then, it selects the instance with the smallest running token count (i.e., the least decode load) from the P→D instance pool. If the P→D pool is empty, it selects from the decode pool. The instance's assignment is then updated according to the instance transition diagram in Figure 5. The process for reassigning a prefill instance to a decode instance, as described in Algorithm 4, is similar: the scheduler selects the instance with the smallest prefill load for adjustment.

SLO-aware Instance Scheduling. The instance scheduling strategy used by Arrow is also SLO-aware:

- Instance scheduling from decode to prefill occurs during the prefill request scheduling process (Algorithm 1): Based on Insights 1 and 2, when the scheduler predicts that the current prefill instances cannot meet the TTFT SLO requirement for a new request, it will attempt to reassign decode instances to the prefill instance pool.

Algorithm 3: try_move_decode_to_prefill

Input: Decode instances I_D, $P \rightarrow D$ instances $I_{P \rightarrow D}$.
Output: New prefill instance p.

```
p ← None;
if |I_D| + |I_{P→D}| > 1 then
    if I_{P→D} ≠ ∅ then
        p ← Instance i ∈ I_{P→D} with minimal running tokens;
    else
        p ← Instance i ∈ I_D with minimal running tokens;
    Move instance p between instance pools;
return p;
```

Algorithm 4: try_move_prefill_to_decode

Input: Prefill instances I_P, $D \rightarrow P$ instances $I_{D \rightarrow P}$.
Output: New decode instance d.

```
d ← None;
if |I_P| + |I_{D→P}| > 1 then
    if I_{D→P} ≠ ∅ then
        d ← Instance i ∈ I_{D→P} with minimal prefill delay;
    else
        d ← Instance i ∈ I_P with minimal prefill delay;
    Move instance d between instance pools;
return d;
```

- Instance scheduling from prefill to decode occurs in the following situations: (1) During the decode request scheduling process (Algorithm 2); (2) When the cluster monitor detects that the average token generation interval of decode instances exceeds the TPOT SLO threshold over a period of time (Insight 3); (3) When prefill instances are idle while decode instances are busy, idle prefill instances are added to decode computation to free up computing resources as quickly as possible in anticipation of potential future bursty traffic.

Scheduling in Overload Scenario. Scenarios in which both prefill and decode tasks are overloaded are not the primary optimization target for Arrow. However, to prevent oscillations in instance allocation caused by prefill and decode tasks competing for resources, Arrow prioritizes allocating compute resources to the decode requests. Specifically, in Algorithm 1, before flipping a decode instance to a prefill instance, the scheduler checks the load on the decode instance. If the decode load is high, the flipping is abandoned. In contrast, when flipping a prefill instance to a decode instance, the scheduler does not check the prefill load. The core rationale for this design is to maximize the number of completed requests in the shortest possible time, and to avoid scenarios where a large number of requests occupy memory resources without progressing beyond the prefill phase. Existing work [33] has also proposed request scheduling schemes for overloaded scenarios, but the design of such schemes is beyond the scope of this paper.

5.6 Summary

We summarize the key design choices in Arrow's scheduling strategies as follows, and highlight how the key insights presented in Section 4 guided the design of these strategies:

- According to Insights 1 and 2, Arrow leverages the strong predictability of TTFT to estimate a request's TTFT before prefill computation begins. It schedules the request to the instance with the minimal expected queueing delay, and determines whether the request might violate the TTFT SLO, allowing the system to promptly allocate more instances to prefill computation when necessary.
- According to Insights 3 and 4, Arrow monitors the token generation interval of each instance to assess the real-time load of decode instances. When the average generation interval exceeds the TPOT threshold, more instances are scheduled to perform decode computation. Due to the non-monotonic nature of TPOT, even though instance scheduling decisions are made based on monitoring metrics with some latency, the system can still effectively satisfy the TPOT SLO.
- According to Insight 5, when bursty traffic arrives, Arrow monitors TTFT predictions during prefill request dispatching to detect rising load on prefill instances. It promptly schedules decode instances with low load to participate in prefill computation. Additional instances are then scheduled for decode computation based on the observed token generation intervals of existing decode instances.

6 Implementation

Arrow is a distributed LLM inference serving system developed based on vLLM [20], and reuses some components from DistServe [52]. The system consists of approximately 3.2K lines of Python, C++, and CUDA code. It includes a RESTful API frontend and the components described in Section 5.

Arrow's frontend provides a set of OpenAI API-compatible interfaces. The cluster monitor is responsible for periodically collecting performance data from each instance. The global scheduler performs request dispatching and instance scheduling based on information including estimated TTFT of new requests and instance metrics. The computing instances are not rigidly assigned prefill or decode roles. KV Cache can be transmitted between any two instances. Instances within the same node utilize cudaMemcpyAsync [25] for KV Cache transmission, whereas instances across different nodes employ NCCL [27]. All components are deployed on a Ray [26] cluster, with inter-component communication handled via Ray's RPC interfaces. Arrow makes minimal modifications to the vLLM [20] inference engine, enabling seamless integration of various optimization techniques from vLLM, including PagedAttention [20], chunked prefill [1], and continuous batching [49].

Table 1. Workloads and SLO settings in evaluation.

Trace	# Requests	TTFT	TPOT
Azure Code	8819	3s	0.1s
Azure Conversation	19366	2s	0.15s
BurstGPT clip	6009	0.25s	0.075s
Mooncake clip	1756	30s	0.1s

7 Evaluation

In this section, we evaluate the performance of Arrow with state-of-the-art PD-collocated and PD-disaggregated serving systems on different real-world workloads and show the effectiveness of its components.

7.1 Experimental Setup

Testbed. We evaluate Arrow on a server with eight NVIDIA H800 80GB GPUs, dual Intel Xeon Platinum 8468V CPUs and 2048GB of host memory. The NVLink bandwidth between two GPUs is 400 GB/s.

Model. We use Llama-3.1-8B [9] as the model in our evaluation. It is a pre-trained model with 128K context length, capable of handling the longest inputs in our datasets.

Workloads. We choose four real-word LLM serving traces as the workload, as shown in Table 1.

- **Azure LLM Inference Traces** [31]: It is a 1-hour serving trace collected from Azure LLM inference services, including both coding and conversation scenarios.
- **BurstGPT** [46]: It is an LLM serving workload with 5.29 million traces from regional Azure OpenAI GPT services over 121 days. We take an 1-hour clip from the original trace for evaluation.
- **Mooncake Conversation Trace** It is an 1-hour conversation trace containing a significant portion of long context requests. Replaying the full trace will exceed the service capacity of all the tested systems, so we only take the first ten minutes of requests for evaluation.

Baselines. We compare Arrow to three baseline systems:

- **vLLM** [20]: vLLM is one of the state-of-the-art LLM serving systems wih PD-collocated architecture. vLLM enables chunked prefill by default and adopts a decode-prioritized scheduling strategy. To fully leverage all GPUs and serve requests with long context, we follow the previous work [47] to set the tensor parallelism to 8.
- **vLLM-disaggregated**: We use vLLM v0.7.3, which introduces Prefill-Decode disaggregated inference as an experimental feature. However, a buffer overflow problem occurs during KV Cache transmission[1] in its default implementation. As this issue remains unresolved, we mitigate it by increasing the buffer size and limiting the batch size[2]. Since PD disaggregation in vLLM v0.7.3 only supports one prefill and one decode instance, we set the tensor parallelism size of each instance to 4 to fully utilize all GPUs.
- **DistServe** [52]: It proposes PD-disaggregation to eliminate prefill-decode interferences. However, due to a lack of long-term maintenance, its performance lags significantly behind existing inference engines. Additionally, it frequently triggers out-of-memory (OOM) errors under high request rates or long input sequences. Therefore, we only evaluate DistServe under low request rate scenarios. Following prior work [47], we allocate four GPUs each for the prefill and decode phases.

[1] https://github.com/vllm-project/vllm/issues/11247

[2] https://github.com/vllm-project/vllm/issues/11247#issuecomment-2652972106

Metrics. We use *SLO attainment* as the major metric. Under a specific SLO setting, we are concerned with the maximum request rate the system can handle. We set the SLO attainment target to 90%, which is a common setting in previous work [31, 47, 52]. We also record the P90 TTFT and P90 TPOT metrics, the 90th percentile of TTFT and TPOT across all requests, to compare the impact of different scheduling strategies adopted by each system on these two metrics.

Evaluation Workflow. We adopt the same evaluation workflow as previous works [32, 46], assessing the performance of different serving systems by replaying service traces. To evaluate system performance under different request rates, we multiply the timestamps by a constant to simulate varying request rates.

7.2 End-to-End Performance

We compare the performance of Arrow with three baselines on four real-world serving traces, are shown in Figure 7. On the Azure Code dataset, which exhibits highly bursty traffic, Arrow leverages the strong predictability of TTFT to proactively allocate more instances to the prefill computation, effectively reducing TTFT SLO violations. Compared to vLLM and vLLM-disaggregated, Arrow achieves 5.62× and 7.78× higher sustainable request rates, respectively. DistServe consistently fails to meet SLO requirements due to its overall lower inference efficiency. Similar results are observed on the BurstGPT dataset, where Arrow achieves 3.60× and 5.04× improvements over vLLM and vLLM-disaggregated, respectively. VLLM-disaggregated fails to complete tests when request rates exceed 25 req/s due to KV Cache transfer issues on the Azure Conversation dataset. Arrow outperforms vLLM and vLLM-disaggregated by 3.76× and 4.06×.

On the Mooncake Conversation dataset which featuring extremely long inputs, vLLM-disaggregated can only complete tests at our minimum request rate. DistServe triggers OOM errors when processing long-context inputs and fails to complete this test. In such long-input, low-rate scenarios, Arrow monitors prefill instance availability after prefill request completion and schedules idle prefill instances to assist with decode computation, quickly freeing up limited computing and memory resources for subsequent requests.

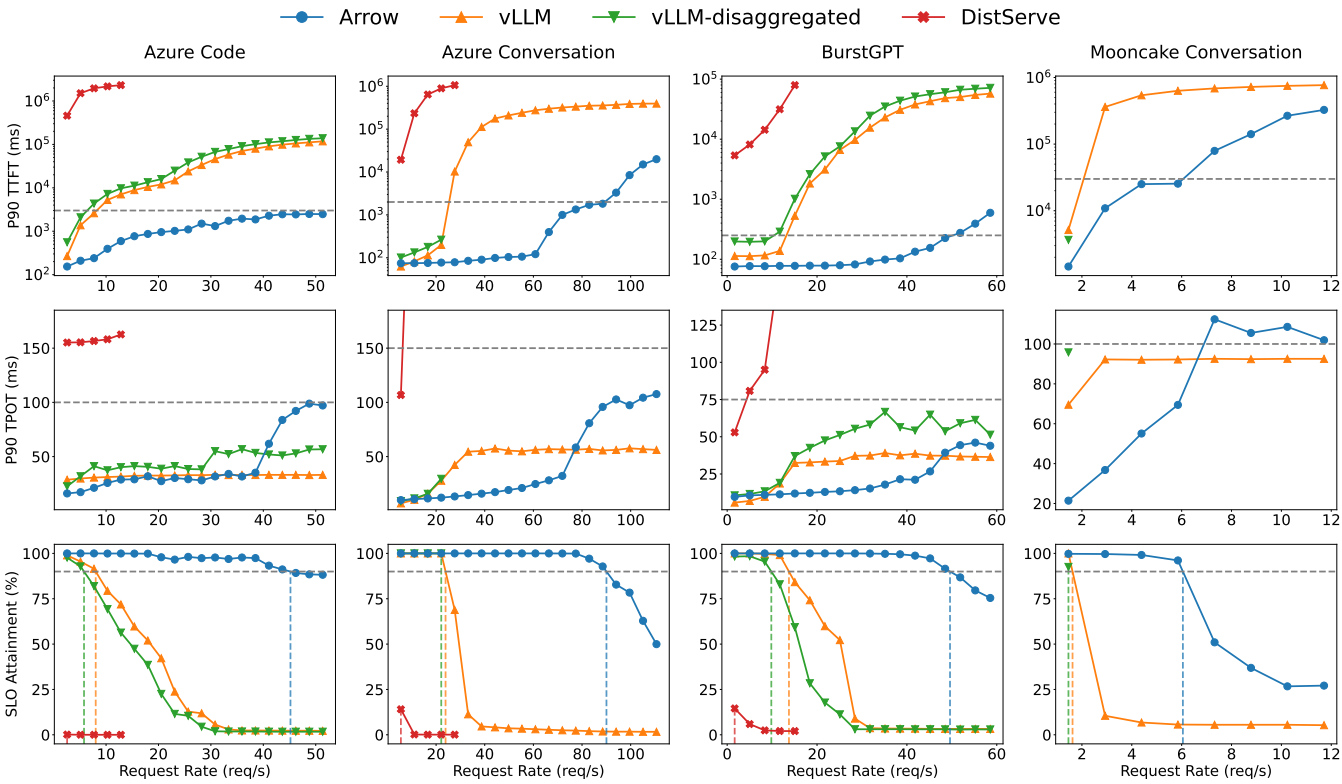

Figure 7. Performance of four LLM serving systems employing Llama-3.1-8B under different traces and request rates.

This approach yields 3.73× and 4.14× improvements over vLLM and vLLM-disaggregated, respectively.

From the first-row TTFT and second-row TPOT in Figure 7, we can observe the scheduling strategy differences between vLLM and Arrow. vLLM employs a decode-prioritized scheduling policy that maintains consistently low TPOT, but causes TTFT to rise rapidly due to extended queuing delay for prefill requests, making it difficult for vLLM to simultaneously satisfy both TTFT and TPOT SLO requirements. In contrast, Arrow adopts an SLO-aware request and instance scheduling strategy that strives to meet both TTFT and TPOT SLO. When both prefill and decode computations become overloaded, Arrow follows the overload scheduling strategy described in Section 5.5, prioritizing decode requests to ensure TPOT SLO satisfaction. As a result, Arrow's TPOT metrics under extremely high request rates consistently remain close to the TPOT SLO setting.

7.3 Ablation Study

In this section, we study the effectiveness of Arrow's adaptive scheduling strategy. We compare the performance of three scheduling strategies: (1) SLO Aware, which is the strategy used by Arrow and includes both request scheduling strategy from Section 5.3 and instance scheduling strategy from Section 5.5; (2) Minimal Load, which only includes the minimum-load request scheduling strategy; and (3) Round Robin.

For the Minimal Load and Round Robin strategies, we use 4 prefill instances and 4 decode instances to process requests. The results are shown in Figure 8. On the Azure Code dataset, which features significant bursty traffic, the SLO Aware strategy used by Arrow achieves 1.67× higher request serving rate compared to the Minimal Load strategy, demonstrating the effectiveness of adaptive instance scheduling. Compared to the Round Robin strategy, the Minimal Load request scheduling strategy achieves up to a 4.3% improvement in SLO attainment, proving that minimal-load scheduling can more closely approximate the optimal scheduling strategy than round-robin approaches. For the Azure Conversation dataset, which exhibits more stable variations in input and output lengths, 4 prefill and 4 decode instances are already sufficient to handle requests. The SLO Aware strategy still achieves 1.1× higher request serving rate than the Minimal Load strategy, serving nearly 10 additional requests per second. In scenarios with relatively balanced request loads, the Minimal Load strategy performs similarly to Round Robin, yet still achieves up to 2.4% improvement in SLO attainment.

7.4 Scalability

We compare the SLO attainment of the SLO Aware and Minimal Load scheduling strategies under varying GPU counts

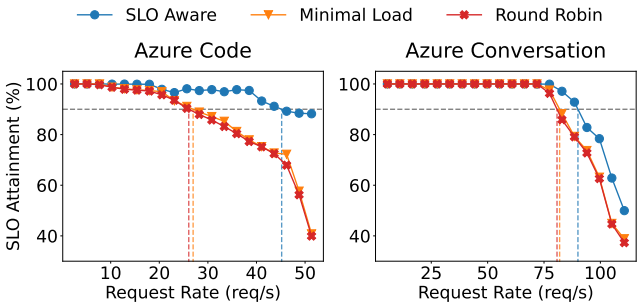

Figure 8. Performance of different scheduling strategies.

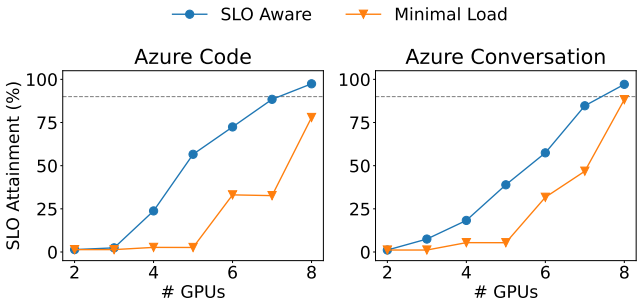

Figure 9. Performance under different number of GPUs.

to evaluate the scalability of Arrow. The results are shown in Figure 9. By employing a flexible instance scheduling strategy, Arrow can fully utilize computational resources to meet the demands of both prefill and decode phases, enabling the serving system to achieve nearly linear improvements in SLO attainment as the number of GPUs increases. In contrast, traditional static PD ratio configurations are prone to hitting either prefill or decode computation bottlenecks under resource constraints, making it difficult to satisfy both TTFT and TPOT SLOs simultaneously. The experimental results demonstrate that our adaptive scheduling strategy exhibits strong universality and scalability, enabling efficient computational resource utilization across different hardware environments to improve SLO attainment.

8 Discussion

In this section, we discuss how Arrow can be extended to large-scale inference clusters and propose several directions for future improvement.

Heterogeneous Cluster Deployment. The adaptive scheduling strategy employed by Arrow can also be applied to inference clusters utilizing heterogeneous hardware and hybrid parallelism strategies. Arrow uses instances, rather than individual GPUs, as the resource unit for monitoring and scheduling. This design allows the global scheduler to make request and instance scheduling decisions without considering the underlying hardware or parallelism strategy used by each instance. Different instances can adopt different deployment methods without affecting the global scheduler's decision-making. During initialization, Arrow also performs profiling at the instance level. For heterogeneous instances, the TTFT predictor can still accurately estimate the computation time of each prefill request on each instance. Therefore, Arrow can leverage heterogeneous hardware to reduce cluster deployment costs and integrate various parallelism techniques to support models with higher computational and memory requirements.

Advanced Scheduling Policy. In distributed serving systems, there are more decision variables that influence scheduling strategies, such as network topology, communication overhead, and state synchronization latency, which increase both the complexity and the optimization space for the scheduler. Arrow provides easily extensible interfaces for implementing scheduling strategies, enabling the integration of various scheduling optimization schemes proposed by existing and future work to accommodate different scenarios and optimization objectives. We also look forward to exploring more efficient scheduling strategies applicable to large-scale PD-disaggregated systems in the future.

9 Related Work

LLM Serving. Existing works have optimized LLM serving systems from multiple perspectives, including kernel [4, 18], KV Cache management [8, 20, 22], and batching strategy [1, 49]. Among these, Orca [49] employs an iteration-level scheduling strategy to reduce request queuing latency, PagedAttention [20] effectively reduces GPU memory fragmentation, and Sarathi-Serve [1] implements chunked prefill to improve compute utilization during the decode phase. These works are orthogonal to our work and have already been integrated into Arrow. To avoid computational interference between the prefill and decode phases, TetriInfer [12], Splitwise [31], and DistServe [52] proposed the Prefill-Decode disaggregated inference architecture. EPD disaggregation [37] further extends this architecture to multi-modal models. However, their static Prefill-Decode ratio configurations are inadequate for handling varying workloads and are prone to SLO violations. In contrast, Arrow proposes an innovative SLO-aware instance scheduling strategy that can effectively improve serving capacity while meeting the given SLO settings.

PD-disaggregation Optimization. As the effectiveness of the PD-disaggregated architecture has been widely validated, numerous optimization efforts have recently emerged. Mooncake [32] and MemServe [13] deploy a distributed KV Cache pool to enable cache reuse. DéjàVu [39] implements a set of high-performance KV Cache streaming APIs to reduce the KV Cache transmission overhead. By simply modifying the KV Cache transmission logic in Arrow, these solutions can be integrated into Arrow to further improve its performance.

Other works have optimized the PD-disaggregated architecture from perspectives including parallelization strategies [47, 52], resource utilization [11, 23, 35], and deployment costs [7]. Our work proposes a simple yet effective instance scheduling strategy that leverages TTFT's strong predictability and TPOT's non-monotonicity to schedule requests and instances, thereby addressing workload diversity.

Request Scheduling. Existing works have optimized request scheduling in LLM serving systems for various objectives, including throughput [3, 48, 49], load balancing [38, 40], and fairness [36]. Recent studies have also proposed diverse request scheduling optimizations for PD-disaggregated architecture, considering aspects such as cache [13, 32], SLO settings [7], and instance load [12, 52]. We design an SLO-aware scheduling strategy based on the minimal-load scheduling policy to enable adpative request dispatching.

10 Conclusion

To address load fluctuations in LLM serving systems, we design Arrow, an efficient and adaptive scheduler that dynamically schedules requests and instances based on cluster load. Arrow employs stateless inference instances and elastic instance pools to enable responsive reconfiguration between prefill and decode instances, while performing adaptive request dispatching and instance scheduling based on SLO settings and real-time performance metrics. Evaluations on multiple real-world datasets demonstrate that Arrow can effectively improve system serving capacity compared to existing solutions.

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)*, 117–134.

[2] Jinze Bai et al. 2023. Qwen Technical Report. (Sept. 28, 2023). arXiv: 2309.16609 [cs].

[3] Ke Cheng, Wen Hu, Zhi Wang, Peng Du, Jianguo Li, and Sheng Zhang. 2024. Enabling Efficient Batch Serving for LMaaS via Generation Length Prediction. In *2024 IEEE International Conference on Web Services (ICWS)*. (July 2024), 853–864. DOI: 10.1109/ICWS62655.2024.00104.

[4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (NIPS '22). (Nov. 28, 2022), 16344–16359.

[5] DeepSeek-AI et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. (Jan. 22, 2025). arXiv: 2501.12948 [cs].

[6] DeepSeek-AI et al. 2025. DeepSeek-V3 Technical Report. Version 2. (Feb. 18, 2025). arXiv: 2412.19437 [cs].

[7] Jiangsu Du, Hongbin Zhang, Taosheng Wei, Zhenyi Zheng, Kaiyi Wu, Zhiguang Chen, and Yutong Lu. 2025. EcoServe: Enabling Cost-effective LLM Serving with Proactive Intra- and Inter-Instance Orchestration. (Apr. 25, 2025). arXiv: 2504.18154 [cs].

[8] Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. 2023. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. In *The Twelfth International Conference on Learning Representations*. (Oct. 13, 2023).

[9] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. (Nov. 23, 2024). arXiv: 2407.21783 [cs].

[10] Connor Holmes et al. 2024. DeepSpeed-FastGen: High-throughput Text Generation for LLMs via MII and DeepSpeed-Inference. (Jan. 9, 2024). arXiv: 2401.08671 [cs].

[11] Ke Hong et al. 2025. Semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage. (Apr. 28, 2025). arXiv: 2504.19867 [cs].

[12] Cunchen Hu et al. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. (Jan. 20, 2024). arXiv: 2401.11181 [cs].

[13] Cunchen Hu et al. 2024. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. (Dec. 21, 2024). arXiv: 2406.17565 [cs].

[14] DeepSeek Inc. 2025. Deepseek. https://chat.deepseek.com/. 2025-05-04. (2025).

[15] OpenAI Inc. 2022. Chatgpt. https://chat.openai.com/. 2025-05-04. (2022).

[16] Perplexity Inc. 2022. Perplexity. https://www.perplexity.ai/. 2025-05-04. (2022).

[17] Yibo Jin et al. 2024. P/D-Serve: Serving Disaggregated Large Language Model at Scale. (Aug. 15, 2024). arXiv: 2408.08147 [cs].

[18] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. 2023. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (ASPLOS 2023). (Jan. 30, 2023), 295–310. DOI: 10.1145/3575693.3575747.

[19] 2025. KV cache strategies. (Mar. 4, 2025). https://huggingface.co/docs/transformers/v4.51.3/kv_cache.

[20] 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 *SOSP '23: 29th Symposium on Operating Systems Principles*. (Oct. 23, 2023), 611–626. DOI: 10.1145/3600006.3613165.

[21] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *Thirty-Seventh Conference on Neural Information Processing Systems*. (Nov. 2, 2023).

[22] Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. 2024. SnapKV: LLM Knows What You are Looking for Before Generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. Vol. 37. (Nov. 6, 2024), 22947–22970.

[23] Yunkai Liang, Zhangyu Chen, Pengfei Zuo, Zhi Zhou, Xu Chen, and Zhou Yu. 2025. Injecting Adrenaline into LLM Serving: Boosting Resource Utilization and Throughput via Attention Disaggregation. (Mar. 26, 2025). arXiv: 2503.20552 [cs].

[24] Yusuf Mehdi. 2023. Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web. The Official Microsoft Blog. (Feb. 7, 2023).

[25] [n. d.] Memory Management. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[26] Philipp Moritz et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 561–577.

[27] [n. d.] NVIDIA Collective Communications Library (NCCL). NVIDIA Developer. https://developer.nvidia.com/nccl.

[28] OpenAI et al. 2024. GPT-4 Technical Report. (Mar. 4, 2024). arXiv: 2303.08774 [cs].

[29] OpenAI et al. 2024. GPT-4o System Card. (Oct. 25, 2024). arXiv: 2410.21276 [cs].

[30] [n. d.] Optimizing Inference on Large Language Models with NVIDIA TensorRT-LLM, Now Publicly Available | NVIDIA Technical Blog.

[31] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. (June 2024), 118–132. DOI: 10.1109/ISCA59077.2024.00019.

[32] 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)*, 155–170.

[33] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2024. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving. Version 3. (July 9, 2024). arXiv: 2407.00079 [cs].

[34] Qwen et al. 2025. Qwen2.5 Technical Report. (Jan. 3, 2025). arXiv: 2412.15115 [cs].

[35] Chaoyi Ruan, Yinhe Chen, Dongqi Tian, Yandong Shi, Yongji Wu, Jialin Li, and Cheng Li. 2025. DynaServe: Unified and Elastic Tandem-Style Execution for Dynamic Disaggregated LLM Serving. (Apr. 12, 2025). arXiv: 2504.09285 [cs].

[36] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 965–988.

[37] Gursimran Singh et al. 2025. Efficiently Serving Large Multimodal Models Using EPD Disaggregation. (Feb. 5, 2025). arXiv: 2501.05460 [cs].

[38] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2024. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. In *The Thirteenth International Conference on Learning Representations*. (Oct. 4, 2024).

[39] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. In *International Conference on Machine Learning*. (July 8, 2024), 46745–46771.

[40] 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)*, 173–191.

[41] Gemini Team et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. (Dec. 16, 2024). arXiv: 2403.05530 [cs].

[42] Gemini Team et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. (June 17, 2024). arXiv: 2312.11805 [cs].

[43] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. (July 19, 2023). arXiv: 2307.09288 [cs].

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*. Vol. 30.

[45] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*, (Nov. 29, 2023).

[46] Yuxin Wang et al. 2024. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. Version 3. (June 17, 2024). arXiv: 2401.17644 [cs].

[47] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (SOSP '24). (Nov. 15, 2024), 640–654. DOI: 10.1145/3694715.3695948.

[48] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Distributed Inference Serving for Large Language Models. (Sept. 25, 2024). arXiv: 2305.05920 [cs].

[49] 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.

[50] Zhihang Yuan et al. 2024. LLM Inference Unveiled: Survey and Roofline Model Insights. (May 1, 2024). arXiv: 2402.16363 [cs].

[51] Ranran Zhen et al. 2025. Taming the Titans: A Survey of Efficient LLM Inference Serving. (Apr. 28, 2025). arXiv: 2504.19720 [cs].

[52] 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)*, 193–210.

[53] Zixuan Zhou et al. 2024. A Survey on Efficient Inference for Large Language Models. (July 19, 2024). arXiv: 2404.14294 [cs].