Prefill-Decode Aggregation or Disaggregation? Unifying Both for Goodput-Optimized LLM Serving

Chao Wang[*]
The Chinese University of Hong Kong

Pengfei Zuo[†]
Huawei Cloud

Zhangyu Chen
Huawei Cloud

Yunkai Liang[*]
Sun Yat-sen University

Zhou Yu
Huawei Cloud

Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

There is an ongoing debate on whether prefill-decode (PD) aggregation or disaggregation is the superior approach for serving large language models (LLMs). This debate has driven optimizations on both sides, each showcasing distinct advantages. This paper presents a comprehensive comparison between PD aggregation and disaggregation, showing that each excels under different service-level objectives (SLOs): PD aggregation is optimal under tight time-to-first-token (TTFT) and relaxed time-per-output-token (TPOT), while PD disaggregation excels under strict TPOT and relaxed TTFT. However, under balanced TTFT and TPOT SLOs, neither approach can deliver optimal goodput.

Based on these insights, this paper proposes TaiChi, an LLM serving system that unifies PD disaggregation and aggregation to achieve optimal goodput under any combination of TTFT and TPOT SLOs. TaiChi leverages a unified disaggregation-aggregation architecture composed of differentiated-capability GPU instances: prefill-heavy instances (fast prefill but high-interference decode) and decode-heavy instances (low-interference decode but slow prefill). It exposes three configurable sliders to control the ratio between prefill-heavy and decode-heavy instances, and the chunk sizes for each. TaiChi adapts to various SLO regimes by adjusting these sliders. When TTFT constraints are tight, TaiChi can be tuned to resemble a PD aggregation configuration; when TPOT dominates, it adapts toward PD disaggregation. Crucially, under balanced SLOs, TaiChi enables a hybrid mode that achieves superior goodput. The key innovation behind this hybrid mode is latency shifting: by selectively reallocating GPU resources from requests that meet TTFT or TPOT SLOs to those at risk of violation, TaiChi maximizes the number of SLO-satisfied requests. This fine-grained, request-level latency shifting is orchestrated through two targeted scheduling mechanisms: flowing decode scheduling to control TPOTs and length-aware prefill scheduling to manage TTFTs, jointly optimizing request assignment. Our extensive experimental results demonstrate that TaiChi improves goodput by up to 77% compared to state-of-the-art systems under balanced TTFT and TPOT SLOs.

Keywords: Large Language Models, LLM Serving, Prefill-Decode, Service Level Objectives, Goodput, Latency Shifting

1 Introduction

Large language models (LLMs) have demonstrated unprecedented performance across various applications, such as personal assistants [4], translation [11, 22], document analysis [15], chatbots [17, 18], and code generators [6, 23]. Serving these LLM applications requires substantial and costly computational resources, particularly GPUs. Consequently, optimizing the LLM serving cost has received significant attention in system research [3, 8, 12, 19, 26, 32, 36].

Meeting service level objective (SLO) constraints is critical for LLM service providers to ensure application-level performance [3, 36]. In LLM serving, a user request is processed in two distinct phases, each with its own SLO constraint [36]. The first phase, known as *prefill*, involves preprocessing the user request and generating the first token with intensive computation. The latency of prefill is constrained by time-to-first-token (TTFT), reflecting system responsiveness [30, 36]. The second phase, known as *decode*, outputs the subsequent tokens autoregressively (i.e., one token per iteration). The average time taken to output a token (except for the first token) is constrained by time-per-output-token (TPOT), which indicates the service speed users can experience [30, 36]. The throughput of requests while meeting both SLO constraints is referred to as *goodput* [30, 36]. Enhancing goodput with the same hardware resources lowers the cost per LLM request [36]. However, application-level SLOs vary widely [3, 9, 33, 36]: some prioritize low TTFT with relaxed TPOT, others require tight TPOT with relaxed TTFT, while many demand a balanced trade-off between the two.

Currently, an active debate is ongoing regarding whether PD aggregation [3] or disaggregation [36] is the superior approach for serving LLMs. PD aggregation (like Orca [32] and Sarathi-Serve [3]) co-locates the prefill and decode phases of a request on the same hardware instance to achieve high resource utilization. In contrast, PD disaggregation (like Splitwise [19] and DistServe [36]) physically separates prefill and decode onto different hardware instances. This approach eliminates interference between the two phases and allows for independent scaling of resources. The debate drives optimizations on both sides, each showcasing the distinct advantages [3, 19, 36].

[*]Work done during their internship at Huawei Cloud.
[†]Corresponding author is Pengfei Zuo (pengfei.zuo@huawei.com).

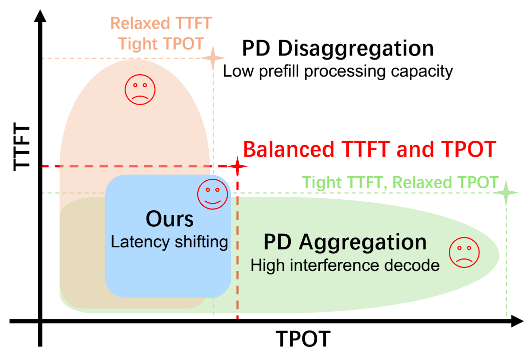


Figure 1. Distribution of requests' TTFT and TPOT under different scheduling approaches, using the same number of compute nodes and QPS. *(PD aggregation performs best when TTFT constraints are tight and TPOT is relaxed, while PD disaggregation excels under tight TPOT and relaxed TTFT. However, under balanced SLO constraints, PD aggregation results in TPOT violations due to high-interference decode, while PD disaggregation leads to TTFT violations due to low prefill processing capacity. By proposing a hybrid-mode inference, we mitigate both issues and achieve better SLO attainment across different combinations of TTFT and TPOT.)*

In this paper, we present a comprehensive comparison between PD aggregation and disaggregation, showing that each approach achieves optimal request goodput under different SLOs: PD aggregation is optimal under tight TTFT and relaxed TPOT, while PD disaggregation excels under strict TPOT and relaxed TTFT, as illustrated in Figure 1. PD aggregation achieves low TTFT by having all instances participate in the prefill phase, but suffers in TPOT due to interference between prefill and decode. In contrast, PD disaggregation improves TPOT by isolating prefill and decode on separate resources, but incurs higher TTFT since fewer instances handle prefill. However, under balanced TTFT and TPOT SLOs, neither approach can deliver optimal goodput. This is because PD aggregation tends to violate TPOT due to decode interference, while PD disaggregation often fails to meet TTFT, as only a subset of instances handle the prefill phase.

To end this debate about these two branches, we propose TaiChi, an LLM serving system that unifies PD disaggregation and aggregation to achieve optimal goodput under any combination of TTFT and TPOT SLOs. TaiChi leverages a unified disaggregation-aggregation architecture composed of differentiated-capability GPU instances: P-heavy instances (fast prefill but high-interference decode) and D-heavy instances (low-interference decode but slow prefill). It exposes three configurable sliders to control the ratio between P-heavy and D-heavy instances, and the chunk sizes for each. TaiChi adapts to various SLO regimes by adjusting these sliders. When TTFT constraints are tight, TaiChi can be tuned to resemble a PD aggregation configuration; when TPOT dominates, it adapts toward PD disaggregation. Crucially, under balanced SLOs, TaiChi enables a hybrid mode that achieves superior goodput. The key innovation behind this hybrid mode is to shift the latency (i.e., TTFT and TPOT) across the prefill and decode phases, as well as across requests. This means that we strategically degrade the latency of requests already meeting SLO constraints, thereby reallocating overprovisioned GPU resources (i.e., GPU time) to prioritize SLO-violating requests through scheduling. However, implementing this request-level latency degradation faces three main challenges:

1) Lack of Architecture Support for Latency Shifting. Existing methods [3, 36] lack architectural support, offering no flexibility in scheduling to reallocate latency across requests. Both PD aggregation and disaggregation approaches rely on uniform GPU instance configurations dedicated either to prefill or decode, preventing differentiated treatment of individual requests. As a result, systems cannot selectively optimize or degrade requests based on their SLO urgency, limiting their ability to shift latency where it is most needed.

2) Request-level TPOT Degradation Hindered by Batching and Output Length Uncertainty. Degrading TPOT at the granularity of individual requests is challenging due to the constraints of batch processing and the unpredictability of output lengths. In batch decode, performance optimizations or degradations applied to one request inevitably affect all co-located requests, regardless of whether they benefit from or can tolerate such changes. This lack of isolation limits the system's ability to selectively degrade TPOT. Furthermore, decode requests with shorter output lengths are more susceptible to prefill-decode interference (as discussed in § 2.5) and should avoid TPOT degradation. However, since the output length of a request is unknown in advance, the scheduler lacks the necessary information to make precise, per-request degradation decisions.

3) Selective TTFT Degradation Constrained by Execution and Queuing Times. Selectively degrading TTFT is complicated by the need to consider both execution and queuing times. Long prefill requests inherently consume more execution time and are more likely to violate TTFT constraints, making them poor candidates for further degradation. Similarly, requests that have already spent considerable time in the queue are at higher risk of SLO violation and should also be protected. Therefore, determining which requests can safely tolerate TTFT degradation requires careful, context-aware scheduling.

Together, these factors hinder fine-grained, request-level latency control in existing LLM serving systems. To address them efficiently, TaiChi introduces the following techniques:

1) Hybrid-Mode Inference. To address Challenge 1, we introduce hybrid-mode inference to enable latency shifting. This approach uniquely combines the advantages of both PD aggregation and disaggregation. To achieve high resource efficiency, it allows all specialized instances—both P-heavy and D-heavy—to process mixed batches containing both prefill and decode tasks, thereby maximizing GPU utilization. Simultaneously, to provide fine-grained control, it adopts the flexibility of disaggregation, allowing the prefill and decode phases of a single request to be executed on different instance types. This decoupling enables strategic latency shifting—the ability to trade latency between the two phases or across different requests. For instance, a request's TTFT can be minimized by processing its prefill on a P-heavy instance, while its decode phase is handled by a D-heavy instance to ensure a low TPOT. This core capability underpins the advanced scheduling techniques that follow.

2) Flowing Decode Scheduling. To address Challenge 2, we propose flowing decode scheduling, which selectively degrades TPOT by dynamically migrating decode requests between D-heavy and P-heavy instances, enabling fine-grained, per-request latency control without cross-request interference. All decode requests are initially assigned to D-heavy instances to prevent unrecognizable short-output requests from completing the decode phase on P-heavy instances and avoid premature TPOT violations. To prevent the degradation of a request's TPOT from impacting others in the same batch, we extract the selected request from its batch in the low-interference (D-heavy) instance and migrate it to a high-interference (P-heavy) instance, thus strategically degrading its TPOT. Since output lengths are unknown a priori, we employ a longest-first approach, which selects the request with the current longest output in the D-heavy instance for degradation, as it has the greatest remaining TPOT budget currently and can better absorb performance degradation. Finally, to prevent over-degradation, we monitor the TPOT of migrated requests in real time. Once the TPOT approaches the SLO constraint, the request is flowed back to a D-heavy instance to preserve service quality.

3) Length-Aware Prefill Scheduling. To address Challenge 3, we propose a length-aware prefill scheduling strategy, which selectively degrades TTFT by assigning short prefill requests to slower instances when doing so does not violate SLO constraints. The key idea is to exploit the lower urgency of short prefill requests by routing them to D-heavy instances, intentionally slowing their execution to free up P-heavy instances for more time-sensitive, long prefill requests. To determine whether a short request is degradable, we estimate its projected TTFT on each D-heavy instance by summing the expected queuing delay and degraded execution time. If this total remains within the TTFT SLO, the request is marked as degradable and scheduled accordingly.

We have implemented TaiChi on vLLM [28] and plan to open-source it in the near future. Experiments show TaiChi improves goodput by up to 77% over SOTA systems. It also reduces TTFT by up to 13.2× and TPOT by up to 1.69×, relative to PD disaggregation and PD aggregation, respectively.

The main contributions of this paper are as follows:

1. We identify a fundamental trade-off in existing systems between optimizing TTFT and TPOT, which limits overall goodput.
2. We propose the TaiChi, a unified LLM serving system that leverages a hybrid aggregation-disaggregation architecture and latency-shifting scheduling policies to resolve this trade-off.
3. We demonstrate the advantages of TaiChi through comprehensive experiments.

2 Background and Motivation

2.1 LLM Inference

Transformer Architecture. The popular LLMs such as GPT-4 [18] and LLaMA [27] are built upon decoder-only transformer models, which are optimized for next-token prediction [3]. These models consist of a stack of identical layers, each including a self-attention mechanism and a feed-forward network (FFN). In each layer, the self-attention module computes contextualized token embeddings by attending over all previous tokens. This involves computing query (Q), key (K), and value (V) vectors and applying scaled dot-product attention. The resulting vector is passed through an FFN block to produce the output embedding for the next layer. Notably, the K and V vectors are cached during decode to avoid recomputation, forming the key-value (KV) cache used for efficient generation.

Two-Phase Inference. LLM inference proceeds in two distinct stages: the *prefill* phase processes the full prompt in parallel to generate the first token, and the *decode* phase generates subsequent tokens one by one. Prefill is compute-intensive, leveraging the parallelism of the transformer to fully utilize the GPU across the input sequence. In contrast, decode is memory-bound and sequential, as it processes one token at a time using cached key/value vectors of the previous tokens. This asymmetry leads to under-utilization of compute during decode.

Batching. To improve GPU utilization, LLM serving systems batch multiple inference requests, particularly for the decode phase. Batching amortizes model loading costs and maximizes throughput, especially during decode, where token-wise generation is lightweight. However, batching heterogeneous requests introduces latency variability. Fixed-size request-level batching is simple but inefficient, as longer requests delay batch progress. To solve it, continuous batching [32] is proposed to allow requests to enter and exit batches dynamically, which improves GPU occupancy.

Performance Metrics. Latency service-level objectives (SLOs) quantify user-perceived performance by specifying bounds on time-to-first-token (**TTFT**) and time-per-output-token (**TPOT**). TTFT measures the latency from request arrival to the first token, primarily determined by prefill time, while TPOT reflects the average per-token latency during decode. Satisfying these SLOs is essential for interactive

Table 1. A comparison of different scheduling approaches.

Scheduling Policy	Batch	Request
PD Aggregation [3, 32]	Aggregated	Aggregated
PD Disaggregation [19, 36]	Disaggregated	Disaggregated
Hybrid Mode	**Aggregated**	**Disaggregated**

responsiveness and smooth token streaming [30]. **Goodput** denotes the maximum request rate that can be sustained while meeting SLO targets [36]. This metric is critical as it influences serving costs: increasing goodput on fixed hardware lowers the cost per query.

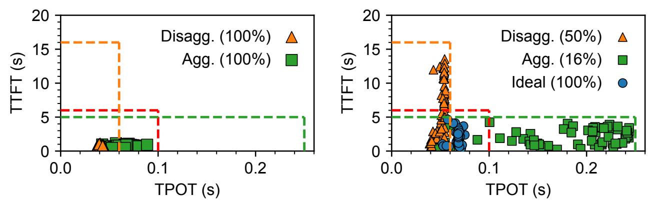
2.2 Scheduling Policies for LLM Serving

The scheduler determines how requests are batched and assigned across the LLM serving instances. Modern policies can be grouped into two main categories: *prefill-decode (PD) aggregation* and *PD disaggregation*, depending on whether the two phases of the requests share the same hardware instance.

PD Aggregation. Most existing systems, including Orca [32], and Sarathi-Serve [3], colocate prefill and decode on the same GPU instance for high resource utilization. Orca improves utilization with iteration-level batching, where requests may join or leave the batch after each iteration. However, a new prefill request can dominate a full iteration, potentially delaying ongoing decode tasks for an extended period. To address this, Sarathi-Serve proposes *chunked prefill*, which divides prefill into smaller chunks that are piggybacked in decode batches as additional computation. This piggybacking approach improves compute resource utilization during decode [3].

PD Disaggregation. Recent systems such as Splitwise [19] and DistServe [36] physically separate prefill and decode across different hardware instances, eliminating prefill-decode interference and enabling independent scaling. After the first token is computed, the KV-cache is transferred from the prefill to the decode instance via high-speed interconnects. Advances in interconnects (e.g., inter-GPU NVLINK at 600 GB/s, inter-node InfiniBand at 800 Gbps) and memory-efficient attention mechanisms (e.g., GQA [5], MLA [7]) have made this transfer overhead negligible [36]. As a result, PD disaggregation offers greater scheduling flexibility and higher goodput under strict TPOT constraints.

As outlined in Table 1, these scheduling policies can be analyzed in two key dimensions: batch handling and request handling. Batch handling determines whether a batch mixes prefill and decode computations (aggregated) or is specialized for a single phase (disaggregated). An aggregated batch can improve GPU utilization, while a disaggregated batch eliminates interference between phases. The request handling defines whether a request's prefill and decode phases are treated as a single scheduling unit (aggregated) or can be separated across different GPU instances (disaggregated). An aggregated request is simpler to schedule, whereas a disaggregated one provides greater scheduling flexibility.



(a) Low QPS level (QPS=6) **(b)** High QPS level (QPS=12)

Figure 2. TTFT and TPOT request distributions for different approaches across varying QPS levels, under multiple SLO constraints (orange for relaxed TTFT and tight TPOT, green for tight TTFT and relaxed TPOT, red for balanced SLOs). The attainment rates (%) under balanced SLOs are in parentheses.

Table 2. SLO attainment rates of different scheduling approaches under varying TTFT and TPOT SLOs (QPS=12).

TTFT & TPOT SLOs	PD Aggregation	PD Disaggregation
Relaxed TTFT & Tight TPOT (16s, 60ms)	7%	98%
Tight TTFT & Relaxed TPOT (5s, 250ms)	97%	42%
Balanced TTFT & TPOT (6s, 100ms)	16%	50%

2.3 Dilemma of Existing Methods

The improvement of goodput in LLM serving systems is constrained by two SLO constraints: TTFT and TPOT. Optimizing for only one SLO constraint may cause the other SLO constraint to become the bottleneck for improving goodput. Our investigation of PD aggregation and disaggregation reveals the following dilemma:

Observation 1: *PD aggregation performs best under tight TTFT and relaxed TPOT constraints, while PD disaggregation is more effective under tight TPOT and relaxed TTFT. However, when TTFT and TPOT constraints are balanced, both approaches struggle to meet SLOs effectively.*

To investigate their performance characteristics, we conduct experiments using Vidur[1]. Figure 2 presents the TTFT and TPOT distributions for individual requests under both PD aggregation and disaggregation at different query per second (QPS) levels. As the load increases (increasing QPS

[1]Vidur is an LLM inference simulator [2] that emulates kernel latency with high accuracy (<3% error), providing rich performance insight. All experiments in this section are performed on a 4-node, 8-GPU A100-DGX cluster, deploying the Llama-2-70B model [27] with 4-way tensor parallelism (TP4). The Arxiv summarization dataset [1] is used, limiting requests to under 4096 tokens to fit the model's context window.

from 6 to 12), PD disaggregation exhibits a significant elongation of TTFT, whereas PD aggregation shows a considerable increase in TPOT. To quantify this performance degradation, we evaluate the SLO attainment rate for both schemes under high load (QPS=12) against three distinct sets of SLO constraints, as detailed in Table 2. Under a relaxed TTFT (e.g., 16 s) and a tight TPOT (e.g., 60 ms) constraint, PD disaggregation performs admirably, achieving a 98% SLO attainment rate, while PD aggregation only reaches 7%. Conversely, with a tight TTFT (e.g., 5s) and a relaxed TPOT (e.g., 250ms), PD aggregation achieves a high attainment rate of 97%, whereas PD disaggregation only attains 42%. These findings indicate that both PD disaggregation and aggregation excel only when one SLO metric is strictly constrained while the other is relaxed. However, under moderately balanced dual SLO constraints (TTFT=6s, TPOT=100ms), the SLO attainment rates for PD disaggregation and aggregation drop to 50% and 16%, respectively. This demonstrates that neither approach can effectively satisfy balanced SLO requirements. Considering that both TTFT and TPOT are crucial to user experience, impacting perceived responsiveness and fluency, it is imperative to conduct further analysis and optimization to improve the goodput in scenarios with balanced SLOs.

In the rest of this section, we further investigate the underlying causes of PD aggregation's struggle to meet TPOT constraints (§ 2.3.1) and PD disaggregation's challenges in satisfying the TTFT constraints (§ 2.3.2).

2.3.1 TPOT Bottleneck of PD Aggregation. PD aggregation demonstrates excellent TTFT but suffers from high TPOT. To investigate the underlying causes of this TPOT degradation, we conduct a series of diagnostic experiments and obtain *Observation 2*.

Observation 2: *The high TPOT in PD aggregation arises from prefill interference due to computation-bound linear operations, with a strong linear relationship between interference intensity and TPOT.*

Figure 3 shows the temporal breakdown of batch execution time with varying chunk sizes. As the chunk size increases, the total execution time increases. This is because larger chunk sizes introduce more prefill tokens into the batch, which leads to increased time spent on linear operations (i.e., matrix multiplications) associated with prefill.

To better understand the relationship between TPOT and prefill-decode interference, we perform a quantitative analysis. Before that, we define *interference intensity* as the ratio of total prefill tokens computed during a decode request to its output length, measured in *prefill tokens per output token.* For example, if a decode request generates 100 output tokens but 50,000 prefill token computations occur concurrently, its interference intensity is 500 (50,000/100) prefill tokens per output token. Figure 4 reveals a strong linear correlation between TPOT and interference intensity, as evidenced by the fitted line's R-squared value of 0.99. The slope of the

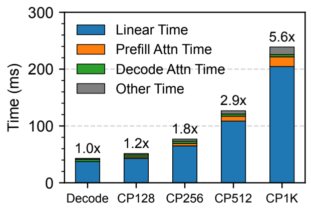


Figure 3. Breakdown of batch execution time with varying chunk sizes (batch size = 16). *CPxxx* refers to Chunked Prefill with a chunk size of xxx.

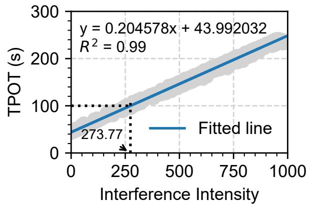


Figure 4. Scatter plot of the requests' TPOT and their suffered interference intensity (CP1024), with a fitted line (R^2 = 0.99, strong correlation).

fitted line's equation represents the increase in TPOT per additional token of interference intensity (here 0.2 ms). The intercept indicates the decode time in the absence of interference (here 44 ms). Furthermore, Figure 4 suggests that the key to controlling TPOT lies in regulating interference intensity. For instance, if we aim to keep TPOT below 100 ms, we must limit interference intensity to fewer than 273.77 prefill tokens per output token.

Although smaller chunk sizes reduce interference by limiting the maximum interference per decode token (or batch), they are not always optimal. As shown in Figure 5, chunk sizes below 1024 (e.g., 128, 256, 512) constrain TPOT but result in prohibitively high TTFT (analyzed later in § 2.3.2), making the system unsustainable for the workload. Thus, the optimal configuration should adopt the smallest chunk size that still satisfies the TTFT constraint.

2.3.2 TTFT Bottleneck of PD Disaggregation. While PD disaggregation achieves high TPOT, it is easy to violate the TTFT SLO constraint. Through systematic experiments, we analyze the root cause of elevated TTFT in PD disaggregation and have *Observation 3*.

Observation 3: *The high TTFT in PD disaggregation stems from request queuing, which occurs because PD disaggregation offers lower prefill processing capacity compared to PD aggregation.*

Figure 6 shows the performance distribution of TTFT and TPOT in PD disaggregation with different PD ratios, compared with those of PD aggregation (CP1024). Experimental results demonstrate that when the PD ratio is adjusted from 4:4 to 7:1, TTFT exhibits a non-monotonic trend of initial decrease followed by an increase. In all configurations, PD disaggregation results in higher TTFT than PD aggregation.

Figure 7 further breaks down the p90 TTFT, revealing that queuing time (including both prefill and decode queues; note that decode queuing time is included in TTFT, as users experience it only once, following the same measurement as vLLM [28]) dominates TTFT in PD disaggregation. Significant queuing times indicate the system has surpassed

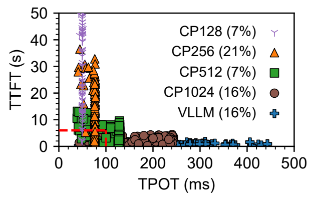


Figure 5. Latency distribution under varying configurations of PD aggregation (QPS=12).

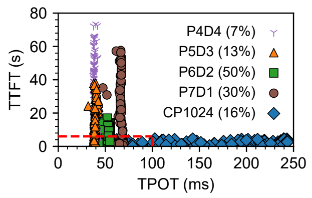


Figure 6. Latency distribution under varying configurations of PD disaggregation (QPS=12).

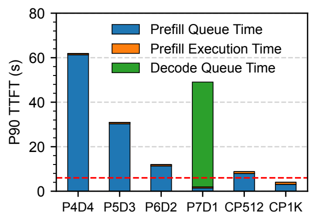


Figure 7. Breaking down the P90 tail TTFT of PD disaggregation (PxDy) and PD aggregation (CPxxx).

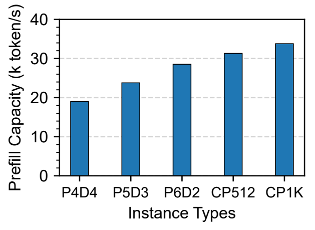


Figure 8. Prefill processing capacity of different configurations of instances.

its processing capacity: high prefill queue times result from insufficient prefill processing resources, while decode queue latency reflects inadequate memory in decode instances. Notably, PD ratios that cause queuing in the decode queue should be excluded, since decode requests generally take much longer to execute than prefill requests.

To identify the root cause of prefill queuing, we quantified the system's prefill processing capacity, defined as the number of prefill tokens computed per second under a given workload. Figure 8 compares the profiled prefill processing capacities of existing methods with a batch size of 16 and a prompt length of 3,000. We observe that increasing prefill processing capacity leads to a shorter prefill queuing time, from Figures 7 and 8. Specifically, as the PD ratio increases (from 4:4 to 6:2), the prefill processing capacity improves, while the prefill queuing time decreases. However, the maximum prefill processing capacity achieved by PD disaggregation remains lower than that of PD aggregation. This is because, in PD disaggregation, only a subset of instances can contribute to prefill processing capacity, whereas in PD aggregation, all instances are capable of handling prefill tasks. Additionally, it is worth noting that in PD aggregation, a larger chunk size results in higher prefill processing capacity. This is because computing the same number of prefill tokens requires approximately twice as many iterations for CP512 compared to CP1024, during which roughly twice the number of decode tasks are executed, thereby slowing down the prefill execution speed. This also explains why CP1024 exhibits better TTFT than CP512 in Figure 5.

2.4 Motivations

To address the dilemma faced by existing methods under balanced TTFT and TPOT SLOs, we propose a *latency-shifting scheduling* paradigm to mitigate the limitations of these two mutually exclusive methods (i.e., PD aggregation and PD disaggregation). The key idea behind it is to shift the latency (i.e., TTFT and TPOT) across prefill/decode phases and across requests. Specifically, by shifting the latency of requests exceeding SLO constraints to those that significantly satisfy the SLO requirements, we maximize the number of requests that meet the SLO constraints and thereby improve the goodput. For example, when certain requests show TPOT exceeding constraints (potentially due to the prefill-decode interference), we shift the surplus TPOT to requests with well-satisfied TTFT or TPOT, ensuring all these requests comply with SLO constraints and enhancing overall goodput.

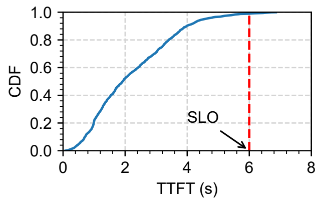
Opportunity 1: *The well-satisfied latency of existing methods performs well, leaving substantial room to accommodate shifted latency.*

While existing methods excel in only one latency metric (either TTFT or TPOT), their strong performance in their respective domains suggests potential for latency shift. As shown in Figure 9a, over 75% of requests in PD aggregation achieve a TTFT less than 60% of the SLO constraint. Similarly, Figure 9b indicates that 100% of requests in PD disaggregation achieve a TPOT below 60% of the SLO constraint. Building on this finding, if requests exceeding SLO constraints can be shifted to these requests with good latency, a significant improvement in goodput can be anticipated.

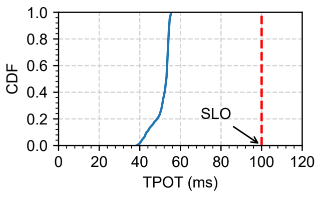
Opportunity 2: *Lantecy (TTFT and TPOT) can be shifted across phases by scheduling resources.*

The scheduling policy prioritizes requests for GPU resource (i.e., GPU time) allocation, allowing prioritized requests to be executed first and reducing their latency. If a request prioritized for resources is in the prefill (or decode) stage, it will reduce TTFT (or TPOT). For example, PD aggregation reduces prefill latency by reserving resources for processing prefill tokens in each iteration using a large chunk size. In contrast, PD disaggregation improves decode latency by allocating all instance resources exclusively to the decode phase. Notably, if certain requests significantly meet their SLO constraints, it indicates resource over-provisioning for those requests. Conversely, the latency of requests that are not prioritized for resource allocation will be degraded, thus achieving latency shifting.

Existing methods can implement latency shifting across the prefill and decode phases. For example, increasing the chunk size in PD aggregation demonstrates that TTFT can be shifted to TPOT. This is because increasing the chunk size in PD aggregation will allocate a greater portion of GPU



(a) TTFT CDF of PD aggregation (CP1024).



(b) TPOT CDF of PD disaggregation (P6D2).

Figure 9. Opportunity to shift latency.

time in each iteration to prefill computation (see Figure 3). This enhances prefill processing capacity (see Figure 8) and optimizes TTFT (see Figure 5). However, this also reduces the portion of GPU time allocated to decode requests, thereby degrading the TPOT of requests (see Figure 5). In contrast, compared to PD aggregation, reserving GPU resources for decode requests, as done in PD disaggregation, allows TPOT to be shifted to TTFT. This prioritization ensures that decode requests obtain resources first, enhancing TPOT at the expense of TTFT. It is worth noting that PD disaggregation cannot flexibly shift TTFT to TPOT, because the disaggregated decode monopolizes GPU resources, preventing further degradation of TPOT.

In summary, requests prioritized for resource scheduling experience better latency, whereas those scheduled later experience degraded latency. This resource scheduling makes latency shifting possible.

2.5 Challenges

However, cross-phase latency shifting alone is insufficient to achieve optimal goodput, as the latency degradation tolerance varies from request to request. The key requirement is to enable latency shifting at the request level, which introduces the following challenges:

Challenge 1: *Existing methods lack architectural support for request-level latency shifting.* Existing methods—whether based on PD aggregation or PD disaggregation—employ uniform GPU instance configurations dedicated either to prefill or decode, resulting in identical service levels for requests across instances. This architectural homogeneity precludes request-level latency optimization or degradation based on their SLO urgency through scheduling, as requests cannot be treated differently.

Specifically, in PD aggregation, all instances share the same configuration. Unifying the chunk size across all instances is both reasonable and efficient. This is because the TPOT upper bound is determined by requests served in the instance with the largest chunk size, where the prefill-decode interference is highest. If the TPOT of these requests meets the SLO constraints, using smaller chunk sizes in other instances offers no additional benefit. However, this uniform configuration prevents architectural support for request-level latency degradation: increasing or decreasing the chunk size affects the TPOT of all requests uniformly. Similarly, in PD disaggregation, the prefill instances (affecting TTFT) and decode instances (affecting TPOT) are configured uniformly, respectively. This results in identical service levels of TTFT and TPOT for all requests, leaving no flexibility for scheduling adjustments.

Challenge 2: *Request-level TPOT degradation is hindered by batch processing and output length uncertainty.* First, batch processing makes request-level TPOT degradation seemingly impossible, as the degradation must occur at the batch level. Batch processing is a crucial optimization that improves throughput and resource utilization by allowing multiple requests to share the overhead of model loading. However, this shared processing creates interdependence, where a performance change intended for one request inevitably affects all co-located requests, regardless of whether they benefit from or can tolerate such changes. For example, using chunked prefill with a larger chunk size can shift the TTFT of a prefill request to the TPOT of a decode request that can tolerate increased TPOT. Nevertheless, this approach may also cause unintended TPOT degradation for other decode requests within the batch that could not tolerate such degradation. This lack of isolation fundamentally limits the system's ability to selectively degrade TPOT at the request level.

Second, the unpredictability of output lengths prevents the system from identifying which decode requests are safe and suitable to degrade. Short-output requests are more vulnerable to PD interference, leading to excessive TPOT degradation, because the output length acts as the denominator in calculating interference intensity (as defined in § 2.3.1). This effect is evident in our experiments, as shown in Figure 10. Consequently, it is necessary to control TPOT degradation for requests with short output lengths. However, the output lengths cannot be predetermined until the *end-of-sequence* token is generated in auto-regressive models. This uncertainty impedes the determination of which requests to degrade and by how much, making it challenging for the scheduler to proactively perform request-level TPOT degradation throughout the decode process. Although some [10, 21, 35] works have attempted to predict output lengths, achieving high prediction accuracy across all datasets remains very difficult; for example, these works can only reach an accuracy of 60%–81% on certain datasets. An x% prediction error rate can result in an equivalent x% decrease in SLO attainment, thereby reducing the cost efficiency of the LLM serving system. Therefore, output length prediction with insufficient accuracy is not suitable for deployment in a production LLM serving environment.

Challenge 3: *Request-level TTFT degradation is non-trivial, as it requires jointly considering both the request's execution and queuing time.* Long prefill requests inherently have longer execution times and are more likely to violate TTFT

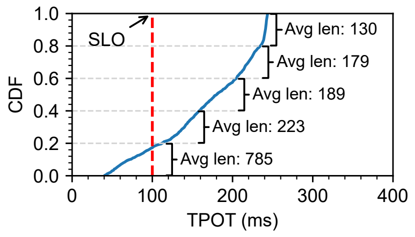


Figure 10. Relationship between TPOT and decode length for CP1024.

constraints, making them unsuitable for further degradation. Recent studies [29, 34] show a broad distribution of prefill lengths: from very short to very long. For example, assuming prefill lengths mostly range from 2k to 16k tokens (as in the Arxiv summarization dataset [1]) and a prefill throughput of 5k tokens per second (consistent with Figure 8), the prefill execution time varies from 0.4s to 3.2s. If the TTFT SLO is set to 3.5s, longer requests already approach this limit, necessitating the avoidance of TTFT degradation. Conversely, short requests are more suitable for accepting shifted TTFT because they can tolerate slower execution. Analogously, requests that have already endured long queuing delays have consumed a significant portion of their TTFT budget. This leaves them with little tolerance for additional execution latency, making it crucial to protect them from further performance degradation. Overall, implementing TTFT degradation for individual requests is not straightforward.

3 The Design of TaiChi

In this section, we first present the architectural overview of TaiChi, which unifies PD aggregation and disaggregation within a single flexible framework (§ 3.1). We then introduce *hybrid-mode inference* (§ 3.2), a capability unique to TaiChi that enables latency shifting to achieve superior goodput. Finally, we describe two targeted scheduling mechanisms that support hybrid-mode inference: *flowing decode scheduling* for controlling TPOT (§ 3.3) and *length-aware prefill scheduling* for managing TTFT (§ 3.4).

3.1 Architectural Overview

We present TaiChi, an LLM serving system that unifies PD disaggregation and aggregation to achieve goodput-optimal performance under any combination of TTFT and TPOT SLOs. TaiChi is built on a unified aggregation-disaggregation architecture composed of differentiated-capability instances, i.e., P-heavy and D-heavy instances. On top of this architecture, TaiChi exposes three configurable sliders to control the ratio between prefill-heavy and decode-heavy instances, and the chunk sizes for each. By adjusting these sliders, TaiChi can dynamically adapt to a wide range of SLO regimes.

Differentiated-Capability Instances. As illustrated in Figure 11, TaiChi comprises a proxy and multiple inference instances. The proxy orchestrates the execution of prefill and

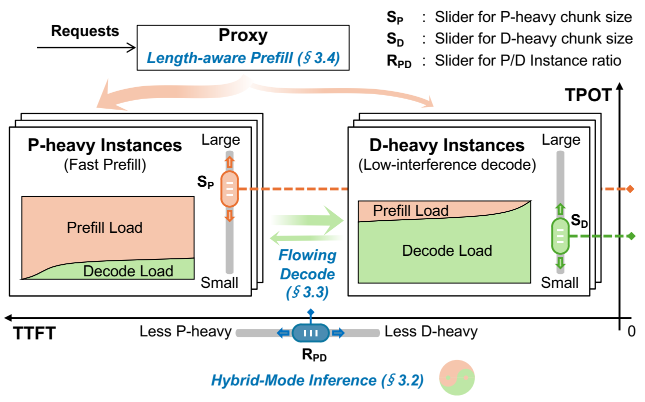


Figure 11. The system overview of TaiChi. The system configures instances as either prefill-heavy or decode-heavy using different chunk sizes, each offering differentiated capability. A length-aware proxy routes requests to an appropriate instance for their prefill phase (either fast or degraded). The subsequent decode execution then "flows" between the two instance types to dynamically manage the TPOT (optimized or degraded).

decode tasks by dispatching them to appropriate instances based on their capabilities. The inference instances in TaiChi are divided into two types:

- *P-heavy* instances are optimized for prefill tasks. They are configured with *larger chunk sizes*, enabling them to efficiently process compute-intensive prefill workloads and minimize TTFT. However, when handling decode tasks, they suffer from *high prefill-decode interference*, leading to higher TPOT.
- *D-heavy* instances are optimized for decode tasks. Configured with *smaller chunk sizes*, they provide low-interference decode, thereby reducing TPOT. Although slower at prefill, D-heavy instances can still handle certain degradable prefill tasks, improving utilization over pure PD disaggregation.

This capability differentiation—achieved purely through chunk size configuration—allows TaiChi to flexibly combine the strengths of PD aggregation and disaggregation within a single unified framework.

Configurable Sliders. TaiChi introduces three sliders to navigate the PD design space:

- R_{PD}: the ratio of P-heavy to D-heavy instances.
- S_P: the chunk size for executing chunked prefill in P-heavy instances.
- S_D: the chunk size for executing chunked prefill in D-heavy instances.

Larger chunk sizes improve an instance's prefill throughput, which benefits TTFT, but also increase prefill-decode interference, thereby degrading TPOT (§ 2.3.1). Similarly, increasing R_{PD} enhances the system's overall prefill processing capacity by allocating more P-heavy instances, but reduces the availability of decode resources (§ 2.3.2).

By adjusting these sliders, TaiChi tunes the system's latency profile to meet different combinations of TTFT and TPOT SLOs. The optimal configuration for a given workload and SLO can be determined via offline search, following approaches from prior work [3, 19, 36]. For example, under a strictly tight TPOT constraint, TaiChi can be configured as a pure PD disaggregation system by setting S_D to exclude the prefill tokens and assigning S_P to the maximum content length, effectively disabling chunked prefill. An appropriate R_{PD} is then selected based on the workload characteristics. Conversely, when TTFT is the primary constraint, TaiChi can operate as a pure PD aggregation system by setting $S_D = S_P$ to a common chunk size across all instances. Under balanced TTFT and TPOT constraints, TaiChi provides a *hybrid mode* that enables latency shifting to achieve optimal goodput, as described in the next subsection. To adapt to dynamic workloads, our system adopts an on-demand search-and-reconfigure strategy like DistServe, which triggers a new search and reconfiguration only upon significant workload changes. This reconfiguration completes in minutes and is far shorter than the typical hourly-scale shifts in workload patterns [36].

3.2 Hybrid-Mode Inference

To support balanced TTFT and TPOT SLOs and maximize goodput, TaiChi introduces *hybrid-mode inference*, which enables *latency shifting*—strategically redistributing latency (i.e., TTFT and TPOT) both across the prefill and decode phases and across requests—by leveraging the differentiated capabilities of instances in TaiChi (§ 3.1).

Unlike traditional PD aggregation or disaggregation, hybrid-mode inference uniquely combines two complementary scheduling principles: *aggregated batch handling* for high resource efficiency and *disaggregated request handling* for fine-grained latency control, as summarized in Table 1. This design enables TaiChi to balance latency and throughput under diverse SLO regimes.

Aggregated Batch Handling for High Utilization. Inspired by PD aggregation, this hybrid mode allows both P-heavy and D-heavy instances to process mixed batches containing both prefill and decode tasks. By enabling all instances to contribute to prefill processing and piggyback decode requests with chunked prefill, this approach boosts the system's total prefill throughput and improves the overall GPU utilization.

Disaggregated Request Handling for Fine-Grained Control. In line with PD disaggregation, this hybrid mode allows the prefill and decode phases of a single request to be executed on different instances. This enables fine-grained, per-request latency optimization or degradation. For example, a request's prefill phase can be routed to a P-heavy instance to ensure low TTFT, while its decode phase can be assigned to a D-heavy instance for low TPOT. Conversely, a request can be deliberately degraded by assigning its prefill to a D-heavy instance and its decode to a P-heavy instance, freeing specialized resources for latency-critical requests. This mechanism forms the core of TaiChi 's latency shifting capability.

In effect, hybrid-mode inference unifies the scheduling flexibility of PD disaggregation with the high utilization efficiency of PD aggregation. More importantly, it provides the necessary foundation for the latency-shifting scheduling strategies described in § 3.3 and § 3.4.

3.3 Flowing Decode Scheduling

To enable request-level TPOT degradation, we introduce *flowing decode scheduling* for fine-grained, per-request latency control. The core mechanism involves dynamically migrating decode requests between D-heavy instances and P-heavy instances. This migration allows the system to intentionally and selectively degrade the TPOT of certain requests, thereby reallocating over-provisioned resources to other requests that require lower latency, without the constraints of batch processing or the need for pre-determined output lengths. In contrast to existing methods, our approach avoids the consistent high interference characteristic of PD aggregation while enabling the dynamic TPOT degradation that PD disaggregation lacks. As illustrated in Figure 12, the flowing decode process encompasses the following three key stages.

① **Low-Interference Decode Init:** After a request completes its prefill phase, it is initially scheduled to a D-heavy instance to begin low-interference decode and prevent premature TPOT violations. This strategy is necessary because if a short-output request (e.g., producing only 2 output tokens, but this is unknown a priori) begins decode on a P-heavy instance with high interference, it may complete decode there and violate its TPOT constraint. The selection of the initial D-heavy instances considers both load balancing and the minimization of KV cache transfers. If the prefill of a request is executed on a P-heavy instance, the proxy schedules it to the D-heavy instance with the lowest decode load (i.e., HBM usage). Conversely, if the prefill stage of a request is executed on a D-heavy instance, it will perform the in-place decode to minimize KV cache transfers between instances.

② **Longest-First Degradation Flowing:** When the HBM capacity of D-heavy instances reaches a predefined memory watermark M (e.g., 95% utilization), we selectively offload a portion of decode requests to P-heavy instances, thereby degrading their TPOT to release GPU resources for other requests. The selection of requests for degradation is strategic: to avoid penalizing interference-sensitive short-output jobs, whose lengths are unknown a priori (Challenge 2), we innovatively prioritize offloading requests with the current longest output. These requests are ideal candidates for offloading, as they have already benefited from numerous iterations on low-interference D-heavy instances and can

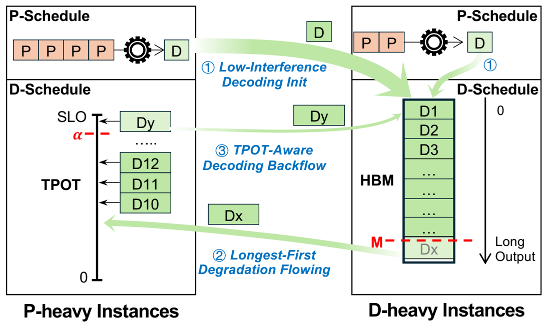


Figure 12. An illustration of flowing decode scheduling. The scheduler manages TPOT by ① starting decode requests on D-heavy instances, ② offloading the longest ones to P-heavy for TPOT degradation, and ③ returning any that approach their TPOT SLO.

Algorithm 1 Decode Scheduling Algorithm in Instances

```
    Input: Decode request set $S$, TPOT SLO $\tau_{tpot}$, Current memory usage $m$, Instance type $type$, Approach factor $\alpha$, Memory watermark $M$
    Output: Optimizing set $O$ or Degrading set $D$
1:  if $type$ is P-heavy then
2:      $O \leftarrow \{r \in S \mid r_{\text{tpot}} > \tau_{tpot} * \alpha\}$    ▷ Approaching SLO
3:      return $O$
4:  else if $type$ is D-heavy then
5:      $D \leftarrow \emptyset$
6:      $m_{\text{release}} \leftarrow 0$    ▷ Memory size to release
7:      while $m - m_{\text{release}} > M$ do
8:          Select $r^* \leftarrow \arg\max_{r \in S \setminus D}(r_{\text{current\_output\_len}})$
9:          $D \leftarrow D \cup \{r^*\}$
10:         $m_{\text{release}} \leftarrow m_{\text{release}} + r^*_{\text{memory}}$
11:     end while
12:     return $D$
13: end if
```

thus better absorb the performance degradation. Notably, the predefined memory watermark M is essential to guarantee sufficient memory is reserved to accept at least one new decode request.

This approach deliberately degrades the most degradable TPOT requests to: (1) free resources on D-heavy instances, optimizing TPOT for new decode requests; and (2) limit resource usage on large-chunk P-heavy instances, improving TTFT for prefill requests. The degrading flowing method effectively addresses the technical challenge of request-level TPOT degradation in batch processing. This is because it isolates requests requiring degradation from the original batch (on D-heavy instances) and reassigns them to high-interference batches (on P-heavy instances), thereby preventing interference diffusion to the other requests in the original batch.

③ **TPOT-Aware Decode Backflow:** To prevent excessive interference for decode requests in P-heavy instances, we monitor their real-time TPOT and migrate those approaching the TPOT SLO to D-heavy instances. Specifically, a request is considered to be approaching its SLO when its current TPOT surpasses the product of the SLO value and a predefined *approaching factor* α (e.g., 0.96 in our experiments), enabling proactive optimization before SLO violations occur. Upon flowing back, the decode request is logically treated as a new request, with its output length reset. This reset accounts for the neutralization between the low interference on D-heavy instances and the high interference on P-heavy instances, ensuring the current TPOT closely adheres to the SLO constraint. Each backflow typically triggers a degrading flowing in D-heavy instances to release HBM capacity. Consequently, this shifts the TPOT from the request in the backflow to other requests in the degrading flowing.

However, we intend for optimizing flowing to serve as a safeguard mechanism for TPOT degradation rather than a frequent occurrence. This is because the triggered degrading flowing may prematurely degrade the TPOT of certain requests, reducing their degradation margin. In fact, the frequent backflow indicates improper architectural configuration, where excessive TTFT optimization compromises TPOT, overloading the decode scheduling. For example, the underlying causes may include: (1) oversized chunk size settings in either D-heavy or P-heavy instances, or (2) insufficient quantity of D-heavy instances. Thus, to avoid the frequent backflow, the system architecture should be adjusted to better match workload characteristics.

Algorithm 1 details the request selection mechanism for backflow and degrading flowing. Implemented in the instance scheduler, this algorithm is invoked during the scheduling phase of each inference iteration. It evaluates the current states of instances and requests, together with the TPOT SLO, to select backflow requests for P-heavy instances and degrading requests for D-heavy instances. In Lines 1-3, the scheduler in P-heavy instances selects requests to conduct backflow. Line 2 calculates each request's current TPOT value and adds those nearing the SLO to the optimizing set. Lines 4–12 handle D-heavy instances: if memory usage exceeds the threshold M, the scheduler repeatedly selects the longest decode request currently (Line 8), adds it to the degrading set (Line 9), and updates the released memory (Line 10) until usage drops below M (Line 7). Finally, the algorithm outputs the chosen optimizing or degrading set. The decode requests within the set are subsequently distributed to the D-heavy or P-heavy instances to optimize or degrade TPOT through the proxy in a load-balanced manner.

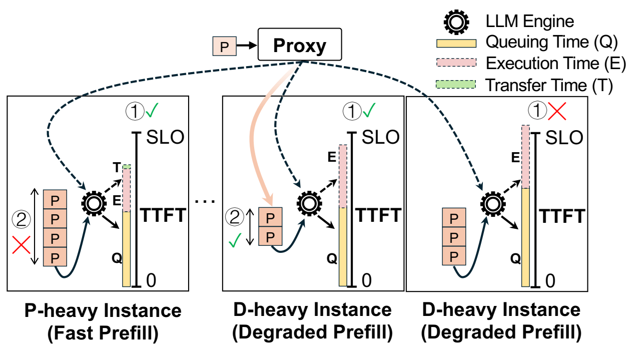

P
Proxy
LLM Engine
Queuing Time (Q)
Execution Time (E)
Transfer Time (T)
SLO
TTFT
P-heavy Instance (Fast Prefill)
D-heavy Instance (Degraded Prefill)
D-heavy Instance (Degraded Prefill)

Figure 13. An illustration of length-aware prefill scheduling. The scheduler preferentially assigns short prefill requests to slower D-heavy instances when their estimated TTFT (queuing + execution + transfer time) satisfies the SLO. This reserves the faster P-heavy instances for long, more time-sensitive requests to ensure they also meet their SLOs.

Algorithm 2 Prefill Scheduling Algorithm in the Proxy

```
Input: Request $r$, Instance set $\mathcal{I}$, TTFT SLO $\tau_{ttft}$
Output: Scheduled instance $i^*$ or $\emptyset$
1:  $\mathcal{I}' \leftarrow \emptyset$
2:  for each $i \in \mathcal{I}$ do
3:      $Q \leftarrow \sum_{r' \in i.\text{queue}} \text{Estimate}(r'.\text{len}, i.\text{chunk}, i.\text{batch})$
4:      $E \leftarrow \text{Estimate}(r.\text{len}, i.\text{chunk}, i.\text{batch})$
5:      $T \leftarrow \mathbb{I}\{i_{type} = \text{P-heavy}\} \cdot \frac{r_{\text{transfer\_size}}}{link\_bw}$
6:      if $Q + E + T < \tau_{ttft}$ then
7:          $\mathcal{I}' \leftarrow \mathcal{I}' \cup \{i\}$
8:      end if
9:  end for
10: if $\mathcal{I}' \neq \emptyset$ then
11:     $i^* \leftarrow \arg\min_{i \in \mathcal{I}'} \sum_{r' \in i.\text{queue}} r'.\text{len}$
12:     return $i^*$
13: else
14:     return $\emptyset$
15: end if
```

3.4 Length-aware Prefill Scheduling

To enable request-level TTFT degradation, we propose a length-aware prefill scheduling strategy. The key idea is to exploit the lower urgency of short prefill requests by routing them to D-heavy instances, intentionally slowing their execution to free up P-heavy instances for more time-sensitive, long prefill requests.

Figure 13 illustrates the scheduling strategy of the prefill algorithm. This algorithm operates within the proxy, assigning each newly arrived prefill request to an instance. The scheduling involves two steps: first, identifying *feasible instances* where assigning the request will not violate its TTFT SLO constraint; second, selecting among feasible instances with the fewest queued prefill tokens, typically favoring a D-heavy instance. The proxy then enqueues the request in the prefill queue of the selected instance, where requests are processed in a first-come, first-served manner.

Specifically, in the first step, the proxy estimates the TTFT for the incoming prefill request on each instance. For D-heavy instances, TTFT is the sum of queuing time (Q) and execution time (E); for P-heavy instances, transfer time (T) is also included due to the need to transfer the KV cache. The queuing time for a request on an instance is defined as the total estimated execution time of the remaining prefill tasks on the instance. Accurately estimating the execution time of a prefill request requires modeling factors such as request length, instance configuration, and batch information. The recent research Vidur [2] models it and provides an accurate and efficient execution time predictor, which we leverage to estimate both queuing time (Q) and execution time (E). According to our experiments, this predictor completes estimation within negligible tens of microseconds. The transfer time (T) is determined by the KV cache size to transfer and link bandwidth, but is typically negligible under high-speed interconnects, as discussed in § 2.2. After estimating TTFTs, instances that can process the prefill request within the TTFT SLO are selected as feasible instances.

In the second step, the scheduler selects the instance with the fewest queuing prefill tokens. This approach is motivated by two reasons. First, if a D-heavy instance is among the feasible instances, it is highly likely to be selected, ensuring that degradable short requests are preferentially degraded. This is because D-heavy instances, having lower prefill processing capacity than P-heavy instances, accommodate fewer queuing tokens under the same TTFT constraint. Second, this strategy helps balance load: if a P-heavy instance has fewer tokens than all feasible D-heavy instances (e.g., after multiple degradable prefill tasks have been assigned to D-heavy instances), the request is assigned to the P-heavy instance, thereby avoiding load imbalance. In this scenario, degradation is unnecessary since a less-loaded P-heavy instance is available for fast prefill.

It is worth noting that if the feasible instance set is empty, the request will inevitably violate the TTFT SLO, often due to a sudden surge in prefill workload. Prior work has proposed the *early rejection* strategy [20] to proactively drop such requests, thus preventing instance overload and subsequent cascading SLO violations. However, to ensure a fair comparison under identical load conditions with PD aggregation—which generally provides sufficient prefill processing capacity—we randomly assign such requests to an instance in our experiments, even though this may occasionally violate the TTFT SLO.

Algorithm 2 presents the detailed implementation of the length-aware prefill scheduling. The algorithm takes three input parameters: a newly arrived request, information about all instances, and the TTFT SLO constraint. It outputs a suitable instance capable of processing the prefill request within

Table 3. Evaluated workloads and SLO constraints.

(TTFT, TPOT)	**SLO1**	**SLO2**
ShareGPT	(3s, 110ms)	(4s, 70ms)
Arxiv Summarization	(4s, 70ms)	(6s, 50ms)

the TTFT constraint, if such an instance exists. Lines 1-9 identify the set of feasible instances capable of processing the request within the TTFT SLO. An instance is considered feasible if the sum of its queuing time (Line 3), the execution time for the new request (Line 4), and the potential transfer time does not exceed the TTFT SLO (Line 5). Specifically, both queuing time and execution time are estimated using an execution time model (e.g., the execution time predictor in Vidur). For only P-heavy instances, the transfer time is determined by dividing the required transfer size by the link bandwidth. Lines 10–12 select the final feasible instance with the fewest queuing prefill tokens to handle the request, considering both TTFT degradation and load balancing. Typically, it is a D-heavy instance, which strategically allows for TTFT degradation. Lines 13-15 indicate that if no instance can execute the request within the TTFT constraint, the algorithm returns an empty result.

3.5 Implementation

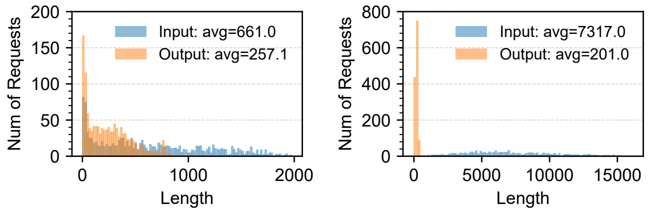
We implement TaiChi on the open-source vLLM [28]. We employ the chunked prefill implementation from vLLM for our P-heavy and D-heavy instances with different chunk sizes. Regarding the KV transfer between the P-heavy instances and the D-heavy instances, we extend vLLM's KV transfer module to enable mutual communication between any two instances via NCCL [16]. To enhance the efficiency of the KV transfer, we decouple the transfer from the critical path of the model execution in vLLM, making it asynchronous. Additionally, we utilize fused CUDA operators to store the received KV cache into vLLM's paged memory, thereby reducing its CPU overhead.

4 Performance Evaluation

4.1 Experiment Setup

Cluster Testbed and Models We deploy our experiments on 8 NVIDIA SXM A100-80GB GPUs connected via NVLINK, using the widely adopted Qwen2.5 series models with FP16 precision. Due to single-node limitations, our experiments focus on multiple instances of Qwen2.5-14B and Qwen2.5-32B. To keep the 32B model's HBM usage below half of total capacity and ensure sufficient KV cache space, we apply tensor parallelism (TP) and set TP=2 for Qwen2.5-32B.

Workloads Setup. To simulate real-world serving scenarios, we assess a chatbot [17] and a summarization application [13], following the methodology of [3, 36]. The chatbot



(a) ShareGPT **(b)** Arxiv Summarization

Figure 14. The input and output length distributions of ShareGPT and Arxiv Summarization datasets.

uses the *ShareGPT* dataset [25], comprising user-shared ChatGPT conversations, while summarization experiments use the *ArXiv Summarization* dataset [1], characterized by long prefill sequences. Since the datasets lack timestamp information, we simulate request arrivals using a Poisson process with varying rates, as in prior work [3, 36]. Figure 14 presents the input and output length distributions. We filter outliers by discarding ShareGPT requests exceeding 2048 tokens and ArXiv Summarization requests over 16,384 tokens.

We evaluate performance under two balanced SLO configurations to highlight our design's benefits across varied user-defined requirements: SLO1 (relatively lower TTFT and higher TPOT) and SLO2 (relatively higher TTFT and lower TPOT), as detailed in Table 3. Summarization tasks typically use longer prefill prompts and demand faster output than chatbot tasks, resulting in higher TTFT but lower TPOT constraints, in line with previous studies [36]. For Qwen2.5-32B, all TPOT SLOs are relaxed by 10 ms to accommodate increased execution time and communication overhead from tensor parallelism.

Metrics. Following prior work [36], we evaluate all methods based on the maximum achievable goodput under the 90% SLO attainment rate (§4.2). Additionally, we demonstrate the reduction in bottleneck latency compared to existing approaches at the maximum achievable goodput of TaiChi, highlighting the direct cause of the improved goodput (§4.3. Moreover, we conduct the performance breakdown to observe changes in both latency and SLO attainment rate, thereby demonstrating the effectiveness of our proposed approach (§4.4). Finally, we present the analysis of the overhead introduced by our design (§4.5).

Baseline. We compare TaiChi to baseline systems, which have both been implemented in the vLLM project [28]:

- **PD aggregation.** Chunked prefill, as a representative of PD aggregation, divides the prefill tasks into small chunks to improve hardware resource utilization and ensure that decode tasks are not stalled for too long. It mitigates but cannot completely eliminate prefill-decode interference caused by long prompts. We prioritize setting its chunk size to meet the required prefill processing capacity for tested workloads, preventing

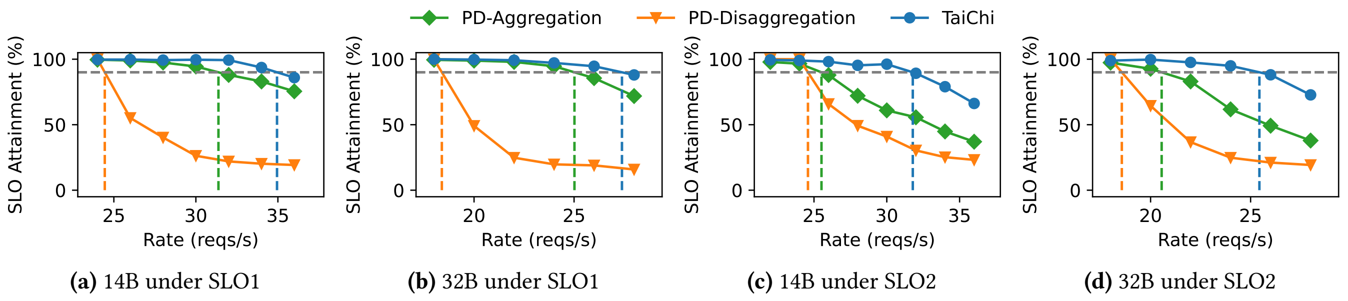


Figure 15. Goodput (vertical lines) for chatbot tasks using Qwen-2.5 models under SLO1 and SLO2.

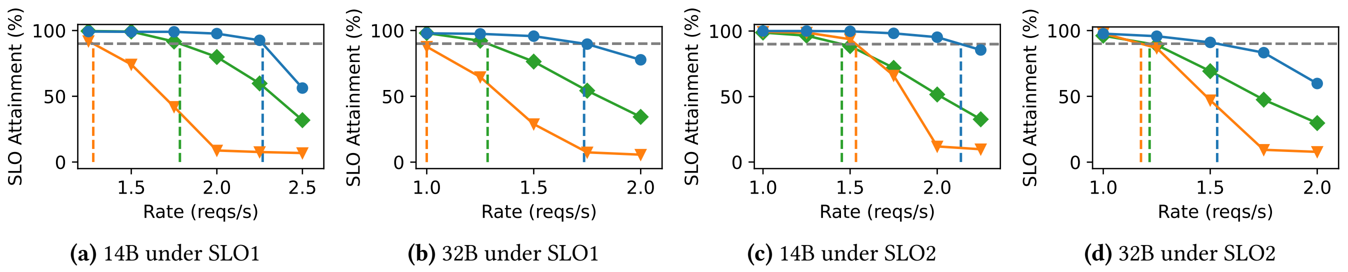


Figure 16. Goodput (vertical lines) for summarization tasks using Qwen-2.5 models under SLO1 and SLO2.

the prefill requests queue from growing excessively and causing TTFT explosion. We also show the latency performance of prioritizing the bounded TPOT with a small-chunk configuration in Section 4.4 for a comparison.

- **PD Disaggregation.** To optimize TPOT, PD disaggregation reserves dedicated instances for decode to eliminate prefill-decode interference issues. Since decode occupies some instances, PD disaggregation faces insufficient prefill processing capacity. We set the PD ratio to the configuration that yields the best TTFT performance, as its TPOT consistently performs well. We extend the PD disaggregation functionality in the vLLM project [28], transforming its original one-to-one KV cache transmission into a many-to-many KV Cache transmission via NCCL [16].

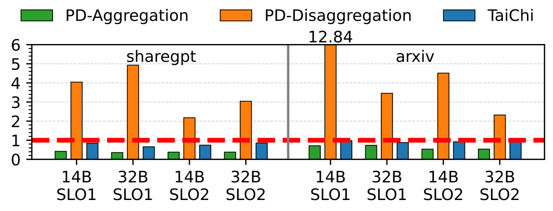
4.2 End-to-end Experiments

In this section, we compare TaiChi with the baseline on real-world application datasets. TaiChi increases maximum goodput by 9–47% over PD aggregation and 29–77% over PD disaggregation across diverse workloads, while maintaining 90% SLO compliance.

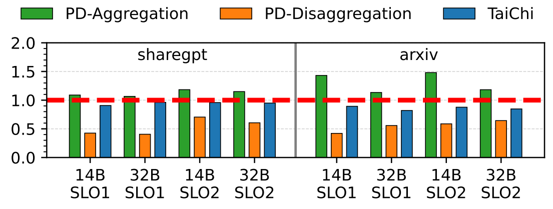
Chatbot. We evaluate the performance of TaiChi on the chatbot application using Qwen2.5 models as Figure 15 shows. Increasing the request rate leads to higher latency violations, reducing SLO attainment. The vertical line indicates the maximum request rate that maintains latency compliance for over 90% of requests.

Compared to PD aggregation, TaiChi achieves 9–11% and 24–25% higher goodput under SLO1 and SLO2, respectively. This improvement is due to TaiChi 's ability to maintain similar prefill processing capacity while effectively bounding TPOT through flowing decode scheduling, which assigns all decode requests first to low-interference D-heavy instances. For SLO1, we use two P-heavy instances (chunk size 1024) and two D-heavy instances (chunk size 512). In contrast, the chunked prefill approach requires all four instances to use a chunk size of 1024 to match prefill processing capacity and TTFT, but this increases interference and causes TPOT violations. For SLO2, we configure two P-heavy instances (chunk size 1024) and two D-heavy instances (chunk size 128), reducing D-heavy chunk size to tighten TPOT and lower prefill processing capacity, thereby taking advantage of the relaxed TTFT SLO. PD aggregation, by contrast, needs four instances with chunk size 512 for similar prefill processing capacity, but this significantly violates the TPOT constraint.

Compared to PD disaggregation, our design increases goodput by 43–49% under SLO1 and 29–37% under SLO2. This improvement is due to our D-heavy instances, which supplement prefill processing capacity that PD disaggregation lacks. In PD disaggregation, two instances each are dedicated to prefill and decode, as decode requires the HBM of two instances. In contrast, our approach allows D-heavy instances to assist with prefill, enhancing overall prefill processing capacity and supporting higher QPS without violating TTFT constraints. Additionally, higher QPS leads to more concurrent decode requests; our solution routes these



(a) TTFT normalized to the SLO



(b) TPOT normalized to the SLO

Figure 17. P90 Latency normalized to the SLOs.

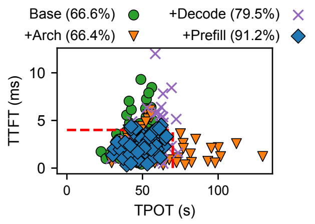


Figure 18. SLO attainment breakdown of proposed techniques: CP256 ⟶ +Arch ⟶ +Flowing Decode ⟶ +Length-Aware Prefill.

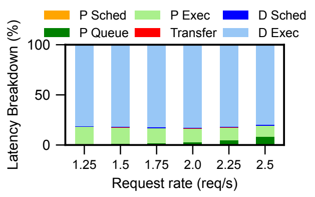


Figure 19. Latency breakdown of requests. Transfer overhead is negligible before the prefill queuing time blows up.

to P-heavy instances without needing extra decode instances, unlike PD disaggregation.

Summarization. We evaluated TaiChi on the summarization task, with results presented in Figure 16. TaiChi improves goodput by 20–47% over PD aggregation and by 30–77% over PD disaggregation.

For SLO1, we used two P-heavy instances with a chunk size of 1024 and two with 256 to meet the 70 ms TPOT SLO. Compared to PD aggregation (chunk size 512), TaiChi achieves 27% and 35% higher goodput for the 14B and 32B models, respectively (Figures 16a, 16b), owing to reduced decode interference. Summarization's long prompts require high prefill processing capacity, but in PD disaggregation, decode instances cannot process prefill, limiting capacity. As a result, TaiChi outperforms PD disaggregation by 77% and 74% for the 14B and 32B models, respectively. Under SLO2, targeting a stricter TPOT, we set the D-heavy chunk size to 128. In this scenario, TaiChi delivers 26–47% and 30–39% higher goodput than PD aggregation and PD disaggregation, respectively (Figures 16c, 16d).

4.3 Latency Reduction

Using latency-shifting scheduling policies, we optimize the latency of requests exceeding SLO constraints by selectively degrading the performance of suitable requests. Figures 17 and 18 show that TaiChi reduces the 90th-percentile (P90) tail latency for TTFT and TPOT under maximum supported load, compared to PD aggregation and PD disaggregation. Figure 17a shows TaiChi achieves a 2.42×–13.20× reduction in TTFT relative to PD disaggregation, due to improved prefill processing capacity and the length-aware prefill scheduling algorithm, which degrades the degradable requests. Figure 17b shows that TaiChi reduces TPOT by 1.11×–1.69× compared to PD aggregation, due to low interference decode and precise flowing decode scheduling at the request level, which also avoids incorrect request degradation.

4.4 Performance Breakdown

To demonstrate the effectiveness of our design, we perform a performance breakdown on the summarization task under SLO1 using Qwen2.5-14B. We start with a 4-instance PD aggregation configured with a chunk size of 256, and add our proposed techniques step by step. Figure 18 shows that we improved the SLO attainment rate from 66.6% (Base) to 91.2% by leveraging the support of the hybrid architecture and applying appropriate degradation decisions through the latency shifting scheduling policies. The latency distribution for CP256 shows that using a small chunk size limits TPOT but causes unacceptable TTFT to exceed the SLO twice, due to limited prefill processing capacity. With our hybrid architecture—setting chunk size to 1024 for the first two (P-heavy) instances—some requests have low TTFT but high TPOT (on P-heavy instances), while others show high TTFT and low TPOT (on D-heavy instances). Although this does not directly optimize latency, it provides a basis for further scheduling strategies. Adding our flowing decode scheduling policy significantly reduces TPOT and increases SLO attainment by 12.9%, as decode is prioritized on low-interference D-heavy instances and only select suitable requests are sent to P-heavy instances for TPOT degradation. Finally, incorporating our length-aware prefill scheduling policy controls excessive TTFT and further raises SLO attainment by 11.7%, by selectively degrading TTFT for suitable requests based on request length and instance queue status.

4.5 Overhead Analysis

To demonstrate the low overhead of our proposed designs, we conducted a latency breakdown analysis for the SLO1 summarization task using Qwen2.5-14B. The primary sources of overhead are the KV cache transfer and the scheduling algorithm execution. As shown in Figure 19, for each request, the transfer, prefill scheduling, and decode scheduling overheads are minimal, accounting for only 0.20%, 0.01%, and 0.89% of the total request time, respectively. The low transfer overhead benefits from modern high-performance interconnects, while the low scheduling overhead is due to the lightweight nature of our algorithms.

5 Related works

PD Aggregation. Orca [32] introduces continuous batching to improve throughput. FastServe [31] employs iteration-level preemptive scheduling to reduce queuing delays for long-running tasks. NanoFlow [37] decomposes batches into nano-batches to overlap computation, memory, and network usage, thereby enhancing GPU resource utilization. In contrast, TaiChi focuses on optimizing goodput, which is orthogonal to these techniques. Sarathi-Serve [3] mitigates decode stalls in PD aggregation by introducing chunked prefill, which divides the prefill task into multiple chunks and piggybacks decode tasks with chunked prefill computation. SOLA [9] establishes an optimization model for PD aggregation, scheduling tasks based on the real-time status of requests and instances to balance TTFT and TPOT. Unlike these works that focus exclusively on PD aggregation, TaiChi unifies PD aggregation, PD disaggregation, and a novel hybrid mode within a single architecture to achieve goodput-optimal performance under diverse SLO constraints.

PD Disaggregation. DistServe [36] and SplitWise [19] propose executing the prefill and decode phases on separate hardware resources to eliminate interference and enable phase-specific optimization. Adrenaline [14] optimizes the resource utilization of PD disaggregation via offloading the attention computation of decode to prefill instances. Unlike PD disaggregation, TaiChi adopts a hybrid strategy that unifies PD aggregation and disaggregation to improve goodput by effectively balancing the trade-off between TTFT and TPOT. Moreover, DynaServe [24], which was developed concurrently with our system, introduces techniques to optimize the goodput of LLM service systems via balancing TTFT and TPOT. It proposes splitting a request into two virtual sub-requests (e.g., the first containing prefill and a small part of early decode, and the second containing the remaining decode tasks), to balance token throughput and the *time between token* (TBT) SLO constraint. However, DynaServe assumes that the output length is known in advance, which is often unrealistic in real-world scenarios, as discussed in Challenge 2 (§ 2.5). In contrast, our proposed TaiChi dynamically controls per-request TPOT without requiring prior knowledge of output length, through an adaptive combination of multi-stage degradation flowing and optimization-triggered backflow.

6 Conclusion

We present a comprehensive study of PD aggregation and disaggregation in LLM serving, highlighting an inherent dilemma between these approaches for maximizing goodput under SLO constraints. To address it, we propose TaiChi, a unified serving system that reallocates GPU resources to shift latency across different phases and requests, thereby increasing the SLO attainment rate. TaiChi features a hybrid-mode inference, flowing decode scheduling, and length-aware prefill scheduling, enabling effective latency shifting for LLM serving. Experimental results demonstrate that TaiChi improves goodput by up to 77% compared to SOTA systems.

References

[1] Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. 2024. InferCept: Efficient Intercept Support for Augmented Large Language Model Inference. arXiv:2402.01869 [cs.LG] https://arxiv.org/abs/2402.01869

[2] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. VIDUR: A LARGE-SCALE SIMULATION FRAMEWORK FOR LLM INFERENCE. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 351–366. https://proceedings.mlsys.org/paper_files/paper/2024/file/b74a8de47d2b3c928360e0a011f48351-Paper-Conference.pdf

[3] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 7, 18 pages.

[4] Inflection AI. 2023. *Inflection-1 Technical Report.* Technical Report. Inflection AI. https://inflection.ai/assets/Inflection-1.pdf Accessed: 2024-05-10.

[5] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebr'on, and Sumit K. Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *ArXiv* abs/2305.13245 (2023). https://api.semanticscholar.org/CorpusID:258833177

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[7] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao

Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[8] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC '24)*. Santa Clara, CA, 111–126.

[9] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Guohao Dai, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Eighth Conference on Machine Learning and Systems*.

[10] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: increasing GPU utilization during generative inference for higher throughput. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 791, 13 pages.

[11] Hannah Calzi Kleidermacher and James Zou. 2025. Science Across Languages: Assessing LLM Multilingual Translation of Scientific Papers. arXiv:2502.17882 [cs.AI] https://arxiv.org/abs/2502.17882

[12] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[13] LangChain. 2023. LangChain Use Case: Summarization. https://www.langchain.com/use-cases/summarization. Accessed: 2023.

[14] 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. *arXiv preprint arXiv:2503.20552* (2025). https://arxiv.org/abs/2503.20552

[15] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G. Parameswaran, and Eugene Wu. 2024. Towards Accurate and Efficient Document Analytics with Large Language Models. arXiv:2405.04674 [cs.DB] https://arxiv.org/abs/2405.04674

[16] NVIDIA Corporation. 2023. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl. Accessed: 2025-04-17.

[17] OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed: 2025-04-15.

[18] OpenAI. 2023. GPT-4. https://openai.com/index/gpt-4/. Accessed: 2025-05-08.

[19] 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)*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[20] 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. arXiv:2407.00079 [cs.DC] https://arxiv.org/abs/2407.00079

[21] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Efficient Interactive LLM Serving with Proxy Model-based Sequence Length Prediction. arXiv:2404.08509 [cs.DC] https://arxiv.org/abs/2404.08509

[22] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL] https://arxiv.org/abs/2308.12950

[24] 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. arXiv:2504.09285 [cs.DC] https://arxiv.org/abs/2504.09285

[25] ShareGPT. 2023. ShareGPT Teams. https://sharegpt.com/. Accessed: [Insert Access Date].

[26] Foteini Strati, Sara McAllister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. DéjàVu: KV-cache streaming for fast, fault-tolerant generative LLM serving. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1902, 27 pages.

[27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[28] vllm project. 2023. vLLM: Easy, Fast, and Cheap LLM Serving for Everyone. https://github.com/vllm-project/vllm. Accessed: 2025-04-14.

[29] Yuxin Wang, Yuhan Chen, Zeyu Li, Zhenheng Tang, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2024. Towards Efficient and Reliable LLM Serving: A Real-World Workload Study. *ArXiv* abs/2401.17644 (2024). https://api.semanticscholar.org/CorpusID:271710854

[30] Zhibin Wang, Shipeng Li, Yuhang Zhou, Xue Li, Rong Gu, Nguyen Cam-Tu, Chen Tian, and Sheng Zhong. 2024. Revisiting SLO and Goodput Metrics in LLM Serving. arXiv:2410.14257 [cs.LG] https://arxiv.org/abs/2410.14257

[31] 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. arXiv:2305.05920 [cs.LG] https://arxiv.org/abs/2305.05920

[32] 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)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[33] Wei Zhang, Zhiyu Wu, Yi Mu, Banruo Liu, Myungjin Lee, and Fan Lai. 2025. Tempo: Application-aware LLM Serving with Mixed SLO Requirements. *arXiv preprint arXiv:2504.20068* (2025).

[34] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. WildChat: 1M ChatGPT Interaction Logs in the Wild. arXiv:2405.01470 [cs.CL] https://arxiv.org/abs/2405.01470

[35] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. 2023. Response Length Perception and Sequence Scheduling: An LLM-Empowered LLM Inference Pipeline. *ArXiv* abs/2305.13144 (2023). https://api.semanticscholar.org/CorpusID:258833168

[36] 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 *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

[37] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2025. NanoFlow: Towards Optimal Large Language Model Serving Throughput. arXiv:2408.12757 [cs.DC] https://arxiv.org/abs/2408.12757