Batched in Back: Characterizing and Optimizing Offline LLM Inference in Production with ACDC

Leping Yang[1,2] Xue Li[*2] Kun Qian[2] Erci Xu[*1] Mingzhen Han[2] Haoran Zhu[2] Tao He[2]
Zuolong Yin[2] Ennan Zhai[2] Wenyuan Yu[2] Jingren Zhou[2] Guangtao Xue[1]
[1]Shanghai Jiao Tong University [2]Alibaba Group

Abstract

Serving offline large language model (LLM) inference workloads (*e.g.,* log summarization and bulk translation) can consume up to 30% of GPUs in production. Despite this significant share, the characteristics of offline inference remain largely understudied. In this paper, we start by analyzing 1.5 million tasks comprising 23 billion requests across text and multi-modal models. We discover that the key properties of offline workloads, namely inherent determinism and throughput orientation, are neither exploited by online LLM serving systems nor by existing offline serving frameworks.

We design ACDC, a task-aware offline serving system that leverages these properties along three orthogonal axes, namely *resource orchestration*, *cache management*, and *execution flow*. ACDC has been deployed in production for over three months, achieving up to 3.9× higher output cost-efficiency. Beyond isolated deployment, we further identify that online prefill stages leave up to 38.3% of capacity idle as resource bubbles. We design a hybrid-serving mechanism that harvests these bubbles for offline tasks, improving end-to-end throughput by up to 35.7% over the state-of-the-art alternative.

ACM Reference Format:
Leping Yang, Xue Li, Kun Qian, Erci Xu, Mingzhen Han, Haoran Zhu, Tao He, Zuolong Yin, Ennan Zhai, Wenyuan Yu, Jingren Zhou, and Guangtao Xue. 2026. Batched in Back: Characterizing and Optimizing Offline LLM Inference in Production with ACDC. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3830418.3843877

1 Introduction

Large Language Models (LLMs) first gained widespread adoption in chatbots and code copilots [2, 21, 26]. These online, human-in-the-loop applications impose strict latency requirements [30] and incur high serving costs. As the application scope expands, workloads that do not require ultra-low interactive latency, such as log summarization, literary creation, and bulk translation, can be served more economically via offline task submission [4, 6, 9, 22]. As a worldwide Model-as-a-Service (MaaS) provider, we receive dozens of thousands of new offline tasks per day, comprising billions of requests, and allocate up to 30% of our GPU capacity to offline serving.

The adoption of offline serving is driven by aligned incentives between users and providers. From the user's perspective, offline tasks are not required to provide immediate responses for individual requests. Instead, users prioritize completing the entire task at a relatively low cost (*e.g.,* 50% discount [4, 5, 22]). From the provider's perspective, accommodating the high variance of online traffic requires provisioning sufficient GPU capacity to meet daily peak demand. During off-peak hours, this provisioned capacity would otherwise remain idle. Therefore, running offline tasks is the most cost-effective way to amortize fixed GPU costs.

Although extensive prior work has optimized online LLM serving [7, 8, 11–13, 16–18, 24, 25, 28–30, 32, 34–36, 40–42, 46, 47], our in-production experience shows that applying these online techniques to offline serving still leaves substantial performance headroom. Meanwhile, existing work on offline-task optimization largely focuses on customized batch construction to improve key-value (KV) cache reuse [43, 45]. While important, these approaches do not capture the full range of bottlenecks we observe in practice.

To the best of our knowledge, we are the first to present a systematic study of LLM offline serving. Based on how offline tasks execute in production, we identify two key differences from online workloads: (1) **Inherent determinism**. An offline task can contain thousands of requests that are fully known at submission (i.e., the entire input set is determined before execution). As a result, optimizations requiring future knowledge that is unattainable in online serving become feasible for offline workloads. (2) **Throughput orientation**. Offline tasks prioritize overall task completion time over time-to-first-token (TTFT) or time-per-output-token (TPOT), making them throughput-oriented. For example, across over 170,000 offline tasks, the average completion time is 149 minutes, with the maximum observed completion time exceeding 1,000 minutes. Since throughput and per-token latency are fundamentally at odds in serving systems, directly applying

*Corresponding authors: Xue Li and Erci Xu.

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843877

online-oriented techniques to offline tasks yields suboptimal performance. Our motivation experiment, which simply removes TTFT/TPOT constraints, achieves a 45.1% throughput gain (see §3.1 for details), confirming that purpose-built optimizations for offline characteristics, rather than mere tuning knobs, are both necessary and promising.

To this end, we conduct a broad measurement study of offline workloads in our production service, spanning over 1.5 million tasks and 23 billion requests, invoking both text-only and multimodal models. We first provide an overview of how offline tasks differ across model families (§3.2). Guided by the insight of inherent determinism, we then analyze request-level properties within a task, including input/output length distributions (§3.3) and prefix sharing (§3.4). We observe significant similarity in input/output length distributions among requests within the same task. Furthermore, our prefix sharing analysis shows that there is almost no prefix overlap across different tasks, whereas intra-task prefix overlap is high, with a median hit rate exceeding 80%. In addition, we point out the inadequacies of legacy offline systems (*e.g.,* MapReduce [10] and Spark [39]) for LLM workloads, where classical optimizations like pipeline parallelism and fixed cost models often backfire (§3.5).

Leveraging these insights, we propose customized optimizations along three orthogonal axes. (1) **Resource orchestration:** Workload patterns vary widely across tasks, and our controlled experiments show that applying task-oriented parallel strategies yields higher efficiency. With task-specific parallelism tuning (§4.1), ACDC improves inference throughput by 22.5%. (2) **Cache management:** KV cache must be proactively managed at task granularity to achieve optimal performance. We reorder and serve all requests within each task and enforce strict KV cache lifetime management to maximize reuse (§4.2), which improves overall KV cache hit rate by up to 35.5%. (3) **Execution flow:** While effective in latency-sensitive scenarios, blindly employing speculative decoding in throughput-oriented offline settings can be counterproductive. We derive an effective adaptive policy for speculative decoding under offline workloads (§4.3), improving decoding throughput by 13.1%. Comprehensive comparative experiments demonstrate that ACDC outperforms the state-of-the-art alternative by at least 14.2% in throughput. The entire system has been deployed at scale on our MaaS platform, operating stably for over three months, and has served hundreds of thousands of offline tasks. In production, ACDC delivers up to 3.9× higher output cost-efficiency (measured by output token generation speed per GPU), compared to our previous deployment (§4.4).

So far, we have focused on isolated deployment, *i.e.,* utilizing idle capacity during off-peak hours. However, even at peak traffic, substantial capacity remains untapped. Under prefill-decode disaggregation, prefill instances typically process one request at a time for responsiveness, creating resource bubbles as large as 38.3% in our observations. These

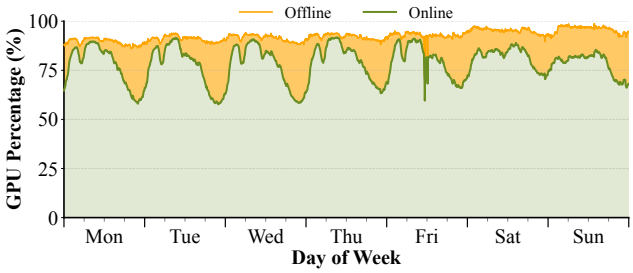


Figure 1. Evolution of online and offline serving instances.

bubbles, unusable by latency-sensitive online requests, can be harvested by throughput-oriented offline tasks. By accurately characterizing and modeling the online prefill stage, we estimate the expected bubble size each time it occurs (§5.1). This enables us to quantitatively convert latency-driven resource bubbles into harvestable capacity for offline tasks. Based on this insight, we design a hybrid-serving mechanism (§5.2) that increases end-to-end offline throughput by up to 90.0% and 35.7% compared to the baseline and the best existing method, respectively (§5.3).

The main contributions of this paper are as follows:

- ACDC presents the first production-scale quantitative study of LLM offline tasks, spanning 1.5 million tasks and 23 billion requests across text-only and multimodal models (§3). Our analysis transforms previously anecdotal intuitions into quantifiable design boundaries. We have open-sourced representative traces to facilitate further research in this domain.*
- Building on these findings, ACDC systematically tailors optimizations for offline serving, including the co-adaptation of parallelism, prefix caching, and speculative decoding. The key contribution lies not only in the customization of each individual mechanism, but also in their non-trivial orchestration, which achieves up to a 3.9× improvement in output cost efficiency in production (§4).
- ACDC further introduces a hybrid serving mechanism that co-schedules offline tasks with online requests by exploiting online prefill bubbles at fine granularity within PD-disaggregated architectures. This mechanism boosts offline task throughput without affecting online SLOs (§5).

2 Inference: Not Just Serving the Online

Online inference, such as chatbots and code copilots, is the archetypical use of LLMs. With requests (i.e., API calls) submitted one at a time, performance is measured by per-token latency metrics, namely TTFT and TPOT. Accordingly, the Service Level Objectives are defined in terms of latency (*e.g.,* 10 seconds for TTFT and 50 ms for TPOT).

However, not all requests come from real-time interaction. A significant proportion of inference is actually performed offline. In stark contrast to online serving, offline requests are not submitted individually but grouped into a task containing thousands or even millions of requests. As a result, users are

*https://github.com/alibaba/ServeGen/tree/main/data/offline

no longer concerned with per-token latency like TTFT or TPOT but instead the completion time of the entire task (i.e., overall throughput). The relaxed deadline in offline inference leaves much more headroom for scheduling, allowing us to achieve higher cost efficiency and attract customers with time-insensitive workloads.

In practice, our platform (Alibaba Model Studio) provides both online and offline inference across a wide range of models, including over 200 foundation models and thousands of their fine-tuned variants. Hundreds of enterprises now rely on our services to run their applications. With nearly 100 K GPUs operating around the globe, our daily API usage has reached the tens-of-billions level.

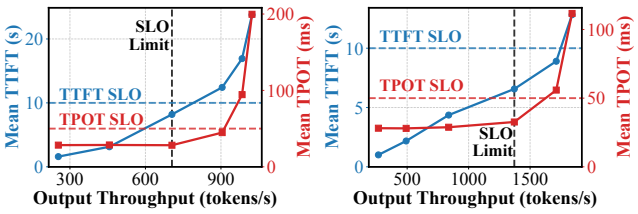
Statistically, Figure 1 shows the proportion of inference resources consumed by offline jobs over one week. Because online requests have stricter latency requirements, we prioritize resource allocation to online serving. During off-peak periods, up to 30% of GPUs (i.e., tens of thousands) are reallocated to offline inference tasks. Note that the nighttime peaks visible in Figure 1 are a deliberate consequence of the production scheduler backfilling offline work into off-peak slots for online serving. Most offline tasks are throughput-oriented, with completion targets ranging from hours to days, making them tolerant of variable execution rates. This property is precisely what enables hybrid serving in §5.

3 Understanding Offline Inference

3.1 Status Quo of Offline Requests Scheduling

With such a large scale of deployment, the efficiency of offline serving becomes critical. We initially adopted the same scheduling strategy for both online and offline tasks for fast deployment. Unsurprisingly, we soon realized this shoehorning is suboptimal. First, unlike the unpredictable online serving, the workload patterns or even the content are already known and fixed at submission time. Moreover, the two types of services have different SLOs (latency for online vs. throughput for offline). If we stick to the aggressive online scheduling strategy designed to meet latency SLOs and ignore the clairvoyance advantage of offline requests, we inevitably miss significant optimization opportunities or even hurt performance.

Here, we further demonstrate a motivating experiment. As illustrated in Figure 2, we profile the Qwen3-32B model under various parallelization strategies (TP = 2 and TP = 4) on NVIDIA H20 GPUs using vLLM, measuring per-instance throughput alongside the corresponding TTFT and TPOT. The results show that online SLOs are exceptionally stringent, forcing the system to sacrifice substantial throughput to guarantee service quality. However, this trade-off is unnecessary for offline tasks. For instance, as illustrated in Figure 2a, directly using the online serving system for offline workloads forfeits 45.1% of potential throughput without justification. This gap measures the difference between the SLO-constrained throughput limit (the vertical dashed line),



(a) TP=2. **(b)** TP=4.

Figure 2. Inherent trade-off between throughput and latency for Qwen3-32B (1,000 input tokens, 500 output tokens), tested on NVIDIA H20 GPUs using vLLM.

Model	Type	Size/Activated	#GPUs	Tasks (K)	Requests (M)
A	Text/MoE	~30B/~3B	1	156	1,032
B	Text/MoE	~30B/~3B	1	134	3,656
C	Text/MoE	~200B/~20B	4	579	9,136
D	Text/MoE	~1T/~80B	8	72	432
L	VL/MoE	~20B/~3B	1	119	3,351
M	VL/MoE	~100B/~15B	4	260	4,244
N	VL/Dense	~30B/~30B	2	268	1,337

Table 1. Number of tasks and requests across seven anonymized models over a three-month period, along with model size, activated parameters for MoE models, and per-instance GPU requirements.

where both TTFT and TPOT satisfy the production online SLO thresholds, and the unconstrained maximum throughput (the rightmost data point), where no latency constraint is imposed. This indicates that throughput-focused optimizations can unlock significant latent performance for offline tasks. In §4, we comprehensively evaluate key optimizations on offline workloads. Using production traces, we identify which mechanisms remain effective, which become ineffective, and which require offline-specific fine-tuning beyond direct adoption from online serving.

3.2 Traces Overview

To understand the practical demands of LLM offline inference workloads, we conduct an extensive analysis of offline task submissions across diverse models in production, including four text-only models (Model-A to Model-D) and three multimodal (vision and language) models (Model-L to Model-N). As shown in Table 1, these models contribute a massive volume of offline tasks on our platform—over 1.5 million task submissions comprising 23 billion requests, collected over three months.

Despite packing thousands to millions of requests into each task, the traces reveal fundamental differences across models and even within a single model.

- **Scale and granularity vary drastically across models.** Model-C dominates with 9,136M requests across 579K tasks, over 21× the request volume of Model-D (432M). Meanwhile, the number of requests per task also differs widely. For example, Model-A has slightly more tasks than Model-B (156K vs. 134K) yet less than one-third of its requests (1,032M vs. 3,656M); similar cases exist among VL

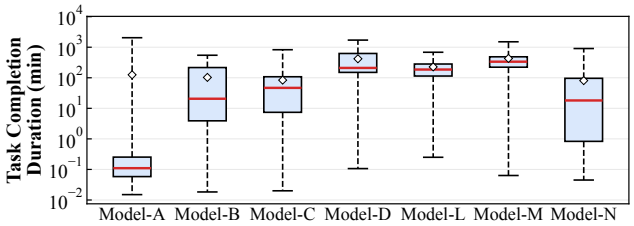


Figure 3. Offline task completion durations across models.

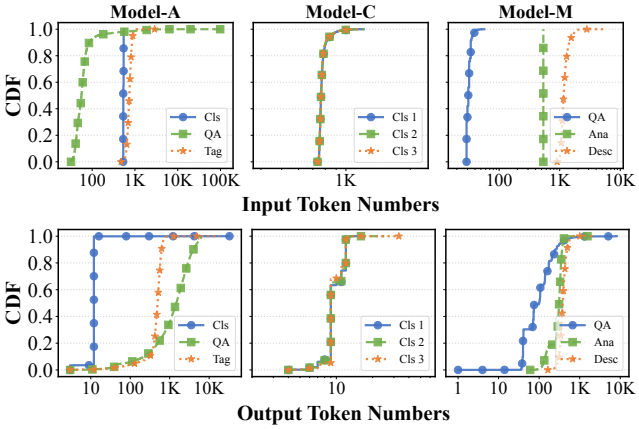


Figure 4. CDFs of input and output token lengths for sampled offline tasks across three models, with five high-level task categories including classification (Cls), question answering (QA), tagging (Tag), content analysis (Ana), and description (Desc).

models (*e.g.*, Model-L with 119K tasks and 3,351M requests vs. Model-N with 268K tasks but only 1,337M requests). Such disparity in both absolute scale and per-task granularity rules out any one-size-fits-all resource strategy.

- **Task completion times span orders of magnitude.** We analyze the distribution of completion times for offline tasks sampled from different models. In Figure 3, the diamond represents the mean, the red horizontal line represents the median, and the box represents the interquartile range (IQR) of 25%–75%. Model-A's median task duration is nearly three orders of magnitude shorter than Model-D's. Even within a single model, the spread is extreme: Model-N ranges from sub-minute tasks to ones exceeding 1,000 minutes.

3.3 Input/Output Length Distributions

Now, we dive deeper into token-level statistics. We find that request length distributions and durations are highly variable *across* tasks yet remarkably similar *within* each task, hinting at the potential for task-level scheduling.

Token distributions are shaped by both the task and the model. We sample three representative offline tasks run by each of the seven models and analyze the distribution of input and output tokens in their requests. We select three models as examples in Figure 4. The cumulative distribution functions (CDFs) in the figure highlight that different tasks within the same model exhibit substantially different distribution shapes and introduce non-trivial variation in token lengths (e.g., Model-A and Model-M). At the same time, models differ in task diversity. For example, Model-C handles a relatively limited range of tasks, primarily text classification, whereas Model-M serves a wider variety of tasks due to its multimodal nature, including image content question answering, analysis, and description. Crucially, requests within the same task cluster tightly in token length, enabling task-aware batching that improves prefix cache reuse and resource utilization.

Request durations vary across tasks and modalities. We further calculate the distribution of completion times for offline requests across the three offline tasks sampled from each model. As shown in Figure 5, requests running on Model-A and Model-C exhibit relatively stable duration profiles, whereas Model-B and Model-D show pronounced inter-task differences. Multimodal models (Model-L, M, N) display wider intra-task duration spreads than text-only models, reflecting the added uncertainty from complex input modalities. The large interquartile ranges and long outlier tails suggest that latency is driven not only by model size but also by dynamic factors such as prompt structure and context length.

3.4 Prefix Sharing Within Tasks

Finally, we examine prefix sharing within offline tasks to assess the potential for KV cache reuse (Figure 6). We calculate the average prefix sharing rate across all requests in each offline task and plot the distribution of rates for tasks on each model. Additionally, for multimodal models, we separately calculate the proportion of images reused within each task and image duplication frequency (Figure 7). We discover that sharing opportunities are abundant for text but scarce for images, suggesting that cache strategies must be modality-aware.

Text prefixes offer extensive cache reuse. Among text-only models, Model-B achieves a median prefix hit rate exceeding 80% (IQR $\approx$ 50–95%), driven by recurring instruction templates. Multimodal models show even higher and more concentrated text-prefix sharing, as their textual prompts tend to be highly standardized. This makes task-level KV cache reuse particularly effective for the text modality.

Image duplication is low and task-dependent. Intra-task image repetition rates remain generally below 20% even in the best case (Model-N). As Figure 7 shows, Model-M images are rarely repeated more than twice, whereas Model-N frequently reuses the same image across many requests—reflecting multi-round questioning over a single image. The contrast indicates that image cache strategies cannot simply mirror those designed for text prefixes and must account for task-specific reuse patterns.

3.5 Why Not Porting Existing Approaches?

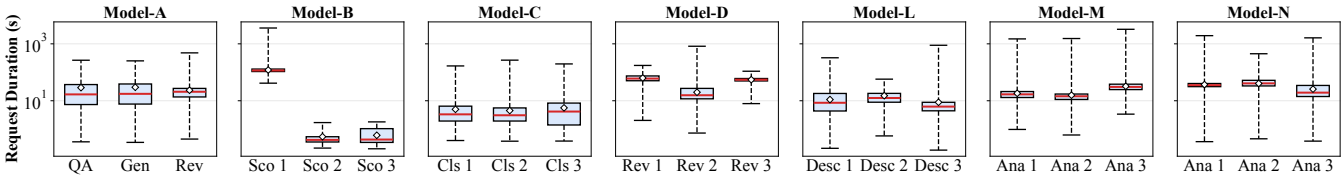


Figure 5. Request duration distributions across sampled offline tasks for each model, grouped into the same high-level task categories as Figure 4, with additional categories of content generation (Gen), review (Rev), and scoring (Sco).

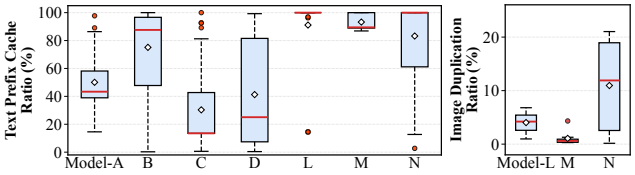


Figure 6. Prefix sharing rates in text-only models (left) and image duplication ratios in multimodal models (right).

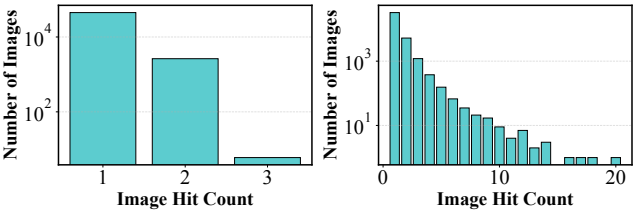
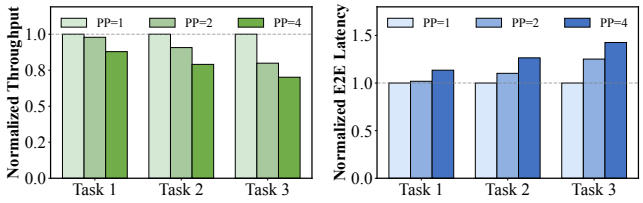


Figure 7. Image hit count distributions for Model-M (left) and Model-N (right).



(a) Output token throughput. **(b)** Mean end-to-end latency.

Figure 8. The impact of PP on decoding throughput and average end-to-end latency of requests across different tasks.

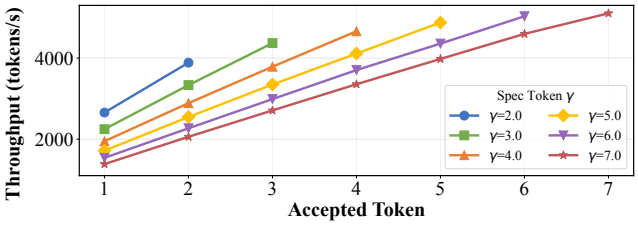


Figure 9. Effect of speculative decoding across different speculative tokens (γ) with various accepted tokens.

LLM-specific serving systems are primarily designed for online workloads. Modern LLM serving frameworks, such as vLLM [15], SGLang [44], and TensorRT-LLM [20], provide high-performance building blocks. However, existing scheduling policies are universally designed to meet online SLOs, leaving throughput on the table for offline workloads, where latency constraints are relaxed and full request visibility is available at submission time.

Hybrid-serving solutions target only co-located architectures. Recent online-offline hybrid-serving proposals, such as HyGen [31], Echo [33], and ConServe [27], focus on optimizing co-located prefill and decode on the same GPU, relying primarily on preemption-based scheduling. As a result, they cannot exploit the idle prefill bubbles that arise in production PD-disaggregated deployments, and may even underperform a simple design that serves online and offline workloads separately (§5).

Clearly, the unique characteristics of offline LLM inference, such as workload heterogeneity and modality-dependent prefix sharing, make it ill-suited to existing LLM serving systems. Intuitively, we then sought to port established practices from large-scale data-processing frameworks [10, 14, 19, 39] to schedule offline LLM inference tasks/requests. However, this set of attempts turned out to be fruitless as well.

DAG-based pipelining degrades LLM inference. DAG-based partitioning from batch processing frameworks (*e.g.*, MapReduce [10], Spark [39]) is counter-productive when applied to LLM inference: Figure 8 shows that increasing pipeline parallelism (PP) steadily degrades decoding throughput due to pipeline bubbles [38].

Fixed cost models miss runtime variability. Another limitation is that the execution engines of existing batch processing frameworks follow a deterministic flow under a fixed cost model once a workload's profile (*e.g.*, input/output distribution) is established. This is because the computational cost of each stage (*e.g.*, map, reduce) is fully determined by the input size and the operation applied, with no content-dependent variation, and thus they can safely ignore fine-grained runtime variability.

Conversely, LLM inference is inherently non-deterministic: output length is unknown until generation completes, and techniques like speculative decoding further amplify this variability. Speculative decoding relies on low-cost drafting followed by parallel verification, whose acceptance rate depends on the specific semantic content of each request. We run an offline task on the Qwen3-32B model deployed on four NVIDIA H20 GPUs, with different numbers of accepted tokens. As illustrated in Figure 9, variations in performance when increasing the number of accepted tokens can lead to a difference of up to 3.69× in throughput. Without accounting for such content-dependent dynamics, achieving near-optimal performance in serving offline LLM workloads becomes impractical.

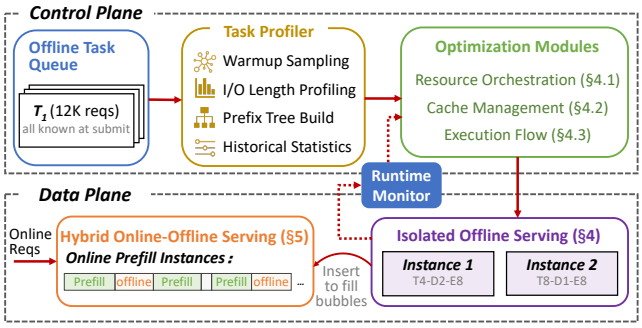


Figure 10. Overview of the ACDC system architecture.

Bridging the gap. The analyses above demonstrate that neither online serving systems nor classical offline task systems can fully exploit the unique characteristics of offline LLM inference (§3.3 and §3.4). This motivates us to design a purpose-built system (§4) for this important workload. In particular, the wide variation in input/output length distributions (§3.3) motivates task-aware resource orchestration (§4.1); the high degree of intra-task prefix sharing (§3.4) motivates prefix-tree-based cache management (§4.2); and runtime variability (§3.5) motivates adaptive execution-flow control (§4.3).

4 ACDC

Driven by our analysis of offline workloads, we build ACDC, as illustrated in Figure 10, to optimize performance along three axes, namely resource orchestration, cache management, and execution flow. At a high level, we leverage offline profiling to build a historical performance database for offline tasks. Inter-task scheduling follows FCFS, while each task is optimized independently. For each new task, ACDC performs initial warm-up executions to derive task-aware strategies, which are then dynamically refined at runtime to maintain high efficiency.

4.1 Resource Orchestration

Multi-GPU inference performance fundamentally hinges on the parallelism strategy. We conduct a series of experiments to evaluate the performance across various Tensor Parallelism (TP), Data Parallelism (DP), and Expert Parallelism (EP) configurations. We select Qwen3-30B-A3B, a Mixture-of-Experts (MoE) model, to evaluate eight offline tasks spanning a range of input-to-output ratios. Tasks 1–4 represent prefill-dominated workloads with a fixed 1-token output and increasing input tokens (100, 1,000, 10,000, and 100,000). Tasks 5–8 represent decoding-intensive workloads with a fixed 1,000-token input and increasing output tokens (100, 1,000, 10,000, and 100,000). Results are shown in Figure 11.

Observation. A salient pattern is that the optimal TP-DP-EP combination shifts as the primary bottleneck transitions. Our experiments in Figure 11a show that the optimal parallelism strategy for the prefill phase is dictated by input prompt length. For short inputs (Task 1), maximizing DP while keeping TP minimal (T1-D8-E8) minimizes communication overhead and delivers the highest throughput. As prompt length increases, the workload becomes increasingly compute-intensive and less sensitive to communication costs. Consequently, the optimal parallelization strategy shifts toward configurations with larger tensor parallelism. Accordingly, the best configuration evolves from T1-D8-E8 for Task 1 to T8-D1-E8 for Task 4.

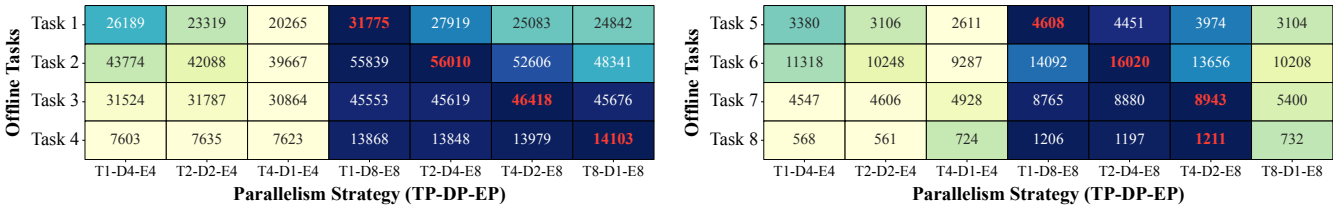
Figure 11b shows that longer outputs increasingly favor higher TP degrees. The reason is that decoding is memory-bandwidth-bound. Offline throughput depends on how many concurrent requests the KV cache can sustain. Increasing TP shards model weights across more GPUs, thereby freeing memory for a larger KV cache. As output length grows and the per-request KV footprint increases, this capacity gain increasingly outweighs the additional communication overhead. In contrast, for short outputs, KV-cache capacity is abundant, so TP=1 is preferable to minimize communication. However, the benefit of higher TP disappears once TP exceeds the number of GQA heads in the model (e.g., four heads in Qwen3-30B-A3B). Because the KV cache is sharded across its four KV heads, setting TP to eight replicates each head across two GPUs, yielding no additional capacity benefit while incurring extra communication overhead. As a result, T8-D1-E8 trails T4-D2-E8 by more than 33%, despite using the maximum TP degree.

Our optimization. Motivated by the observations above, we propose a three-step task-oriented parallel strategy that dynamically selects the optimal parallelism configuration for each offline task.

Foundation. We maintain a historical database that records the optimal parallelism strategies for various GPU types and models across different input-output sequence lengths. For each incoming task, we uniformly sample 5% of its requests to profile the output length distribution. We then select an initial parallelism configuration by matching the profiled input-output length ratio against the database.

Adaptation. During the remaining 95% of execution, we continuously monitor the input-output length distribution of finished requests. The monitoring window is set at intervals of 5% of total task requests. A strategy switch is triggered when the observed distribution diverges sufficiently from the profile used to select the current configuration.

Rotation. Implementing runtime switching of parallelism strategies is non-trivial, as mainstream engines (*i.e.,* vLLM [15] and SGLang [44]) require service interruption and restart, incurring minutes of extra overhead that may prolong task completion. To address this, we reserve a rotating host within each homogeneous GPU cluster to enable live strategy switching. To switch, we launch a new instance with the target strategy on the rotating host to handle incoming requests. The existing instance drains its remaining workload and then becomes the new rotating host. The required KV cache is pre-built on the destination host



(a) Input throughput (tokens/s per instance). **(b)** Output throughput (tokens/s per instance).

Figure 11. Instance-level token throughput of offline tasks under different TP-DP-EP combinations.

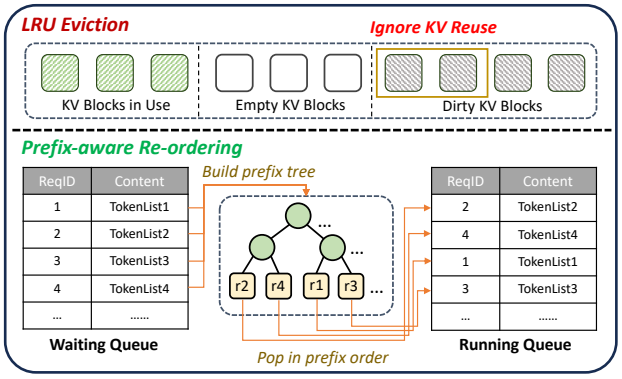
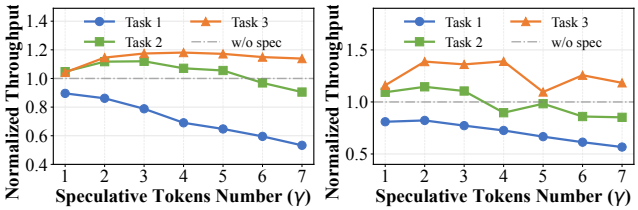


Figure 12. Reorder offline requests in tasks by prefix.



(a) Practical acceptance rate. **(b)** Fixed acceptance rate.

Figure 13. Impact of speculative decoding across tasks with practical and fixed acceptance rates.

before handover using a Mooncake-like [28] distributed storage system, enabling an instantaneous and non-disruptive transition for the live offline task. Such reconfigurations are triggered only when the optimal configuration changes. In practice, rotations during a task are rare and usually zero, since most tasks converge after the initial decision. We manage offline tasks across the cluster to ensure that the overhead of reserving a rotating host remains negligible.

4.2 Cache Management

Current inference engines, such as vLLM, primarily rely on prefix caching managed by a Least Recently Used (LRU) eviction policy. As shown in the upper half of Figure 12, GPU memory often contains empty KV blocks and stale blocks awaiting release. When requests with identical prefixes are scattered across the queue, this mechanism may evict reusable KV blocks before subsequent requests can hit them, resulting in redundant prefill computations and wasted GPU cycles. Some existing systems, including BlendServe [43] and BatchLLM [45], propose cache-aware scheduling. ACDC extends this line of work to multi-modal workloads and maximizes prefix reuse at the task level.

Our optimization. As illustrated in the lower half of Figure 12, we design a prefix-aware KV cache reusing system that transforms cache management from passive LRU eviction to active request reordering.

Prefix tree construction. For each task, we construct a prefix tree from the known token sequences of all requests before execution. Leaf nodes under the same parent node represent requests that share common prefixes, while the tree depth corresponds to the length of the shared prefix. This preprocessing is performed once at task submission, incurring no runtime overhead. The resulting prefix tree is maintained only for the lifetime of the task and discarded upon completion, without being persisted across tasks.

Depth-first scheduling. During execution, we traverse the prefix tree in depth-first order to schedule requests consecutively, thereby maximizing KV cache reuse. By processing sibling leaf nodes (requests sharing the same parent prefix) back-to-back, the KV blocks loaded for one request remain hot in GPU memory for subsequent requests, eliminating redundant prefill computations. This scheduling strategy ensures that the working set of KV blocks remains minimal and that evicted blocks are no longer needed by future requests.

Prefix-aware eviction. We augment the standard LRU eviction policy with prefix-tree awareness. The policy prioritizes evicting KV blocks corresponding to leaf nodes with no remaining descendant requests. When GPU memory is constrained, the system evicts KV blocks from subtrees whose requests have all been processed, preventing premature eviction of prefixes required by upcoming requests. This approach transforms eviction from a reactive, history-based decision into a proactive, clairvoyant optimization.

Duplicated images reuse. Based on the analysis in §3.4, certain images may be accessed multiple times in an offline task, making caching worthwhile. Since all requests are known in advance, we sort them by text prefix and identify, for each duplicated image, the first and last request that references it. We cache the image only between these two endpoints and evict it immediately after the last access. For example, consider the Model-N task with the highest image-duplication rate in Figure 6, which contains 50,000 requests with over 21% duplicate images. Caching the corresponding encoded representations avoids re-encoding 1,200–3,000 vision tokens per image during prefill, depending on the resolution.

4.3 Execution Flow

The execution flow presents its own optimization challenges. In our experience, the most significant factor is speculative decoding, a widely adopted technique for reducing TPOT in online serving. To substantiate this point, we perform experiments on Qwen3-32B with EAGLE-3 speculator under varying numbers of speculative tokens (γ). Figure 13a shows the throughput of three tasks with different input lengths (100, 1,000, and 10,000 tokens) and fixed output length (1,000 tokens), normalized to the baseline without speculative decoding. Figure 13b shares the same settings, except that the per-position acceptance rate is fixed at 0.5.

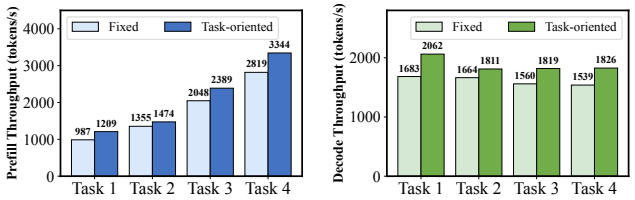
Observation. The critical factor turns out to be input length. For Task 1 (short input), speculative decoding consistently underperforms the baseline, with throughput declining monotonically as γ increases, even when the acceptance rate is fixed. This occurs because, under large batch sizes with short inputs, the decoding phase becomes compute-bound. The overhead of invoking the draft model and verifying multiple tokens outweighs the benefits of parallel generation. In contrast, Tasks 2 and 3 (longer inputs) exhibit performance improvements both in Figure 13a and Figure 13b. During decoding, the system becomes predominantly memory-bandwidth bound, as the target model must load its full weights and the cumulative KV cache to generate each token. Speculative decoding alleviates this bottleneck by amortizing the high cost of a single target model invocation across multiple speculative tokens.

Combining the results from Figure 13a and Figure 13b, we further observe that for mid-to-long context tasks (Tasks 2 and 3), speculative decoding exhibits a non-monotonic throughput curve. There is a clear sweet spot for γ. For Task 2, throughput peaks at $\gamma = 2$ or 3 and drops below baseline beyond $\gamma = 5$. For Task 3 (longest input), performance remains above baseline for all γ values, peaking at $\gamma = 4$ with approximately 18% improvement in Figure 13a. Generally, as input length increases, the optimal γ increases, reflecting the greater benefits of speculative decoding.

Our optimization. Motivated by the observations above, we propose an adaptive speculative decoding approach that continuously tunes γ based on runtime conditions.

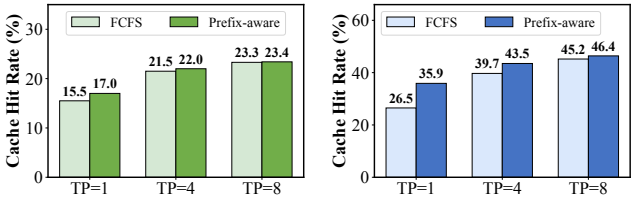
Warmup. Following the same warmup-then-adapt paradigm as §4.1, we maintain a historical database of optimal speculative token counts (γ) for various GPU types, models, batch sizes, and input length ranges. For each incoming task, we uniformly sample 5% of its requests to profile the acceptance rates under different γ and batch sizes. We then select an initial γ value based on the sampled batch size distribution and the historical data.

Runtime adaptation. During execution, we monitor the batch size and acceptance rate across decoding iterations. The speculation policy is re-evaluated at fixed intervals (*e.g.,* every 5% of total requests) to account for shifts in workload characteristics. When the acceptance rate drops below a threshold or the measured throughput indicates suboptimal performance, the system adjusts γ gradually within a bounded range. This prevents oscillation while keeping speculative decoding beneficial under the dynamic batch conditions of offline inference.



(a) Prefill throughput. **(b)** Decode throughput.

Figure 14. Task-oriented parallel strategy effects on tasks with various input-output length ratios.



(a) Cache hit rates of online. **(b)** Cache hit rates of offline.

Figure 15. Prefix-aware KV cache reusing effects on online and offline tasks.

4.4 Evaluation Results

Micro-benchmarks. We evaluate the three techniques proposed in §4. Specifically, we test task-oriented parallel strategy selection on Qwen3-30B-A3B (MoE). Prefix-aware KV cache reusing is executed on Qwen3-32B, and adaptive speculative decoding is tested with EAGLE-3 speculator. Both models are deployed on eight NVIDIA H20 GPUs.

- **Task-oriented parallel strategy selection.** To compare the parallel strategy with a fixed-configuration baseline, we select four tasks whose input-output length ratios shift across phases (0.59, 0.81, 1.31, and 1.83). The task-oriented method adjusts parallelism during execution, whereas the baseline fixes the configuration based on the initial 5% warmup sample. The results in Figure 14 show that task-oriented strategy selection achieves up to 22.5% increase in throughput during the prefill and decode stages compared to the fixed configuration.
- **Prefix-aware KV cache reusing.** We evaluate the prefix-aware KV cache against the First-Come-First-Serve (FCFS) baseline across varying TPs. As shown in Figure 15, the KV cache hit rates for both online and offline tasks increase after adopting the prefix-aware strategy, particularly when TP is small (*e.g.,* 35.5% improvement at TP=1). This is because, at smaller TP values, the limited GPU memory capacity causes more KV blocks to be evicted at runtime, making prefix-aware sorting more impactful. At the same time, we observe that the KV cache hit rate in offline tasks

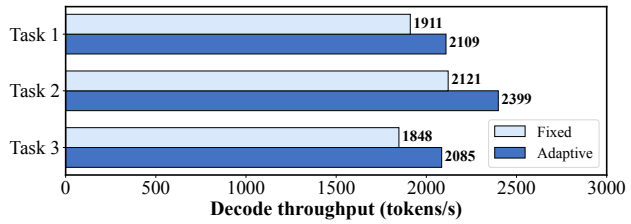


Figure 16. Adaptive speculative decoding tested on tasks with different batch sizes.

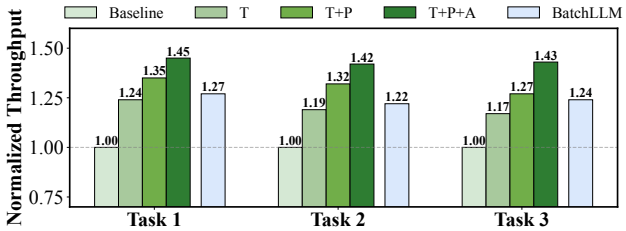


Figure 17. Comparison of normalized throughput between the offline optimization methods and the baseline.

is much higher than that for online tasks (up to 2.11×). This reflects the greater prompt similarity within offline tasks compared to the diverse prompts of online requests (§3.4). Therefore, prefix-aware sorting for offline requests is more beneficial.

- **Adaptive speculative decoding.** We compare fixed-γ strategies against adaptively adjusting γ based on batch size across three tasks with short, medium, and long inputs respectively, where concurrency varies across different stages of each task. As shown in Figure 16, the adaptive strategy increases decoding throughput by up to 13.1%, compared to the fixed-γ strategy.

Comparisons with the state-of-the-art alternative. We evaluate the cumulative throughput impact of adding Task-oriented parallel strategy selection (T), Prefix-aware KV cache reusing (P), and Adaptive speculative decoding (A) on Qwen3-32B across three offline tasks, using the online serving mode as the baseline, with prefix caching and chunked prefill enabled and no SLO constraints. We also compare against BatchLLM [45], which employs prefix-sharing group-based scheduling. Beyond such prefix-aware scheduling, ACDC further improves performance through task-aware parallelism selection and adaptive speculative decoding. As shown in Figure 17, after combining three aspects (T+P+A), our method outperforms BatchLLM by at least 14.2%. Notably, Task 3 benefits most from adaptive speculative decoding, as its requests predominantly produce long outputs, amplifying the decoding-phase gains.

Overhead. Operationally, all ACDC components run on CPUs and impose no impact on GPU execution. Runtime monitoring relies exclusively on existing system logs, introducing no additional computation. Prefix-tree construction is performed only once per task at submission time. For example, building a prefix tree for 30,000 requests on a single CPU core takes approximately 400 ms, which is negligible compared with hour-long task durations. Profiling samples only 5% of offline requests, making it an inherent part of the task workflow rather than additional overhead.

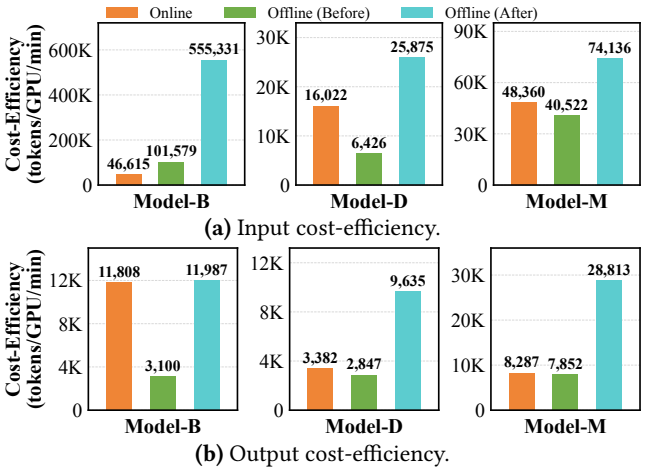


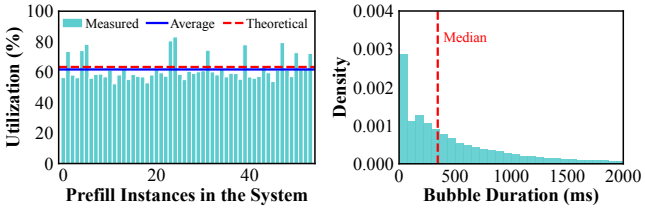
Figure 18. Cost-efficiency improvements of ACDC in production (for serving Model-B, Model-D, and Model-M).

Generality and robustness. To validate hardware portability, we benchmark ACDC on NVIDIA H100 GPUs using the same model, tasks and configuration as in Figure 17. Across the three tasks, ACDC achieves a 37%–41% end-to-end throughput improvement over the baseline, consistent with the gain observed on H20.

ACDC also exhibits strong transferability: (1) *cross-model generality*. ACDC's mechanisms are largely model-agnostic, as they rely primarily on task-specific characteristics rather than model internals. (2) *cold-start capability*. The profiling database is maintained independently for each model and GPU type. When no historical profiling data is available for a new model or task type, ACDC defaults to a conservative configuration during the initial 5% warm-up window. The resulting throughput penalty is bounded and amortized over the full task duration, yielding negligible end-to-end impact. (3) *robustness*. Our sensitivity analysis shows that noise injected into the profiled length distribution causes only minor utilization loss. Moreover, the 5%-interval re-evaluation window enables continuous adaptation, allowing ACDC to track gradual distribution drift without manual intervention.

4.5 ACDC in Production

We are currently rolling out the deployment of ACDC across Alibaba Model Studio. At present, ACDC handles offline batch inference for a subset of models (10 models, spanning more than 2,000 instances in total), with plans to extend its coverage to all models incrementally. Figure 18 summarizes the cost-efficiency gains achieved by the techniques in §4, based on the production deployments of Model-B, Model-D, and Model-M, whose characteristics are listed in Table 1.



(a) Utilization of prefill instances. **(b)** Bubble duration distribution.

Figure 19. Analysis of utilization and bubble duration.

These models are among the most widely used on our platform, serving tasks such as user persona generation, creative writing, generation of Standard Operating Procedures (SOPs), and information summarization from unstructured documents. To evaluate ACDC, we focus on cost-efficiency, defined as the number of input or output tokens processed per GPU per minute, as this metric directly correlates with the provider's operational serving costs. In all scenarios, the serving engine is vLLM [15]. ACDC is elastically deployed and consumes an average of 570, 1,007, and 363 GPU-hours per day for Model-B, Model-D, and Model-M, respectively.

Figure 18 presents the cost-efficiency results for both online and offline serving over a two-week period, comparing performance before and after the deployment of ACDC. Prior to ACDC, offline tasks were served using our online serving system, which employs a series of online-optimized techniques [17, 28, 32, 46] with online SLO constraints ("Offline (Before)" in Figure 18). As shown, without tailored optimizations, offline tasks can be even less cost-efficient than online serving. After deploying ACDC, the input cost efficiency of offline serving improved by 1.8–5.5×, while the output cost efficiency improved by 3.4–3.9×.

5 Fusing ACDC with Online Inference

The optimizations presented in §4 target an *isolated* setup where offline tasks run on dedicated GPU capacity that would otherwise sit idle during off-peak periods. Although this setup has already delivered substantial throughput gains in production, we ask whether ACDC can be pushed further to harvest resources *even during peak online traffic*. At first glance, this appears infeasible because GPUs are fully occupied by latency-sensitive online requests during busy periods. However, the popular prefill-decode disaggregation [24, 28, 46] paradigm reveals an underlying opportunity.

5.1 Observation and Analysis

We next analyze the resource inefficiency inherent in the PD-disaggregated architecture and quantify the opportunity it creates for offline serving. Prefill-decode disaggregation effectively eliminates cross-stage interference and guarantees TPOT SLO attainment, but it can lead to low resource utilization in the prefill stage. This is because online request arrivals are bursty and unpredictable. In our production data, prefill instances exhibit frequent and sizeable resource *bubbles* (up to 38.3% of prefill capacity).

We formalize this inefficiency using queueing theory. Since the prefill phase is compute-intensive, batching multiple requests yields diminishing throughput gains [1]. In practice, the prefill batch size is therefore typically set to 1 to return the first token promptly [35]. For analytical tractability, we model the prefill stage as an *M/D/1* queue (Markovian arrivals, deterministic service time, single server), corresponding to one prefill instance in the system. Let λ denote the request arrival rate and $s = \mathbb{E}[S]$ the service time per request. The average waiting time $\mathbb{E}[W]$ in the queue is then given by:

$$\mathbb{E}[W] = \frac{\lambda s^2}{2(1-\lambda s)} = \frac{\rho s}{2(1-\rho)}, \tag{1}$$

where $\rho = \lambda s < 1$ is the server utilization. The total expected TTFT latency $\mathbb{E}[T]$ is the sum of waiting and service times:

$$\mathbb{E}[T] = \mathbb{E}[W] + s = \frac{\rho s}{2(1-\rho)} + s. \tag{2}$$

This reveals a major drawback since TTFT grows rapidly as server utilization approaches 1. In other words, to comply with TTFT SLOs, the prefill stage must operate at low utilization, leading to significant resource waste. Specifically, server utilization ρ can be expressed in terms of the expected TTFT $\mathbb{E}[T]$ and service time s as:

$$\rho = \frac{2\big(\mathbb{E}[T] - s\big)}{2\mathbb{E}[T] - s}. \tag{3}$$

Analysis with production data. Figure 19a illustrates a real-world serving scenario for a 310B-parameter model with 54 concurrent prefill instances. Here, the average prefill time is $s = \mathbb{E}[S] = 756$ ms, and the observed average TTFT is $\mathbb{E}[T] = 1.41$ s, yielding a theoretical server utilization of only 63.4%. The measured utilization in practice is even lower at 61.7%, meaning that 38.3% of the prefill capacity is wasted as "bubble time".

Figure 19b further analyzes the distribution of bubble durations across all prefill instances in the same scenario. Over a one-day period, more than 3.3 million bubbles occur, exhibiting a long-tailed distribution with a median duration of 345 ms. Many of these idle intervals last hundreds of milliseconds, representing valuable computational opportunities that can be repurposed to process offline requests without impacting online SLO attainment.

5.2 Hybrid Scheduling Strategy

We explore a hybrid scheduling strategy that simultaneously serves online and offline requests under the PD-disaggregated architecture. The core idea is to opportunistically schedule offline prefill requests (in a chunk-by-chunk manner) during idle "bubble" periods in the online prefill stage, while routing their decoding to separate instances, as illustrated in Figure 20.

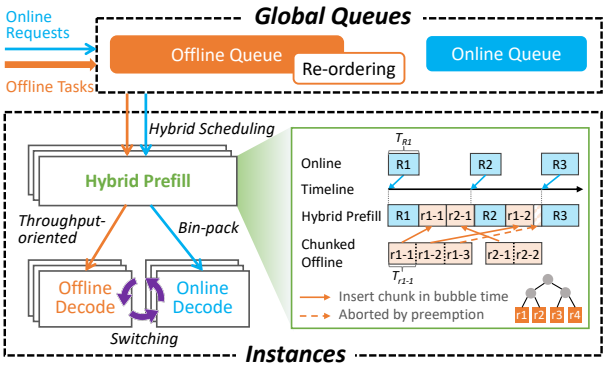


Figure 20. Hybrid scheduling strategy.

Heuristics for prefill stage. The goal of hybrid scheduling in the prefill stage is to fill bubbles with offline requests without violating online TTFT SLOs. As shown in Figure 20, we maintain a queue of offline prefill requests that can be scheduled opportunistically. This queue can be reordered as a prefix tree to maximize the prefix caching benefits discussed in §4.2. Each offline prefill request is split into multiple chunks of length c and processed using chunked prefill [1]. Shorter chunks are more likely to fit into transient idle bubbles and incur fewer preemptions.

When a bubble is detected, we attempt to schedule a chunked offline prefill task that maximizes benefit based on the following heuristics: (1) The chunk must be the first unscheduled chunk of its request to guarantee correctness. (2) Scheduling offline requests according to the reordered prefix-tree queue can maximize prefix cache reuse. (3) Larger chunks, measured by their theoretical computation time without prefix caching, yield higher benefit. (4) The chunk is expected to complete within the bubble duration to avoid preemption overhead. (5) If the chunk is not the last one in its request, scheduling it increases GPU memory usage for the KV cache. We therefore require sufficient remaining memory to accommodate the KV cache for subsequent chunks of ongoing offline requests and any incoming online request, preventing preemptions due to memory exhaustion.

Formulation. We formalize the above heuristics into a benefit-maximization formulation that guides chunk selection at each bubble. To operationalize this, we first define the expected computation time of the i-th chunk ($0 \le i < \lceil l/c \rceil$) of an offline request of length l, given that the first p tokens in this chunk are already cached, as $t_c(i,p)$. The theoretical computation time (*i.e.,* with no prefix cached) is then $t_c(i,0)$. We formulate the benefit of scheduling this chunk as:

$$\text{Benefit} = t_c(i,0) \cdot (1-P), \tag{4}$$

where P is the probability of preemption due to either exceeding the bubble duration or insufficient GPU memory:

$$P = 1 - \min\Big(\Pr\big[B \ge d + t_c(i,p)\big],\ \Pr\big[L \le m - \max(k_r)\big]\Big). \tag{5}$$

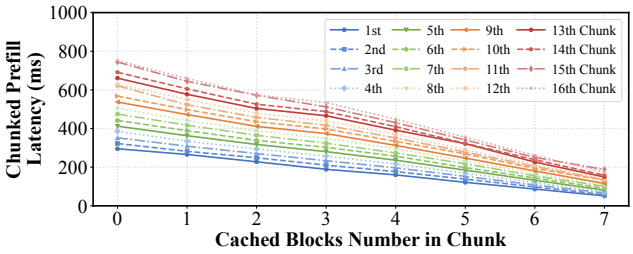


Figure 21. Profiling of chunked prefill latency $t_c(i,p)$.

Here, B is the random variable representing bubble duration, d is the time already elapsed in the current bubble, L is the random variable denoting the input length of the next incoming online request, m is the maximum number of additional tokens that can be accommodated in the KV cache after processing this chunk, k_r is the number of tokens required by the remaining chunks of an offline request r that has been scheduled but not yet completed, and $max(k_r)$ is the maximum value of k_r across all such pending requests.

The first condition ensures that the bubble duration is sufficient to accommodate the chunk's execution. Regarding memory constraints, we account for L to accommodate the next incoming online request, since online requests have higher priority and will preempt offline requests when memory is insufficient. We also incorporate $\max(k_r)$ because the remaining chunks of any partially completed offline request must eventually be processed, so their KV cache memory must be reserved in advance. Including $\max(k_r)$ in the memory budget acts as a penalty to discourage overly aggressive concurrent scheduling of multiple offline requests. For both online and offline requests, KV cache memory is logically freed immediately upon prefill completion by overlapping prefill computation with prefill-to-decode communication. Physical memory management is handled via the LRU eviction policy in vLLM.

Implementation. In practice, the distributions of B (bubble duration) and L (online request length) are estimated from historical traces, *e.g.,* Figure 19b shows the empirical distribution of B. The computation time $t_c(i,p)$ is obtained through offline profiling. Figure 21 illustrates the profiling results of $t_c(i,p)$ for Qwen3-32B, with the experimental setup described in §5.3. For scheduling, we traverse the prefix tree (*i.e.,* the reordered offline request queue). Upon bubble detection, we identify the first group of unprocessed requests that share a common parent node in the tree, compare the first unscheduled chunk of each request in this group, and select the one with the highest benefit.

If an online request arrives while an offline chunk is being processed, we attempt to complete the current chunk only if the online request's TTFT SLO can still be satisfied, as determined using offline profiling data. Otherwise, we preempt the offline chunk and immediately prioritize the online request, while preserving the already computed KV cache to avoid redundant work. Such preemption is naturally

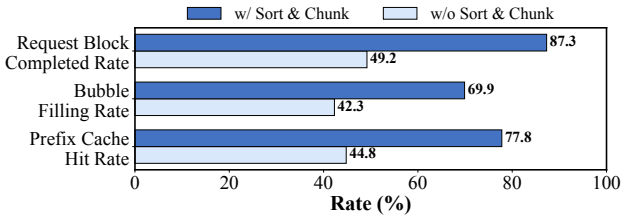
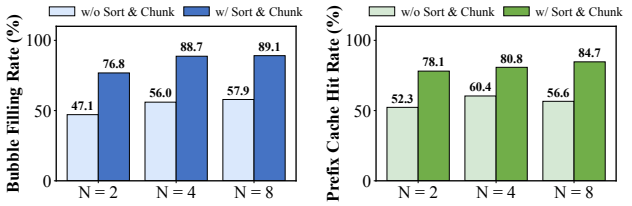


Figure 22. Impact of prefix sorting and chunked prefill.



(a) Bubble filling rate. **(b)** KV cache hit rate.

Figure 23. Hybrid scheduling effect on multiple instances.

supported by our implementation. In our PD-disaggregated architecture, the prefill stage executes incrementally and transfers KV cache to decode instances asynchronously on a layer-by-layer basis to overlap computation and communication, rather than as a monolithic forward pass. To preserve this flexibility, our prefill implementation in production does not use CUDA graphs, which require replaying a fixed end-to-end execution sequence. This layer-by-layer execution naturally enables preemption, as the system can simply stop an offline chunk at the next layer boundary and switch to the online request with negligible control overhead.

Decoding Stage. We continue to serve online and offline requests separately on decoding instances. This separation is motivated by their divergent objectives. Online decoding requests must meet stringent latency requirements to satisfy TPOT SLOs, whereas offline decoding requests prioritize throughput and can tolerate higher latency. These differing goals lead to distinct optimal engine configurations (*e.g.,* `max_batch_size`) and scheduling policies.

For online decoding, we employ continuous batching [15] combined with a bin-packing scheduling strategy to maximize GPU utilization while meeting TPOT SLOs. For offline decoding, we adopt the task-oriented parallel strategy and adaptive speculative decoding discussed in §4 to maximize overall throughput. To adapt to fluctuating online workloads, we enable dynamic reconfiguration of decoding instances, allowing them to switch between serving online and offline requests as needed.

5.3 Evaluation Results

Setup. We evaluate the hybrid scheduling strategy using the Qwen3-32B model on a single instance deployed on four NVIDIA H20 GPUs with tensor parallelism (TP) degree 4, and also scale to 2, 4, and 8 nodes, powered by the vLLM framework [15]. The chunk size c is set to 2048 tokens, the KV cache block size to 256 tokens, and the maximum input length to 32,768 tokens. Both online and offline request traces are collected from production logs of our serving platform. Online traces are sampled from typical online services for the same model, containing over 60,000 requests with various bubble time distributions. Offline traces consist of 30,000 requests collected from 75 distinct task types.

Results. To demonstrate the effect of prefix sorting and chunked prefill for offline requests, we compare three key metrics: (1) the prefix block cache hit rate for offline requests, (2) the bubble filling rate (*i.e.,* the fraction of total bubbles that is utilized), and (3) the block completion rate for offline requests (*i.e.,* the proportion of offline request blocks successfully completed during bubble periods). The baseline approach schedules offline requests without prefix sorting or chunking. Instead, it aggressively fills available bubbles with full offline requests, which often leads to poor cache reuse and frequent preemptions.

- As shown in Figure 22, on a single instance, constructing a prefix tree for sorting improves the block cache hit rate by 73.6%. This higher hit rate reduces chunk processing time, enabling more offline chunks to be scheduled into bubble periods and increasing the bubble filling rate by 65.2%. Consequently, the number of offline request blocks completed within these bubbles rises by 77.4%, significantly boosting offline request throughput and increasing resource utilization without impacting online SLOs.
- We further test with multiple instances working together to process tasks. By grouping similar requests based on shared prefixes, we maximize KV cache reuse on each instance with chunked prefill support. As shown in Figure 23, compared to the baseline that sequentially assigns requests to instances without chunking, the bubble time fill rate and the KV cache hit rate improve by up to 63.1% and 49.6%, respectively.

We further evaluate the end-to-end throughput when combining hybrid scheduling with the offline optimizations from §4. To this end, we run four online instances to process online tasks and four offline instances to process offline tasks simultaneously. A portion of the sorted offline requests are inserted into bubble time on online instances via hybrid scheduling. We compare three methods: (1) Baseline, which uses the same settings for online bubble insertion and offline processing as §4.4 and §5.3. (2) BatchLLM+C, which runs BatchLLM on offline instances combined with inserting chunked offline requests into online bubbles. (3) HyGen [31], a recent online-offline serving system that co-locates prefill and decode. Each instance runs the Qwen3-32B model on eight NVIDIA H20 GPUs. We evaluate three different offline task-sets, each containing four different offline tasks.

As shown in Figure 24, compared to Baseline, BatchLLM+C, and HyGen, ACDC improves end-to-end throughput by up to 90.0%, 35.7%, and 5.6×, respectively. Task-set 1 shows the greatest improvement because it consists mainly of long-input/short-output requests, where hybrid

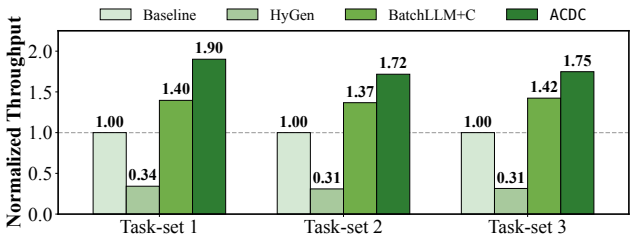


Figure 24. Comparison of normalized throughput between ACDC and other methods.

scheduling during prefill yields greater benefits. HyGen performs significantly worse because it inherently suffers from prefill–decode interference and relies on priority-based preemption to protect online SLOs. In our experiments, each offline request is preempted 259 times on average, resulting in substantial overhead. In contrast, ACDC leverages the PD-disaggregated architecture, in which prefill and decode run on physically separate instances, allowing offline tasks to fill prefill bubbles with nearly no preemption overhead.

Online SLO impact. A critical requirement for hybrid scheduling is that harvesting bubbles must not degrade the quality of online serving. Since ACDC inserts offline work exclusively during the prefill phase, online TPOT remains completely unaffected. For TTFT, we compare the tail latency of online requests before and after hybrid serving throughout the entire measurement period. The increases are only 0.48% at P95, 0.67% at P99, and 0.69% at P999, all well within the production SLO budget, which confirms that bubble harvesting is an effective mechanism with negligible latency overhead.

Sensitivity. ACDC's hybrid scheduler estimates bubble durations using a 24-hour sliding window of historical observations. To characterize the temporal stability of this signal, we measure the standard deviation of the daily mean bubble duration across 150 online prefill instances over one week and observe only 2.8%–6.1% variation, indicating that the distribution evolves slowly relative to the re-estimation interval. We further quantify robustness to prediction error by injecting uniform noise of ±5%, ±10%, and ±20% into the duration estimates. The corresponding bubble-time utilization degrades by only 2.8%, 7.9%, and 10.3%, respectively, confirming graceful degradation under moderate misprediction.

6 Experiences and Lessons Learned

Why "adaptive" is harder than it sounds. In optimizing offline LLM serving, task-oriented parallel strategy selection (§4.1) and adaptive speculative decoding (§4.3) sound straightforward but hide a subtle trap.

In our first implementation, both mechanisms triggered adaptation upon observing throughput degradation. This caused a vicious cycle: a parallel strategy change would temporarily perturb throughput, which triggered speculative decoding adjustment, which further destabilized throughput, which re-triggered parallel configuration tuning. The system oscillated continuously between configurations without converging. The root cause was *shared triggering signals*. Both adaptations monitored the same metric (throughput), so any perturbation cascaded across mechanisms. The solution required *independent indicators* for each adaptation.

After extensive evaluations and critical internal review, we discovered that the optimal parallel strategy depends solely on *input/output length distribution*, which can be observed independently of runtime performance. Correspondingly, the best indicator for speculative decoding is the *batch size*. If the distribution remains stable but the batch size drops, only speculative decoding should adapt. Conversely, if the distribution shifts, the parallel configuration should be re-evaluated based on our historical profiling database (§4.1), not on observed performance. This decoupling eliminated oscillation and led to the stable design presented in §4.

The hidden cost of hybrid serving: memory capacity as a first-class constraint. Our key operational lesson is that *memory capacity*, not compute scheduling, governs hybrid serving efficiency. While PagedAttention [15] eliminates traditional fragmentation concerns, a subtler problem emerges: offline requests accumulate partial KV state across chunked prefill, competing for the same memory pool that online requests need on-demand.

Our initial design optimized solely for bubble utilization. This led the scheduler to favor scheduling *new* offline request chunks. First chunks, lacking accumulated KV cache from prior iterations, execute faster than subsequent chunks of in-progress requests. The unintended consequence: the KV cache became saturated with half-finished offline requests, each holding partial state that could not be released until the entire request completed. When online requests arrived, these incomplete offline requests had to be preempted and swapped out, wasting all prior computation. Paradoxically, aggressive bubble-filling led to *worse* bubble utilization due to massive preemption overhead.

Once we identified this pattern, we introduced the $\max(k_r)$ term in our scheduling formulation (§5.2), which penalizes starting new offline requests when existing ones remain incomplete. This encourages the scheduler to finish in-progress requests before admitting new ones, reducing both memory pressure and preemption waste. The lesson: in hybrid serving, *completing existing work often matters more than starting new work.*

In hindsight, clairvoyance is the assumption doing the heavy lifting. Much of ACDC's gains rest on *clairvoyance*, *i.e.,* full request visibility at task submission time. Knowing all requests up front lets us profile once and commit: both profiling-based parallelism selection (§4.1) and global prefix-aware sorting (§4.2) critically depend on it.

However, emerging multi-turn agent workloads partially break this assumption: agentic pipelines emit requests incrementally, each depending on prior outputs [3, 23, 37].

ACDC's mechanisms then split along their reliance on visibility: those needing only a stable per-task length distribution (*e.g.,* parallelism selection) still apply to observed sub-sequences, whereas prefix-aware global ordering, which assumes the whole request set at once, needs rethinking. Bridging this gap calls for ahead-of-time preprocessing that surfaces partial visibility before execution, combined with runtime adaptation that consumes newly revealed requests as they arrive.

7 Related Work

LLM serving systems. There is a large body of existing work on optimizing LLM serving, focusing on improving throughput or reducing latency. vLLM [15] and SGLang [44] are two popular frameworks for LLM inference that integrate extensive high-performance techniques. Most system-level efforts are motivated by online serving, such as prefill-decode disaggregation [24, 46], parallelism strategies [18, 34, 47], request scheduling [7, 13, 30], resource allocation and autoscaling [11, 25, 35, 40], KV cache management [12, 28, 32, 36, 42], speculative decoding [8, 16, 17, 29, 41], and more. Since ACDC is built upon vLLM, it naturally inherits and benefits from all of its existing and future optimizations. We revisit representative techniques in the context of offline LLM inference serving and show that careful selection and combination of these techniques is essential for task-heterogeneous offline workloads.

Optimizations for Offline LLM inference and hybrid serving. With the emergence of offline LLM inference workloads, there is growing research interest in optimizing offline inference. BlendServe [43] and BatchLLM [45] highlight the importance of reusing common prefixes among offline requests to improve throughput. Other studies, such as Echo [33], HyGen [31], and ConServe [27], propose colocating online and offline requests to improve resource utilization while meeting SLOs, with preemption as the primary scheduling mechanism. We acknowledge these efforts and go further to tackle a more complex and practical scenario involving diverse model types, heterogeneous workloads, and large-scale deployment. We validate the effectiveness of prefix-aware KV cache reuse and other optimization techniques tailored to offline inference workloads in a task-aware manner. We also explore opportunities for online-offline co-scheduling in the structurally distinct PD-disaggregated setting using real production statistics. Specifically, we demonstrate that idle online prefill bubbles can be effectively harvested and propose a novel probability-guided strategy to maximize this benefit.

8 Conclusion

This paper presents ACDC, the first systematic study of offline LLM inference. We identify its unique characteristics and show that directly applying online-serving (or classical offline-serving) optimizations is suboptimal. Through tailored optimizations, ACDC achieves up to 3.9× higher output cost-efficiency in production. We also propose a novel hybrid-serving mechanism that harvests resource bubbles from online serving, improving offline throughput by 35.7% over the enhanced alternative.

Acknowledgments

We sincerely thank the anonymous reviewers and our shepherd for their constructive feedback. We also thank engineers from Alibaba Model Studio for their support with infrastructure and data access. This work is supported by Alibaba Group through the Alibaba Innovative Research Program, and Erci Xu's work is funded by WDZC20265290107.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. arXiv:2403.02310 [cs.LG]

[2] Anthropic. 2023. Claude AI. https://claude.ai/.

[3] Anthropic. 2025. Claude Code. https://docs.anthropic.com/en/docs/claude-code

[4] Anthropic. 2025. Introducing the Message Batches API. https://docs.anthropic.com/en/docs/build-with-claude/message-batches

[5] AWS. 2025. Amazon Bedrock pricing. https://aws.amazon.com/bedrock/pricing/

[6] AWS. 2025. Process multiple prompts with batch inference. https://docs.aws.amazon.com/bedrock/latest/userguide/batch-inference.html

[7] Siyuan Chen, Zhipeng Jia, Samira Khan, Arvind Krishnamurthy, and Phillip B. Gibbons. 2025. SLOs-Serve: Optimized Serving of Multi-SLO LLMs. arXiv:2504.08784 [cs.DC] https://arxiv.org/abs/2504.08784

[8] Ziyi Chen, Xiaocong Yang, Jiacheng Lin, Chenkai Sun, Jie Huang, and Kevin Chen-Chuan Chang. 2023. Cascade Speculative Drafting for Even Faster LLM Inference. arXiv:2312.11462 [cs.LG]

[9] Databricks. 2025. Introducing Simple, Fast, and Scalable Batch LLM Inference on Databricks Model Serving. https://www.databricks.com/blog/introducing-simple-fast-and-scalable-batch-llm-inference-mosaic-ai-model-serving

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. doi:10.1145/1327452.1327492

[11] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[12] In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. arXiv:2311.04934 [cs.CL] https://arxiv.org/abs/2311.04934

[13] Xiannan Hu, Tianyou Zeng, Xiaoming Yuan, Liwei Song, Guangyuan Zhang, and Bangzheng He. 2025. BestServe: Serving Strategies with Optimal Goodput in Collocation and Disaggregation Architectures. arXiv:2506.05871 [cs.LG] https://arxiv.org/abs/2506.05871

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 59–72. doi:10.1145/1272998.1273005

[15] 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. doi:10.1145/3600006.3613165

[16] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. arXiv:2211.17192 [cs.LG]

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

[18] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[19] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. doi:10.1145/2517349.2522738

[20] NVIDIA. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM.

[21] OpenAI. 2023. ChatGPT. https://chat.openai.com/.

[22] OpenAI. 2025. Batch API. Process jobs asynchronously with Batch API. https://developers.openai.com/api/docs/guides/batch

[23] OpenClaw. 2026. Openclaw. https://openclaw.ai

[24] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. arXiv:2311.18677 [cs.AR] https://arxiv.org/abs/2311.18677

[25] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Shengkun Cui, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2024. One Queue Is All You Need: Resolving Head-of-Line Blocking in Large Language Model Serving. arXiv:2407.00047 [cs.DC] https://arxiv.org/abs/2407.00047

[26] Perplexity. 2025. Perplexity. https://www.perplexity.ai/

[27] Yifan Qiao, Suyi Li, Haoran Ma, Shan Lu, Miryung Kim, and Harry Xu. 2026. ConServe: Harvesting GPUs for LLM Inference and Model Training. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*. arXiv:2410.01228 [cs.DC] https://arxiv.org/abs/2410.01228

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

[29] Benjamin Spector and Chris Re. 2023. Accelerating LLM Inference with Staged Speculative Decoding. arXiv:2308.04623 [cs.AI]

[30] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 173–191. https://www.usenix.org/conference/osdi24/presentation/sun-biao

[31] Ting Sun, Penghan Wang, and Fan Lai. 2025. HyGen: Efficient LLM Serving via Elastic Online-Offline Request Co-location. arXiv:2501.14808 [cs.DC] https://arxiv.org/abs/2501.14808

[32] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association. https://www.usenix.org/conference/atc25/presentation/wang-jiahao

[33] Zhibin Wang, Shipeng Li, Xue Li, Yuhang Zhou, Zhonghui Zhang, Zibo Wang, Rong Gu, Chen Tian, Kun Yang, and Sheng Zhong. 2025. Echo: Efficient Co-Scheduling of Hybrid Online-Offline Tasks for Large Language Model Serving. arXiv:2504.03651 [cs.DC] https://arxiv.org/abs/2504.03651

[34] 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* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. doi:10.1145/3694715.3695948

[35] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)* (Seoul, Republic of Korea, October 13–16). ACM, New York, NY, USA, 1–16. doi:10.1145/3731569.3764815

[36] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.

[37] Shunyu Yao, Jeffrey Zhao, Dian Yu, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

[38] 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

[39] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. doi:10.1145/2934664

[40] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. 2025. BLITZSCALE: Fast and Live Large Model Autoscaling with O(1) Host Caching. arXiv:2412.17246 [cs.DC] https://arxiv.org/abs/2412.17246

[41] Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2023. Draft & Verify: Lossless Large Language Model Acceleration via Self-Speculative Decoding. arXiv:2309.08168 [cs.CL]

[42] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. 2023. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 34661–34710. https://proceedings.neurips.cc/paper_files/paper/2023/file/6ceefa7b15572587b78ecfcebb2827f8-Paper-Conference.pdf

[43] Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. 2024. BlendServe: Optimizing Offline Inference for Auto-regressive Large Models with Resource-aware Batching. arXiv:2411.16102 [cs.LG] https://arxiv.org/abs/2411

.16102

[44] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2025. SGLang: Efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.

[45] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2025. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. arXiv:2412.03594 [cs.CL] https://arxiv.org/abs/2412.03594

[46] 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. 193–210 pages. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[47] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. 2025. MegaScale-Infer: Efficient Mixture-of-Experts Model Serving with Disaggregated Expert Parallelism. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 592–608. doi:10.1145/3718958.3750506