Axon: Asynchronous Global KV-Aware Scheduling Across Distributed Instances for Agentic Workloads

Yuqi Qiu*
University of Macau
Macau SAR, China
yc47436@connect.um.edu.mo

Jinting Zou*
University of Macau
Macau SAR, China
mc45059@um.edu.mo

Zizhao Mo†
Great Bay University
Dongguan, China
University of Macau
Macau SAR, China
yc17461@connect.um.edu.mo

Junlin Chen
University of Macau
Macau SAR, China
yc57440@um.edu.mo

Liao Chen
University of Macau
Macau SAR, China
yc27428@connect.um.edu.mo

Huanle Xu†
University of Macau
Macau SAR, China
huanlexu@um.edu.mo

Guoyao Xu†
Alibaba Group
Hangzhou, China
yao.xgy@alibaba-inc.com

Guodong Yang
Alibaba Group
Hangzhou, China
luren.ygd@taobao.com

Liping Zhang
Alibaba Group
Hangzhou, China
liping.z@alibaba-inc.com

ChengZhong Xu
University of Macau
Macau SAR, China
czxu@um.edu.mo

Abstract

Large language model (LLM) applications are increasingly moving toward agentic workflows, where models interact with users and tools over many turns, causing context lengths to dominate sequence composition and accumulate across dozens of turns. Existing serving systems fail to address this setting because they often overlook the reuse of KV caches across distributed instances within a cluster.

We present Axon, an asynchronous global serving architecture that treats distributed GPU HBM as a unified KV cache resource pool for multi-turn agentic workloads. Axon introduces Asymmetric Query Context Parallelism to distribute attention over context shards across instances by communicating only query tensors, eliminating the synchronization bubbles and heavy KV traffic of conventional sequence-parallel designs. Its fully asynchronous scheduler decouples new-request allocation, KV cache restoration, and batch preparation from GPU execution, leveraging aggregated PCIe bandwidth to restore missing fragments in parallel, while a context-locality-aware scheduling policy jointly optimizes cache hit ratio and per-instance load. Evaluation on production agent traces shows that Axon reduces prefill latency by up to 4.14×, decode latency by 1.87×, and end-to-end latency by 2.27× compared to state-of-the-art systems.

ACM Reference Format:
Yuqi Qiu, Jinting Zou, Zizhao Mo, Junlin Chen, Liao Chen, Huanle Xu, Guoyao Xu, Guodong Yang, Liping Zhang, and ChengZhong Xu. 2026. Axon: Asynchronous Global KV-Aware Scheduling Across Distributed Instances for Agentic Workloads. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857095

1 Introduction

LLM applications are increasingly moving from single-shot question answering toward agentic workflows, where models reason, call tools, observe external results, and continue the interaction over many turns. Coding assistants and autonomous agents such as Claude Code [4] and OpenClaw [30] exemplify this shift. In these workloads, each new turn is not processed in isolation: the request usually includes the newly appended prompt, prior messages, intermediate reasoning traces, tool outputs, and earlier responses. Preserving this history is essential for coherent and context-aware behavior [33], but it turns serving into a long-context, multi-turn inference problem.

This workload pattern creates a severe bottleneck for LLM serving systems. Each request proceeds through a prefill phase, which computes key-value (KV) cache states for the input sequence, followed by an autoregressive decode phase, which reuses those cached states to generate output tokens. As an agent session progresses, the accumulated context quickly dominates the sequence. Our trace analysis of one

*Co-first authors: both authors contribute equally.
†Corresponding authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, Hong Kong
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857095

week of production coding-agent traffic shows that sessions contain 20+ turns on average. The resulting contexts are long: the average context length reaches 33.5K tokens and the P95 reaches 78.3K tokens, accounting for 84.1% of the total sequence on average. In contrast, newly appended prompts are moderate in size, averaging 3.7K tokens, and model outputs are short, averaging only 270 tokens. Thus, the dominant cost in agent serving is repeatedly managing and reusing large historical contexts, not generating long responses.

Existing long-context serving systems do not fully address this setting. Sequence-parallel serving frameworks accelerate long requests by distributing computation across instances [40], but they typically treat each request as independent and recompute KV cache states from scratch. This misses the substantial reuse opportunity across turns within the same session. Multi-turn cache management systems such as CachedAttention [12] and HCache [13] improve state restoration within a single instance by combining hierarchical storage, transfer scheduling, and recomputation. However, they assume that a request's reusable KV cache is restored locally. This assumption breaks down in production clusters, where load balancing naturally routes different turns of the same session to different instances.

On the other hand, distributed KV cache systems adhere to a *data-to-computation* paradigm. Mooncake [32] and LMCache [24] tier each instance's storage into a cluster-wide hierarchy, so KV state can be retained across instances rather than within one. Execution, however, remains confined to the instance that admits a request, so fragments cached elsewhere must be aggregated onto it before computation, incurring a heavy communication cost on every turn. The cluster-wide distribution of KV state is thus treated as a cost to be undone rather than a locality to be exploited. Moreover, each request is still served serially by one instance, leaving long-context requests bottlenecked by single-node capacity.

Our trace analysis shows that this distributed-cache behavior is common. Specifically, about 65% of agentic sessions are routed to more than five distinct instances for prefill computation. As a result, fragments of a session's KV cache can be scattered across the GPU memory of many instances. Treating each instance independently either sacrifices KV cache reuse, forcing expensive recomputation or host-to-device restoration, or pins long sessions to individual instances, causing memory fragmentation and load imbalance. Agent serving therefore requires a cluster-level design that can reuse distributed KV fragments locally while keeping the instance pool balanced.

This observation reveals two opportunities. First, the GPU HBM across instances can be managed as a global memory pool. A cluster scheduler with global visibility can employ *computation-to-data* scheduling to improve the cache hit ratio—that is, by placing computation on nodes where their existing KV cache fragments reside. This can also shard ultra-long contexts across multiple instances, alleviating per-instance memory fragmentation. Second, when cache fragments must be restored from CPU memory, multiple instances can restore different fragments in parallel, exploiting the aggregate PCIe bandwidth of the cluster rather than being limited by one instance's host-to-device bandwidth.

However, realizing these opportunities is challenging. Maximizing KV cache reuse can conflict with load balancing: routing requests only to cache-rich instances may overload them, whereas routing only to lightly loaded instances may discard reuse benefits. Global scheduling can also become a bottleneck in large-scale clusters if every batching or decoding iteration depends on the scheduler. Finally, scaling a single request across multiple instances can introduce synchronization bubbles: if all participants must progress in lockstep, a slow or busy instance can stall the entire request and interfere with unrelated in-flight batches.

To tackle the challenges, we present Axon, an agentic LLM serving system that provides asynchronous global scheduling across distributed instances with KV-locality awareness. Axon's scheduler is fully asynchronous: it determines prompt placement and distributed context sharding for newly arrived requests, while proactively generating future batches for ongoing inference in parallel. In addition, the scheduler facilitates concurrent KV cache restoration and migration through a unified GPU/CPU memory pool, leveraging aggregated PCIe bandwidth without interrupting ongoing inference. This separation prevents both new-request scheduling and cache restoration from blocking GPU execution.

To efficiently execute requests whose contexts span multiple instances, Axon introduces Asymmetric Query Context Parallelism (AQCP). AQCP exploits a key property of agent workloads: newly appended prompts are much shorter than accumulated contexts. For each request, Axon selects one instance as the `AxonMaster` to perform the linear operations for the new prompt, while multiple `AxonLets` compute attention over their local context fragments. AQCP also provides request-level flexibility: requests within the same batch may adopt different asymmetric context-sharding plans across distributed instances. By communicating only query tensors rather than transferring full KV tensors, AQCP preserves distributed KV cache reuse while avoiding the heavy communication and synchronization overheads.

Building on asynchronous scheduling mechanism and AQCP, Axon introduces a context-locality-aware scheduling policy, aiming to reduce the Time-To-First-Token (TTFT) of newly arrived requests while limiting the additional Inter-Token Latency (ITL) imposed on ongoing requests. To handle these coupled objectives, Axon adopts a two-phase scheduling strategy for each new request. The scheduler first selects an `AxonMaster` from feasible low-load instances, then refines the context sharding plan by formulating a min-max load-balancing problem over participating instances. Because this problem follows a load-leveling structure, Axon solves it with a lightweight balancing procedure that shifts context

work across GPU-resident cache fragments and CPU memory. In particular, Axon prioritizes CPU-to-GPU restoration on each instance, allowing multiple PCIe links to be used in parallel while avoiding unnecessary GPU-to-GPU transfers. When GPU-to-GPU migration is required, Axon formulates a small matching problem among the associated instances to minimize communication overhead.

We have implemented Axon on top of vLLM [19], incorporating session-level information into the global scheduler to better support agentic workloads. Moreover, we conduct extensive experiments on a local cluster with four A100 GPU servers, using production agent traces. Experimental results on LLaMA-2 and Qwen models show that Axon reduces TTFT by up to 4.14× and ITL by 1.87×, while enhancing serving capacity by 2.25×. Moreover, the asynchronous scheduler introduces minimal system overhead when scaling to clusters with hundreds of instances. In summary, this paper makes the following contributions:

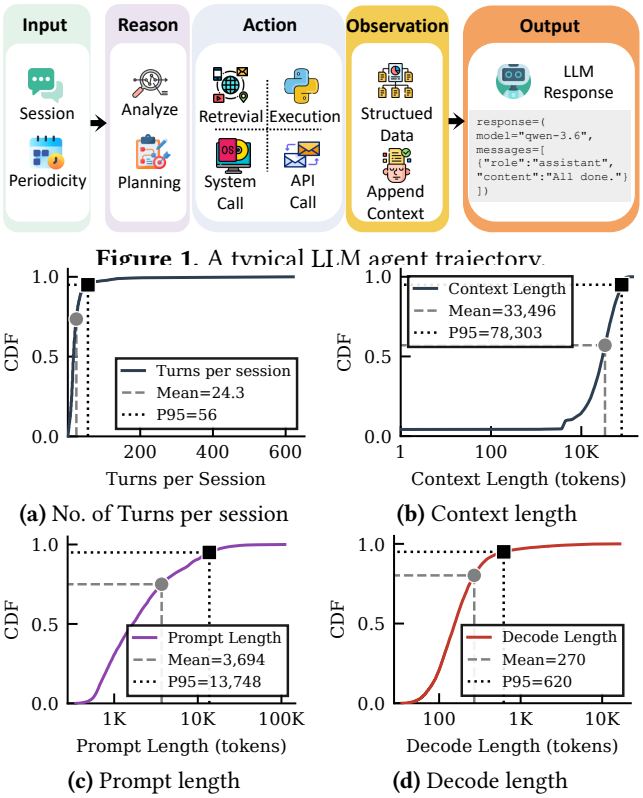
- We characterize production agent-serving traces and show the importance of reusing KV caches across distributed serving instances within a cluster.
- We design Axon, a fully asynchronous global serving architecture that manages distributed GPU HBM as a global KV cache resource while decoupling request allocation, cache restoration, and batch preparation.
- We introduce Asymmetric Query Context Parallelism, which enables requests to reuse context fragments locally across multiple instances through asymmetric context-sharding plans across requests within a batch.
- We develop a context-locality-aware load-balancing policy that jointly optimizes the KV-cache hit ratio and load balance across distributed instances.

2 Background and Motivations

2.1 LLM Agent Serving

ReAct loop. LLM agent serving typically follows the ReAct (Reasoning–Action–Observation) loop [44], as shown in Fig. 1. Upon receiving a user request, the agent enters the *reasoning stage*, where it analyzes the prompt and devises a plan of tool calls tailored to the execution environment. In the *action stage*, the agent invokes APIs sequentially within a sandbox and awaits the execution results. Once the tool outputs return, the agent aggregates/analyzes the results in the *observation stage*, then decides whether to trigger another ReAct loop or emit the final answer. Each loop appends its intermediate traces to the conversation history, collectively forming the context that participates in future serving.

Multi-turn LLM Inference. A complete user–agent interaction—referred to as a *session*—unfolds over multiple turns, where each turn consists of a user prompt (typically augmented with a ReAct trace) and the agent's corresponding response. When a new turn arrives, the serving system processes the entire input sequence—including both the



(a) No. of Turns per session **(b)** Context length **(c)** Prompt length **(d)** Decode length

Figure 2. Characterization of multi-turn agentic workloads from Alibaba copilot agent.

fresh prompt and the accumulated context from previous turns—as an LLM request. This computation proceeds in two phases. During *prefill*, all tokens in the input sequence are processed in parallel to compute their KV cache. During *decode*, the model generates response tokens autoregressively, one by one, reusing the cached KV states at each step. Because agents often need many turns to solve complex tasks (interacting with users and external tools repeatedly), the context grows rapidly across turns. These defining traits of agent workloads—multiple turns and long contexts within a session—impose storage pressure and substantial computational demands on the serving infrastructure.

2.2 LLM Agent Serving Characteristics

To meet substantial computational demands, cloud providers typically deploy multiple LLM instances—each running a complete version of the model—to serve ever-increasing agentic workloads. Consequently, a single session may be served across different instances over multiple turns [10, 48]. To better understand the bottlenecks of agent serving in production, we analyze one week of agentic serving traces from Alibaba's enterprise coding copilot agent Aone [5].

High multi-turn interaction counts within a session. Fig. 2 shows the distribution of turn counts across all sessions, revealing the striking multi-turn nature of agent workloads. The average session comprises 24.3 turns, and over 10% of sessions exceed 37 turns. More significantly, the P95 turn

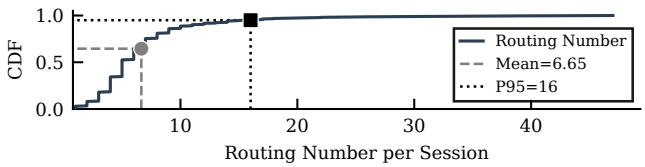


Figure 3. The number of serving instances for all sessions.

count spikes to 56, indicating a pronounced tail that differs from traditional long-context LLM serving scenarios [40].

Long contexts, medium prompts, but short outputs. We decompose each turn into three subsequences—context, newly appended prompt, and LLM output—and analyze their composition in detail. As shown in Fig. 2, the average context length can reach 33.5K, with the P95 reaching 78.3K. On average, the context accounts for approximately 84.1% of the total sequence length. This indicates that agent serving is dominated by long contexts. Meanwhile, the average prompt length is 3.7K tokens, accounting for a non-negligible 15.8% of the sequence. In contrast, the output length is relatively short, only contributes 270 tokens per sequence on average.

Distributed KV caches over multiple instances. The increasing context length and the large number of agent sessions make it difficult to persistently serve an individual session within a single instance across all its turns. Once an LLM request is processed on a new instance that its previous turns have not been scheduled on, the KV cache of the preceding contexts is restored via recomputation. For each session, we analyze the number of instances it is processed over its lifecycle. Fig. 3 shows that about 65% of sessions are routed to more than 5 distinct instances for prefill computation, and nearly 5% of sessions are routed to more than 16 instances. This means that multiple instances may contain overlapping fragments of the same session's KV cache.

Takeaway. Our analysis reveals a distinctive profile of LLM serving in multi-turn agentic workloads. The wide distribution of a session's KV cache across instances makes it critical to design scheduling algorithms that can efficiently place multiple turns of each session onto distributed instances in order to maximize KV cache reuse.

2.3 Limitations of Existing Solutions

Existing solutions have devoted significant effort to optimizing long-text LLM serving; however, they fall short of supporting agentic workloads across distributed instances.

Long-sequence serving systems fail to optimize KV cache reuse. Recent long-sequence inference frameworks focus on parallelizing both prefill and decode computations across instances using sequence parallelism [15, 18, 22, 23, 43]. LoongServe [40] is a representative system. While these systems are effective for long sequences, they process each request independently, ignoring the KV cache sharing among requests. Consequently, they need to compute the KV cache for all requests from scratch. As the number of turns increases, most prefill latency is spent on context recomputation, degrading inference latency for agentic workflows.

Single-instance serving cannot exploit distributed KV cache. Current multi-turn serving systems focus on rapidly restoring the KV caches of each ongoing request during inference to support low latency. Representative methods such as CachedAttention [12] and HCache [13] design elaborate data-transfer (from host memory and disks) and computation-overlapping strategies to restore KV caches for long contexts. One key limitation is that they only restore KV caches within a single instance, neglecting the fact that a session's KV cache can be distributed across multiple instances due to request scheduling. Forcing each agent session to be served on a single instance can easily lead to GPU memory fragmentation because of extremely long context lengths.

Distributed KV cache systems cannot compute on the state they distribute. Another line of work, Mooncake [32] and LMCache [24], expands the storage of each instance with hierarchical tiers. These systems adhere to a *data-to-computation* paradigm. Because execution is confined to the instance that admits a request, KV fragments cached elsewhere must be aggregated onto it before execution, incurring a heavy communication cost on every turn. Additionally, each request is served serially by a single instance, so long-context requests are bottlenecked by the single node capacity.

Takeaway. Existing systems fail to schedule agentic workloads efficiently across many instances, resulting in poor KV reuse and low memory utilization; and cannot exploit distributed KV reuse in cluster-wide computation.

2.4 Opportunities and Challenges

2.4.1 Opportunities. Recognizing the multi-turn nature of agentic workloads and the fact that multiple instances may contain overlapping fragments of the same session's KV cache (§ 2.2), we identify two key opportunities that can be leveraged to efficiently serve agentic workflows across distributed instances.

Global locality-aware scheduling on a unified HBM pool can improve cache hit ratio and alleviate memory fragmentation. In production clusters, a large number of context KV cache fragments become scattered across the GPU HBM of different instances. If GPU HBM is managed as a unified pool, global scheduling can follow the *computation-to-data* paradigm and dispatch each request's attention to the instances that already hold its KV fragments. Its guiding principle is that *attention should run where the KV already resides*, rather than the KV being aggregated to wherever the computation happens to run. This significantly improves the cache hit ratio for LLM requests and eliminates the prohibitive overhead of frequently restoring contexts from hierarchical storage or via recomputation. Furthermore, by sharding ultra-long contexts across multiple instances, memory fragmentation on a single instance can be greatly alleviated, thereby improving resource efficiency.

Aggregated PCIe bandwidth can multiply KV cache restoration speed. As computation of an agent session proceeds, fragments in GPU HBM may be evicted to free space for other requests, leading to cache misses. When a KV cache needs to be restored from CPU memory, we can parallelize the restoration across multiple instances to leverage the aggregate PCIe bandwidth of the cluster, thereby avoiding the bottleneck of single-instance restoration. The aggregate PCIe bandwidth scales proportionally with the number of participating instances, enabling a linear restoration speedup and significantly reducing recovery time.

2.4.2 Challenges. Despite the opportunities, several challenges remain in achieving optimal global scheduling of all agentic sessions across distributed instances.

Simultaneously optimizing KV cache hit ratio and load balancing is difficult. Although distributing contexts across a large number of instances can improve the overall cache hit ratio and accelerate KV cache restoration, excessively sharded contexts introduce substantial communication overhead, thereby increasing iteration latency. Moreover, always routing requests to instances that already hold their contexts can lead to uneven load, potentially degrading overall performance. Conversely, if requests are directed to lightly loaded instances without regard for KV cache affinity, context reconstruction sacrifices reuse opportunities.

Cluster-wide scheduling can easily become a bottleneck. As the cluster size grows in production, the global scheduler must determine the placement of each LLM request across a large number of instances. Under heavy load—where decoding iterations are ongoing and new turns keep arriving across a huge number of sessions—the scheduling overhead can become extremely high. This overhead can easily block inference on GPUs, because batching must be performed before both prefill and decode computations. Computation can be further delayed if the required KV cache needs to be restored from host memory.

Synchronization across distributed instances can cause bubbles. Scaling out individual requests across instances tightly couples the participating instances and introduces substantial synchronization overhead. When a context is sharded across instances, all participants must progress layer by layer, so performance jitter on any single instance can stall the entire request and create pipeline bubbles. This coupling also complicates scheduling. In particular, coordinated KV-cache restoration for new sessions can delay the computation of in-flight batches: conventional designs often force them to proceed in lockstep, leading to high synchronization costs.

3 Axon Overview

3.1 Key Design Ideas

To address the challenges, we design Axon, an agent-oriented serving system built upon the following key ideas:

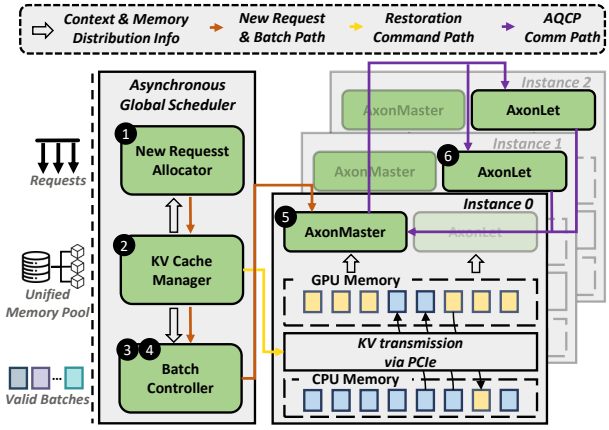


Figure 4. The system architecture of Axon.

Asynchronous scheduling over large-scale cluster. The global scheduler in Axon only makes decisions for new requests, completely avoiding involvement in per-iteration or per-batch scheduling of ongoing inference. To prevent this limited decision-making from becoming a bottleneck, Axon further decouples new-request allocation from periodic batching. It proactively generates future batches and triggers HBM management without waiting for current batches to finish, while separating the scheduling of bursty arrivals from the already stressed batching process.

Eliminating synchronization via Asymmetric Query Context Parallelism. This new parallelism mechanism supports request-level flexibility: requests within the same batch can adopt different asymmetric context-sharding plans across distributed instances. Moreover, it mitigates synchronization overhead by confining all linear computation (e.g., matrix multiplications) to a single AxonMaster instance per request, while distributing attention computation across multiple AxonLet instances. Additionally, it decouples KV cache restoration from instance execution, allowing context preparation to overlap seamlessly with computation.

KV-cache-locality-aware global load balancing. To maximize local KV cache reuse while maintaining global load balance, Axon uses a two-phase scheduling policy that dynamically determines optimal context sharding and prompt placement based on real-time cluster workload. It seeds this optimization with global context distribution data, then quantifies inference latency for candidate placements, migrating context shards across instances to balance load while preserving KV cache affinity and reducing KV cache transfers.

3.2 System Architecture

As shown in Fig. 4, Axon features an asynchronous global scheduler composed of three modules: New Request Allocator, KV Cache Manager, and Batch Controller. Additionally, each local instance is equipped with AQCP parallelism, which includes one AxonMaster and one AxonLet.

During serving, ❶ the scheduler's New Request Allocator processes incoming requests in FIFO order, determining optimal prompt placement and context sharding by solving

optimization objectives. ❷ The resulting allocations are forwarded to the KV Cache Manager, which restores KV caches in parallel across instances via the unified memory pool. To hide restoration latency, the Manager maintains a restoration list enabling asynchronous concurrent cluster-wide recovery. ❸ Once restoration is complete, requests are admitted to the Batch Controller and formed into batches. ❹ Even in the absence of new arrivals, the Batch Controller proactively generates batches and manages memory for ongoing inference requests, thereby preventing execution stalls. ❺ On each AQCP-enabled instance, AxonMasters periodically pull batches from the Batch Controller for local execution. Specifically, ❻ AxonMasters execute linear and local attention operations, while AxonLets handle remote invocations to compute distributed attention over local KV cache fragments, eliminating cross-instance synchronization barriers.

4 Axon Design

In this section, we first present the asynchronous global scheduler (§ 4.1), followed by our proposed Asymmetric Query Context Parallelism (§ 4.2). We then introduce a context-distribution-aware load-balancing algorithm (§ 4.3).

4.1 Asynchronous Global Scheduling

The scheduler in Axon takes a global view of cluster resources. We describe its asynchronous scheduling for new and ongoing requests, along with KV cache management.

4.1.1 New request scheduling. The New Request Allocator manages incoming requests and generates scheduling decisions. It maintains a **`FIFO`** waiting queue and, for each request, determines two key components of the scheduling decision: the prompt allocation, which selects the instances responsible for processing the new prompt, and the context sharding plan, which specifies how the reusable context is distributed across instances. During scheduling, the Allocator queries the KV Cache Manager to retrieve the session's context distribution and the available GPU blocks on each instance. Once scheduling is complete, the Allocator sends the decision to the KV Cache Manager, which uses it to guide KV-cache restoration and subsequent computation.

By running the Allocator for new requests in a dedicated process, Axon decouples new request scheduling from inference request batching, which is handled by the Batch Controller. This allows the Allocator to absorb bursty arrivals asynchronously, while the Batch Controller continues generating batch metadata for all active instances. Under high request rates and at large cluster scales, such decoupling improves system throughput and responsiveness by preventing new-request scheduling from blocking ongoing inference batching.

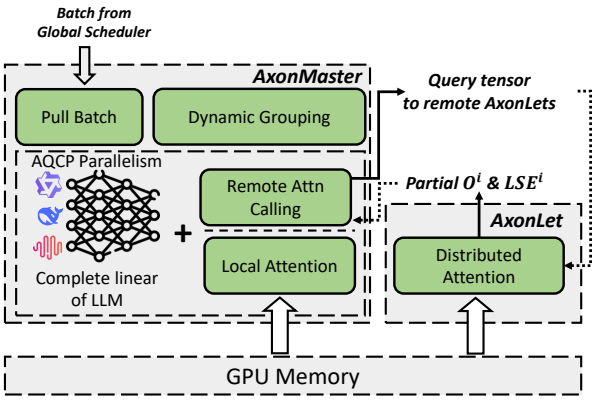
4.1.2 Global KV cache management. The KV Cache Manager efficiently manages and migrates KV caches across the cluster by integrating CPU and GPU memory into a unified memory pool. Specifically, it maintains global KV cache placement information and tracks each request's KV cache throughout its lifetime. The Manager updates cache placement in response to runtime events such as cache eviction, new request admission, and autoregressive token generation, while also tracking the availability of cache blocks on each device. As a result, when the Allocator reuses context for a new request, or the Batch Controller requires memory for a decoding request, the Manager provides the necessary placement and availability information for global memory allocation. This design mitigates memory fragmentation and enables cluster-wide context reuse orchestration.

The KV Cache Manager also incorporates a context Restoration Controller to support asynchronous cache restoration. It maintains the restoration state for each involved request. Specifically, upon receiving a restoration signal for a new request from the global scheduler, the Restoration Controller enqueues the request and marks its state as **`Init`**. Additionally, when the Batch Controller requests GPU memory adjustment due to growing decode demands, the Manager first evicts unused KV caches from GPU to host memory following the widely-adopted LRU (Least-recently-used) policy [7]. If this is insufficient, the Manager may further evict the KV cache of an ongoing request; however, this case is rare because decode-phase generations are typically short, as analyzed in § 2.2. The evicted request is then enqueued by the Restoration Controller and marked as **`Init`**. The Restoration Controller next identifies reusable context and issues commands to the instances to migrate the required hit-miss fragments from host memory to GPU memory if needed. Once KV cache transmission begins, the state changes to **`Restoring`**. After transmission completes, the state is marked as **`Finished`**, signaling the Batch Controller to admit the request for subsequent computation. This design allows multiple inference requests (each potentially spanning multiple instances) to restore contexts concurrently, fully utilizing PCIe bandwidth.

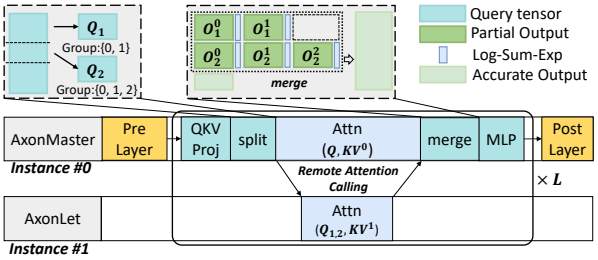
To facilitate cache restoration, each instance exposes an interface that the KV Cache Manager can invoke. Once called, the interface performs cache movement in the background non-interruptibly, mitigating interference with computation.

4.1.3 Proactive batching. Once a request completes either context restoration (guided by the scheduler to schedule a new request) or memory adjustment (guided by the Batch Controller to handle growing decode demands), it is enqueued into the ready queue of the Batch Controller. The Batch Controller then generates inference metadata for requests in each instance—**`slot mapping`** and **`block table`**. This metadata specifies, for each request in the batch, the slots for storing new KV tokens and the locations of historical KV, both of which are essential for attention computation.

However, global scheduling introduces non-negligible challenges to inference efficiency. Although inference computation is decoupled from metadata preparation in the Batch Controller, the inference process may still experience stalls.



(a) AQCP serving architecture



(b) Pipelined execution

Figure 5. Distributed serving architecture and workflow of Asymmetric Query Context Parallelism.

This is because the Batch Controller is unaware of when inference computation will complete, and may therefore fail to prepare metadata in time for subsequent batch processing, leaving the instance idle. Moreover, the network communication overhead incurred when delivering metadata further prolongs this idle period.

To address these issues, the Batch Controller adopts a proactive batching design that starts preparing metadata for distributed instances before it is actually required. To achieve this, Axon introduces a new coordination scheme between instances and the Batch Controller, which keeps track of the metadata status over time. Initially, the Batch Controller generates the metadata for an instance and sets its status to **`Valid`**. Once this metadata is consumed by the instance for batch processing, its status automatically shifts to **`Used`**. At the same time, the Batch Controller is notified to generate new metadata in advance. After the new metadata is generated, it overwrites the old one and sets the status back to **`Valid`**. Axon designs an interface for the interaction between the Batch Controller and distributed instances. Specifically, the instance starts fetching the metadata for the $(k+1)$-th iteration from the Batch Controller as the computation of the k-th iteration begins.

4.2 Asymmetric Query Context Parallelism

In this section, we introduce Asymmetric Query Context Parallelism (AQCP), the mechanism through which Axon ships computation to the data.

4.2.1 Fine-grained distributed computation. Agentic workloads exhibit a distinct pattern. As shown in Fig. 2, the average prompt length of agent workloads is only 3.7K tokens, whereas the mean context length can reach as high as 33K. This pattern reveals two key insights. First, a single instance is sufficient to handle the computation of linear operations (e.g., matrix multiplication) even during the prefill phase, as thousands of tokens can be processed promptly. Second, when the KV-cache hit ratio is high, the attention computation between the prompt and the long context becomes the dominant bottleneck.

Based on this observation, Axon develops AQCP. The key of AQCP is to distribute the attention computation between a new prompt and its contexts across multiple instances by sending only the query tensors of the prompt to the participating instances, thereby maximizing KV cache reuse, while confining the linear operations of the new prompt to a single instance. Moreover, AQCP supports request-level flexibility: requests within the same batch can adopt asymmetric context sharding plans across distributed instances. As a result, each instance continues to manage its own assigned inference batch, including both linear operations and attention computation. By restricting the parallelization scope of linear operations, which account for most SM occupancy, AQCP effectively mitigates resource contention when multiple inference workflows from different sessions are consolidated on the same instance. This paradigm differs fundamentally from conventional context parallelism and sequence parallelism, where participating instances typically partition the sequence dimension and must advance synchronously through the layer-wise linear operations, attention computation, and required communication steps for the same batch under continuous batching [46].

As illustrated in Fig. 5(a), Axon assigns two roles to each request: **`AxonMaster`** and **`AxonLet`**. The `AxonMaster` exclusively performs all linear computation, while the `AxonMaster` and `AxonLets` jointly execute attention computation. For each request, only one instance is selected as the `AxonMaster`, whereas multiple instances may serve as `AxonLets`, depending on the metadata prepared by the Batch Controller. Within each instance, the `AxonLet` is designed as a lightweight daemon process that only participates in distributed attention computation. To mitigate the interference introduced by `AxonLets`, Axon leverages NVIDIA Multi-Process Service (MPS) to enable flexible hardware multiplexing. Once its assigned computation is completed, an `AxonLet` proactively yields CPU and GPU control to the `AxonMaster`, thereby minimizing resource contention.

4.2.2 Inference pipeline of AQCP. Each instance can simultaneously serve as an `AxonMaster` for some requests and as an `AxonLet` for others. To fully utilize resources, Axon designs a new computation workflow for each inference

batch. At the beginning of a token-generation iteration, every instance processes the requests that select it as their AxonMaster and registers invocation metadata, such as token lengths and context-fragment locations, with the corresponding AxonLets to enable distributed attention computation. The batch metadata for these requests is fetched from the Batch Controller to ensure smooth execution.

During each iteration, an AxonMaster dynamically groups inference requests according to their AxonLet assignments: requests that share the same set of AxonLets are placed in the same group. As shown in Fig. 5(b), within each model layer, the AxonMaster **broadcasts** the **Query** tensors of each group after the **QKV_Proj** module asymmetrically, thereby triggering distributed attention computation. It then **gathers** the intermediate results from the corresponding AxonLet i, including the partial attention outputs O^i and log-sum-exp values LSE^i, and **merges** them into the final attention output:

$$O = \sum_i O^i \cdot \exp\left(LSE^i - LSE^{\max}\right) \Big/ \sum_i \exp\left(LSE^i - LSE^{\max}\right).$$

The outputs from multiple groups are then assembled and fed into the next module. This one-to-many distributed attention design implicitly imposes a latency requirement: the time spent on locally assigned attention workloads should be longer than that spent on remotely processed workloads. Otherwise, Axon cannot fully saturate the hardware resources of AxonMaster as model execution is blocked by remote peers.

4.3 Context-locality-aware Load Balance

Building upon the asynchronous scheduling mechanism and AQCP, Axon incorporates a new scheduling policy to globally coordinate computation across distributed instances and reduce inference latency. This policy aims to minimize the Time-To-First-Token (TTFT) of each incoming request while limiting the additional Inter-Token Latency (ITL) imposed on existing requests in the cluster.

4.3.1 Optimization formulation. For an incoming request j, Axon makes two coupled scheduling decisions: selecting its AxonMaster $\psi_j \in \mathcal{I}$ and determining its context-restoration sharding plan $\overrightarrow{\mathbf{c}_j}$ across distributed instances. These decisions jointly affect the TTFT of request j and the ITL of ongoing requests. Intuitively, assigning request j to lightly loaded instances reduces interference with existing decoding iterations, while assigning it to instances that already hold reusable KV cache fragments reduces context-restoration latency.

Let $\mathcal{I}$ denote the set of candidate instances, and let $\mathcal{J}^i(t)$ denote the set of active requests whose AxonMaster is instance i at time t. We use $\mathcal{M}^i(t)$ to denote the current memory usage of instance i, $\mathcal{M}^i(\psi_j, \overrightarrow{\mathbf{c}_j})$ to denote the additional memory usage incurred by placing request j under a given allocation, and $\mathcal{M}^i_{\max}$ to denote the memory capacity of instance i. The overall scheduling objective can be formulated as the following multi-objective optimization problem:

$$\min_{\psi_j, \overrightarrow{\mathbf{c}_j}} \quad \left\{ \mathsf{TTFT}\left(\psi_j, \overrightarrow{\mathbf{c}_j}\right), \mathsf{ITL}\left(\psi_j, \overrightarrow{\mathbf{c}_j}, \{\mathcal{J}^i(t)\}_{i\in\mathcal{I}}\right) \right\} \tag{1}$$

$$\text{s.t.} \quad \mathcal{M}^i(t) + \mathcal{M}^i(\psi_j, \overrightarrow{\mathbf{c}_j}) \le \mathcal{M}^i_{\max}, \quad \forall i \in \mathcal{I}.$$

To handle the coupling between these objectives and the scale of large clusters, Axon adopts a two-phase heuristic that produces effective scheduling decisions with low overhead. For each incoming request j, the scheduler first selects an instance ψ_j as its AxonMaster according to the real-time load conditions $\{\mathcal{J}^i(t)\}_{i\in\mathcal{I}}$. Once ψ_j is fixed, the scheduler determines a context-restoration sharding plan that reduces the additional ITL imposed on ongoing requests while preserving KV cache reuse for request j.

4.3.2 Determining AxonMaster. In the first phase, Axon selects the AxonMaster for each incoming request, processing requests in FIFO order. The goal is to limit the scheduling impact caused by admitting a new prefill request. Specifically, for an incoming request j, Axon considers only instances whose remaining KV-cache capacity can accommodate the request, and assigns j to the instance with the smallest $n^i_j(t)$, where $n^i_j(t)$ denotes the number of live requests currently hosted on instance i. This criterion is particularly suitable for agent workloads: because their prompt lengths are typically moderate and their decode lengths are short, the number of live requests becomes a dominant factor affecting latency. Therefore, by favoring the least-loaded feasible instance, Axon reduces contention and improves TTFT while satisfying the KV-cache capacity constraint.

4.3.3 Determining context sharding plan. Given the selected AxonMaster, the second phase distributes the context-attention workload across participating AxonLets to minimize the latency imbalance introduced by the new request. This decision must jointly consider two factors: the existing load on each instance and the amount of request j's context that is already locally reusable from GPU memory.

For each instance $i \in \mathcal{I}$ that holds part of request j's context or is selected to process its prompt, let $\mathcal{W}^i$ denote its estimated iteration latency before admitting request j. This estimate captures both roles that an instance may play in AQCP: as an AxonMaster, it performs linear computation and local attention; as an AxonLet, it performs attention over assigned context fragments. Let q_j denote the prompt length of request j, and let h^i_j denote the length of request j's context that is already cached in GPU memory on instance i. Since attention cost scales with the product of prompt length and assigned context length, the additional workload introduced on instance i is proportional to $c^i_j q_j$ where c^i_j is the i-th element of $\overrightarrow{\mathbf{c}_j}$. The assignment $\overrightarrow{\mathbf{c}_j}$ must cover the full context of request j; letting C_j denote the total length of context, the scheduling objective is to minimize the maximum post-assignment latency across candidate instances:

$$\begin{aligned} \min_{\vec{c_j}} \quad & \max_{i \in \mathcal{I}} \left(\mathcal{W}^i + \alpha c_j^i q_j \right) \\ \text{s.t.} \quad & \sum_{i \in \mathcal{I}} c_j^i = C_j, \\ & 0 \le c_j^i \le C_j, \quad \forall i \in \mathcal{I}, \end{aligned} \tag{2}$$

where α is a profiling-derived coefficient that converts attention work into latency. This formulation captures the key trade-off: assigning more context to an instance increases its attention latency, but assigning context to instances that already hold KV caches can reduce restoration overhead.

To account for KV cache reuse, we further decompose the assigned context into two parts. For each instance i, the first $\min(c_j^i, h_j^i)$ tokens can be served from existing GPU-resident KV cache locally, while the remaining

$$r_j^i = \max(c_j^i - h_j^i, 0), \tag{3}$$

tokens must be restored from CPU memory or migrated from another instances. The total amount of missing context is

$$R_j = \sum_{i \in \mathcal{I}} r_j^i. \tag{4}$$

Thus, Axon favors assignments that balance attention latency while keeping R_j small.

The optimal solution to Eq. (2) follows a simple load-leveling structure. Axon therefore uses a lightweight balancing procedure. It first estimates the target post-assignment latency T by averaging the expected total workload over all candidate instances. Instances with $\mathcal{W}^i + \alpha h_j^i q_j > T$ are considered overloaded: although they may already hold useful KV cache fragments, assigning all of those fragments to them would increase the critical path. Instances with $\mathcal{W}^i + \alpha h_j^i q_j \le T$ are considered underloaded and can accept additional restored or migrated context. The scheduler then shifts context work from overloaded instances or host memory to underloaded ones until the estimated post-assignment latencies approach T or the GPU memory is fully occupied.

When additional context must be materialized, Axon prioritizes CPU-to-GPU restoration on underloaded instances, because this allows multiple PCIe links to be used in parallel and increases aggregate restoration bandwidth. In rare cases where the required restored context has already been covered and imbalance remains, Axon migrates GPU-resident KV fragments from overloaded instances to underloaded ones. To reduce communication overhead, the scheduler formulates this migration step as a small matching problem between source instances with excess assigned context and destination instances with available load capacity, minimizing the number of migration edges. This matching is efficient in practice because a session typically spans only a limited number of instances, and the cluster is already approximately load-balanced before request j is admitted.

5 System Implementation

We develop Axon on top of the vLLM framework [19], which provides flexible inference functionality and high extensibility. Our implementation consists of approximately 14K lines of Python, C++, CUDA, and Rust. During development, we found that PyTorch's collective communication APIs introduce substantial CPU overhead in performance-critical paths. Consequently, we replace them with custom communication operators built directly on top of NCCL [29], enabling efficient runtime communication.

Agent-oriented scheduling. We extend vLLM's scheduler to support agent serving. vLLM is session-agnostic: it performs cross-request context reuse through prefix caching [39] independently for each new arrival, which does not scale cost-effectively to cluster-wide context reuse. Axon explicitly introduces a **`session_id`** and maintains direct mappings between session identifiers and their backing storage, providing native session management, with session sequences partitioned by blocks. We also rewrite the GPU and CPU memory managers to track session-granular placement and location metadata across hierarchical storage, enabling cluster-scale context retrieval and storage management.

Control-plane messaging flow. Axon uses ZeroMQ (ZMQ) [49] for control-plane messaging between the scheduler and instances, as well as among instances. Both the scheduler and each instance run a ZMQ server in router mode to handle remote calls from other components. We design specialized message formats for batch metadata and dynamic grouping information, and use shared-memory buffers for intra-instance cross-process data transfer. This design eliminates unnecessary serialization and deserialization overhead, ensuring low-latency control-plane communication.

Tiered KV cache transfer. Axon uses a custom CUDA kernel to transfer KV cache blocks between non-contiguous CPU and GPU memory regions, avoiding redundant memory copies and excessive kernel-launch overhead in conventional CPU–GPU transfer paths. In addition, Axon launches a dedicated process on each instance to service a FIFO queue for new-request KV cache restoration. The same process synchronizes evicted blocks from GPU memory to local CPU memory and broadcasts them to the CPU memory of relevant instances within the same group using ZMQ multicast primitives. This eager backup mechanism enables fast context restoration via exploiting aggregated PCIe bandwidth.

Efficient primitives for AQCP execution. We implement NCCL-based communication primitives for AQCP, including customized broadcast and gather operations, to support dynamic grouping and asymmetric communication while minimizing CPU involvement in communication launch. These primitives leverage NVIDIA GPUDirect RDMA [31] to enable low-latency GPU-to-GPU data transfer. We also design a fused merge CUDA kernel that aggregates partial

attention outputs from multiple instances in a single GPU invocation, avoiding the overhead of repeated kernel launches.

6 System Evaluation

6.1 Experimental Setup

TestBeds. We conducted experiments on a local cluster of 4 physical hosts, each equipped with 2 Intel Xeon Gold 6342 CPUs, 4 NVIDIA A100-SXM4-80GB GPUs, and 256 GB of DRAM. Each GPU connects to its host via PCIe Gen 4 ×16, offering a theoretical unidirectional bandwidth of 31.5 GB/s. All hosts are interconnected via a 100 Gbps GPUDirect RDMA network, and intra-host GPU communication traverses the PCIe interconnect. We integrated Tensor Parallelism (TP) into the evaluated models to support larger models and longer contexts, restrict each model instance to a single host, and enable continuous batching for all experiments.

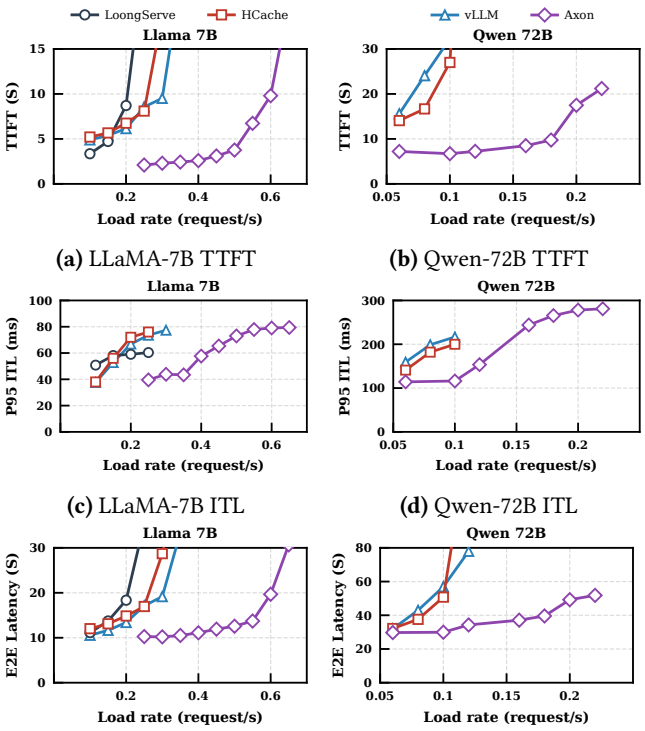
Models. We evaluated LLaMA-2-7B [37] (TP=1), LLaMA-2-13B [37] (TP=2), and Qwen-2-72B [6] (TP=4), with the latter adopts GQA [2], while the others use MHA [38]. We used BF16 for weights and activations. To accommodate long sequences within HBM capacity limits, we extended the context window of LLaMA models to 119K tokens.

Workloads. We extracted two weeks of production traffic from Aone Agent, Alibaba's enterprise coding copilot agent, whose workload characteristics are detailed in § 2.2. More specifically, we sampled 80 sessions from the trace and capped each session at 20 turns, ensuring the workload fits within the cluster capacity. As the production timestamps usually span a long period, we reset the request arrival pattern using a Poisson distribution ($\lambda = 1.0$), following prior work [12, 13, 40], to keep experiment durations tractable.

Baselines. We used the following baselines for comparison:

- **LoongServe [40]**: It accelerates inference of long-sequence requests via elastic scale-out, yet recomputes all KV caches for multi-turn interactions. We found it load-rate sensitive and underperforming on the workload, so we only included it for LLaMA-2-7B.
- **vLLM [19]**: vLLM serves as a de facto standard for LLM inference. It enables intra-instance context reuse via effective prefix caching and recovers missing KV caches through recomputation.
- **HCache [13]**: It strikes a balance between caching and recomputation by selectively storing hidden states in hierarchical storage and restoring KV caches from them with only lightweight recomputation. It functionally subsumes CachedAttention [12], which only prefetches and migrates over hierarchical storage.

Metrics. Following standard LLM serving evaluation practices [1, 20, 41, 45], we measured Time-To-First-Token (TTFT) for the prefill stage and Inter-Token Latency (ITL) for the decode stage, reporting both the average and tail latency. We also reported other system metrics, such as End-to-End (E2E) latency and context splitting ratio, for completeness.



(a) LLaMA-7B TTFT **(b)** Qwen-72B TTFT

(c) LLaMA-7B ITL **(d)** Qwen-72B ITL

(e) LLaMA-7B E2E **(f)** Qwen-72B E2E

Figure 6. Overall performance comparison across LLaMA-7B and Qwen-72B for TTFT, ITL, and end-to-end latency.

6.2 End-to-End Performance

This section presents end-to-end results that demonstrate Axon's effectiveness on real-world agent workloads across different LLM models, as shown in Fig. 6. Due to space constraints, results for the 13B model are omitted from the figure.

TTFT. Axon substantially improves TTFT and throughput over the baselines. In particular, it shifts the TTFT knee point, i.e., the load rate at which TTFT begins to rise sharply, by 1.67×, 2.00×, and 2.25× relative to the best baselines. This indicates higher serving capacity, as Axon's global memory pool enables shared storage, avoids single-instance resource bottlenecks, and better absorbs bursty requests. Within these extended capacity limits, Axon reduces TTFT by up to 4.14× for 7B at a load rate of 0.25, 2.89× for 13B at a load rate of 0.2, and 4.02× for 72B at a load rate of 0.12 compared with the best baseline. These gains come from Axon's ability to efficiently reuse existing KV caches and leverage aggregate PCIe bandwidth across the cluster to accelerate context restoration for cache misses, thereby substantially reducing context preparation latency for new requests.

ITL. Axon reduces tail ITL by leveraging AQCP to distribute attention computation across the cluster. It P95 ITL speedups of up to 1.87× for 7B at a load rate of 0.25, 1.62× for 13B at a load rate of 0.2, and 1.72× for 72B at a load rate of 0.1, respectively. As the load rate increases, more requests contend for cluster-wide compute resources, which reduces

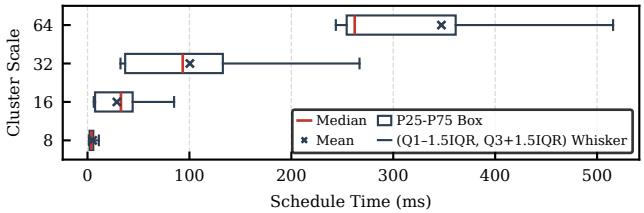
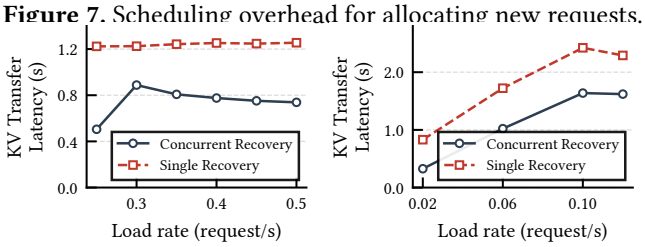


Figure 7. Scheduling overhead for allocating new requests.



(a) Llama 7B model **(b)** Qwen 72B model

Figure 8. KV cache transfer latency.

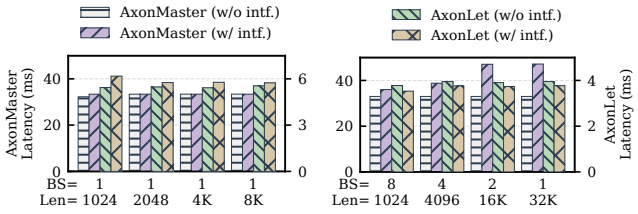
the acceleration benefit of AQCP and causes P95 ITL to converge and eventually plateau. In addition, Axon improves mean ITL by 1.16× for 7B at a load rate of 0.25, 1.34× for 13B at a load rate of 0.2, and 1.23× for 72B at a load rate of 0.1 compared with the best baseline.

E2E latency. Because agent workloads exhibit a short average decode length of 270 tokens, E2E latency is dominated by TTFT. Building on Axon's substantial TTFT improvements, Axon delivers E2E latency speedups of up to 1.65× (7B at load rate 0.25), 1.68× (13B at load rate 0.2), and 2.27× (72B at load rate 0.12), respectively.

6.3 Asynchronous Scheduling Performance

6.3.1 New request allocation overhead. Fig. 7 presents the new-request allocation latency across varying cluster scales. The median scheduling time scales approximately linearly with cluster size, from 4 ms on an 8-instance cluster to 262 ms on a 64-instance cluster, while the interquartile range widens proportionally, indicating increased variance at larger scales. Even at the largest scale, the scheduling overhead remains sub-second and accounts for less than 5% of the typical LLM agent's TTFT, which commonly exceeds 5 seconds and often reaches 10 seconds. This suggests that new-request allocation is not a bottleneck, and we should prioritize prefill computation efficiency, KV cache utilization, and request batching over scheduler's micro-optimizations.

6.3.2 KV cache transfer overhead. Concurrent KV cache recovery via aggregated PCIe bandwidth is a central design advantage of Axon. Fig. 8 compares the KV cache restoration latency of Axon's concurrent recovery against the single-instance recovery used by baselines. By harnessing aggregated PCIe bandwidth across multiple instances, concurrent recovery restores missing KV cache fragments in parallel, substantially reducing restoration time. At low load rates, concurrent recovery achieves speedups of 2.42× for the 7B model (at load rate 0.25) and 2.5× for the 72B model (at load rate 0.02) on average. As the load rate increases, however,



(a) Prefill interference latency **(b)** Decode interference latency

Figure 9. Mutual interference between AxonMaster and AxonLet under varying load levels.

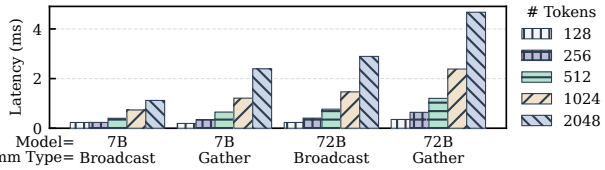


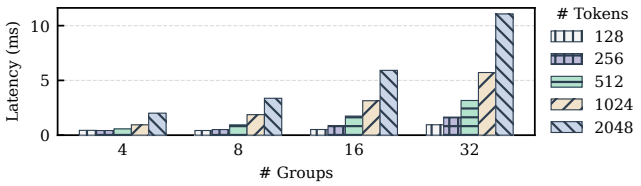
Figure 10. Latency overhead of Axon's customized communication primitives for AQCP.

the cluster's compute and communication overhead rises, forcing KV cache transfers to compete with frequent tensor-parallel and AQCP traffic for PCIe bandwidth. Consequently, the speedup drops to 1.70× at load 0.50 for the 7B model and to 1.41× at load 0.12 for the 72B model.

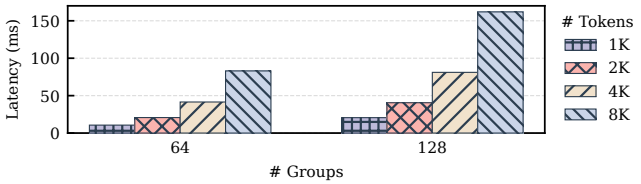
6.4 AQCP Mechanism Performance

6.4.1 Interference between AxonMaster and AxonLet. Co-locating AxonMaster and AxonLet introduces acceptable mutual interference. To evaluate this interference, we ran Llama-7B on AxonMaster with a persistent batch of two decode requests (sequence length 37K each) and measured its ITL, while running on AxonLet a prefill batch (1 request, prompt length 256, sequence length 1024–8K) and a decode batch (1–8 requests, sequence length 1024–32K), and measured the per-iteration latency. Results are shown in Fig. 9. Because AxonMaster is memory-bound (decode) while AxonLet prefill is compute-bound, resource demands are complementary. As shown in Fig. 9a, AxonMaster's ITL barely changes under interference (w/ intf.) versus isolation (w/o intf.), and AxonLet latency degrades by merely 13.5% at BS=1, Len=1024 and only 3%–6% elsewhere, confirming negligible interference and efficient heterogeneous resource utilization. When both sides contend for memory bandwidth during decode, AxonMaster degradation worsens with load, reaching 43.0% at BS=1, Len=32K (33 ms to 47 ms), yet AxonLet latency improves by 4.3%–6.5% (Fig. 9b). We believed that AxonMaster's attention operations prefetch KV cache into L2, warming the access path that AxonLet reuses and giving it a "free ride" on lower latency. Nevertheless, under typical loads (e.g., BS=8, Len=1024 or BS=4, Len=4096), AxonMaster degradation remains modest at 9.1% and 17.5% (to 36 ms and 38.8 ms), which is acceptable.

6.4.2 AQCP communication overhead. The customized AQCP communication primitives are efficient and scale well with context length. Fig. 10 reports their per-layer latency



(a) Typical AQCP scales (#Groups ≤ 32).



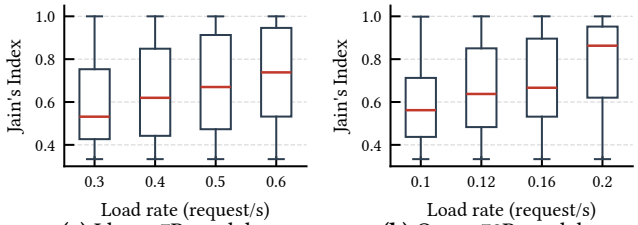
(b) Extreme scales (#Groups ≥ 64).

Figure 11. Latency overhead of the merge primitive under varying context lengths.

for all models. During decoding, each AQCP invocation typically transfers one token per request, for which broadcast or gather takes only 0.05–0.1ms per layer. Since this communication is fully overlapped with attention computation on the `AxonMaster` side, and the ITL of the 7B and 72B models is 30–40ms and 70–80ms, respectively, the communication overhead during decoding is small.

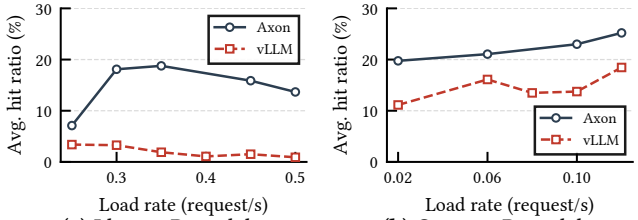
During prefill, the per-iteration communication volume can increase to 1K–2K tokens. With GPUDirect RDMA, broadcasting and gathering 2K tokens take 1.12ms and 2.39ms on 7B, and 2.89ms and 4.67ms on 72B, respectively. Although this overhead is no longer negligible, it remains a small portion of TTFT: Fig. 6 shows TTFT of 3–5s for 7B and about 10 s for 72B. Overall, AQCP communication adds only 5%–10% to prefill latency and does not become a bottleneck.

6.4.3 Merge primitive overhead. The merge primitive introduces negligible latency overhead to the iteration. Fig. 11a quantifies the latency of our merge primitive across all 32 layers in the Llama-7B setup, which is responsible for combining partial attention outputs from dynamic AQCP groups. At typical AQCP scales (#Groups ≤ 8 and #Tokens ≤ 128), the cumulative latency across all 32 layers stays below 1 ms, rendering its impact on end-to-end latency negligible. When #Groups ≥ 16 and #Tokens ≥ 1024, the per-iteration latency rises sharply beyond 3 ms; at #Groups=32 and #Tokens=2048, merge operations alone consume 11.3 ms per iteration. Fig. 11b extends this scaling study to the extreme regime, where the per-iteration cost reaches 10.61, 20.70, 41.50, and 83.01 ms at #Group=64, and 20.56, 40.51, 81.11, and 161.89 ms at #Group=128, as #Token grows from 1K to 8K. However, these extremes occur primarily during the prefill stage, where TTFT already spans several seconds to over ten seconds , so the merge overhead remains a small fraction of the overall latency. Moreover, group counts as high as 64 or 128 almost never appear in production, where the per-request collective width stays in the single digits. Thus, the merge primitive incurs acceptable overhead under normal AQCP parallelism.



(a) Llama 7B model **(b)** Qwen 72B model

Figure 12. Jain's index of context sharding across instances.



(a) Llama 7B model **(b)** Qwen 72B model

Figure 13. Average hit ratio across varying load rates.

6.5 Optimality of Global Scheduling Policy

6.5.1 Distribution of context sharding. As the load rate increases, the scheduler generates increasingly balanced context-sharding plans. We adopted Jain's Index [16] to measure how the scheduling algorithm partitions request contexts. The metric ranges from $\frac{1}{n}$ to 1, where $n = 4$ denotes the total number of instances, and higher values indicate more even partitioning and more balanced load distribution. We computed the index under different models and load levels, with results presented in Fig. 12. At low load rates, the median index falls between 0.4 and 0.6, because low load leads to short local attention latency in AxonMaster. To avoid being blocked by AQCP communication and remote attention computation, the scheduling algorithm tends to concentrate request contexts on a few instances. As the load rate increases, local attention latency grows, providing sufficient computation to overlap AQCP overhead. The scheduling algorithm therefore partitions contexts more evenly across the cluster, with the median index approaching or even exceeding 0.8, significantly improving load balance.

6.5.2 Hit ratio improvement. Fig. 13 plots the KV cache hit ratio of vLLM and Axon across different LLMs as the load rate varies. Overall, Axon achieves a higher hit ratio compared to baselines, indicating that the scheduler effectively exploits cross-instance context reuse. For the 7B (TP=1) and 13B (TP=2) configurations, the per-instance context capacity is severely constrained (approximately 119K and 154K tokens, respectively). Consequently, vLLM—which relies solely on intra-instance resources—suffers frequent cache misses when serving requests whose average context length reaches 37K tokens, rendering its prefix-caching mechanism largely ineffective. By contrast, Axon consolidates distributed memory into a global pool, eliminating the single-instance capacity bottleneck and improving the hit ratio by nearly 20% over vLLM. For the 72B model (TP=4), the larger per-instance

Table 1. Communication volume for serving one request on LLaMA-7B in BF16, with a 33.5K-token context and a 3.7K-token prompt taken from our production trace

Design	Data moved
Full KV aggregation	17.6 GB
AQCP per remote shard	0.9 GB
AQCP with two remote shards	1.8 GB

Table 2. Metadata pull latency under varying instance counts. Each instance pulls its next-iteration metadata every 20 ms to emulate a realistic iteration interval, and the bottom row reports the latency as a share of that interval

Instances	1	128	256	512
Pull latency (ms)	1.034	1.11	1.40	1.44
Share of 20 ms interval	5.2%	5.6%	7.0%	7.2%

capacity allows vLLM to achieve a substantially higher baseline hit ratio; nevertheless, Axon still improves it by 9.5%, demonstrating that its context-locality-aware scheduling policy exploits cluster-wide resources more effectively to increase KV reuse.

6.6 Overhead Analysis

This subsection quantifies the overheads of Axon along the serving path in large-scale settings, covering communication volume, control-plane scheduling, and AQCP collectives. Across all three sources, the overhead remains a small fraction of the latency it helps reduce.

6.6.1 Communication Volume under Long Inputs. AQCP moves substantially less data than designs that aggregate KV state toward computation, such as Mooncake [32] and LMCache [24]. The saving follows from the asymmetric decomposition, under which a remote instance receives query tensors and returns partial attention outputs rather than shipping its cached state. Query tensors scale with the length of the newly arrived prompt, whereas KV state scales with the context accumulated over all previous turns.

We instantiate this comparison on our production trace, where the average context reaches 33.5K tokens and the average new prompt is 3.7K tokens. Table 1 summarizes the resulting communication volume. For LLaMA-7B in BF16, the full context KV amounts to roughly 17.6 GB, while the AQCP traffic is about 0.9 GB per remote shard. With two remote shards, Axon moves about 1.8 GB in total, which is roughly 9.8× less than transferring the full KV, as adopted by Mooncake [32] and LMCache [24]. The gap widens further as sessions accumulate more turns, since the context grows while the prompt length stays roughly constant.

6.6.2 Control-Plane and Scheduling Overhead in Large-scale Clusters. The control-plane overhead of Axon stays small and grows slowly with cluster size. Instances pull their next-iteration metadata from the global scheduler over ZMQ, and this pull is the only per-iteration control-plane operation on the critical path.

Table 2 reports the measured pull latency. A single puller observes 1.034 ms, and the latency rises to 1.11, 1.40, and 1.44 ms at 128, 256, and 512 instances respectively. Even at 512 instances the pull consumes under 8% of a 20 ms iteration interval, so metadata retrieval leaves ample headroom for GPU execution.

The scheduling overhead itself scales with cluster size rather than with the number of in-flight requests. The scheduler makes decisions only for newly arrived requests, and both the placement decision and the metadata assembly run asynchronously with respect to GPU execution. Per-request scheduling cost therefore stays small and off the critical path, which is consistent with the scaling trend reported in § 6.3.1. Taken together, the control plane is unlikely to become a bottleneck at practical cluster scales.

Table 3. Cost breakdown under a prompt-heavy request with a 16K-token prompt over a 78K-token context, which matches the P95 context length of our trace

Metric	Value
Local attention per layer	7.78 ms
Broadcast and gather per layer	6.34 ms
TTFT under HCache	5.9 s
TTFT under Axon	2.5 s
TTFT speedup	2.36×

6.6.3 Collective Latency Measurement for Extra-Long Prompts. The width of the AQCP collectives is bounded by per-request parallelism rather than by cluster size. On each batch, the Dynamic Grouping mechanism of the AxonMaster derives the collective width from the sharding plan of the requests it serves. In our deployment this width rarely exceeds 8 per request or 32 per batch, so adding instances to the cluster does not widen the collectives.

Even at a width of 32, current scale-out fabrics leave ample headroom. A 2K-token iteration incurs 10.8 ms of communication per layer at 400 Gbps interconnects and 5.4 ms per layer at 800 Gbps interconnects. Token counts of this magnitude arise only during prefill, whose TTFT already spans seconds, and the communication is largely hidden behind the local attention of the AxonMaster.

Moreover, prompt-heavy requests preserve this overlap. We evaluate a 16K-token prompt over a 78K-token context, which corresponds to the P95 context length of our trace, and Table 3 breaks down the resulting cost. Local attention takes 7.78 ms per layer and fully hides the 6.34 ms per layer spent on broadcast and gather, while TTFT improves by 2.36× from 5.9 s under HCache to 2.5 s under Axon. Chunked prefill is what keeps this invariant intact, since it caps the per-iteration query length at the token budget and thereby

bounds the AQCP communication latency. AQCP communication is thus overlapped rather than exposed in both the typical and the prompt-heavy regime.

7 Related Work

LLM Serving System. Existing LLM serving systems can be broadly classified into two categories: single-instance and distributed systems. The former[8, 13, 19, 28, 34, 51] optimize inference efficiency within a single instance through sophisticated memory scheduling and compute optimizations. Distributed serving systems[3, 32, 35, 36] provide integrated solutions that combine inference engines with cluster-aware management. Building on these efforts, Axon targets agentic workloads across multiple instances through resource sharing and cooperative execution, substantially improving serving efficiency for agentic LLM applications.

KV Cache Offloading. Many works [9, 11, 12, 24, 42, 47] exploit hierarchical storage to offload KV caches, alleviating the capacity limits of GPU HBM. To further reduce data movement overhead, others [14, 17, 21, 25–27, 50] push computation down to heterogeneous GPUs or even CPUs, leveraging heterogeneous execution to lower GPU HBM pressure and migration costs, thereby improving inference efficiency.

Context Parallelism. Sequence parallelism [15, 18] was originally developed to reduce activation memory in training by sharding tensors along the sequence dimension. In the inference domain, sequence parallelism and its derivative context-parallelism methods [22, 23, 40, 43] have evolved into a foundational mechanism for long-context serving. These approaches leverage context parallelism through sequence scaling or phase disaggregation, yet typically incur non-negligible synchronization overhead during execution.

8 Conclusion

This paper presents Axon, an agent serving system that reduces both prefill and decode latency by reusing local KV caches across distributed instances in a cluster. Its key contributions are asynchronous global scheduling over a unified memory pool and Asymmetric Query Context Parallelism. Asynchronous scheduling ensures that request scheduling does not interrupt ongoing inference, mitigating the bottleneck of a global scheduler. Asymmetric Query Context Parallelism differs fundamentally from existing context and sequence parallelism by enabling fine-grained, module-level parallelism and allowing requests within the same batch to adopt different parallelism plans across instances. Overall, Axon opens a new direction for distributed agentic LLM inference systems by effectively trading off KV-cache reuse with load balancing.

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. 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.

[2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Feder ico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transf ormer Models from Multi-Head Checkpoints. In *The Conference on Empirical Methods in N atural Language Processing (EMNLP)*.

[3] Alibaba. 2025. RTP-LLM: A High-Performance LLM Inference Engine. https://github.com/alibaba/rtp-llm.

[4] Anthropic. 2025. *Claude Code: An Agentic Coding System.* Retrieved April 2026 from https://www.anthropic.com/product/claude-code

[5] Aone. 2026. *Aone Copilot.* https://marketplace.visualstudio.com/items?itemName=Aone.aone-copilot Visual Studio Code Extension.

[6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2024. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671* (2024). https://arxiv.org/abs/2407.10671

[7] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101. doi:10.1147/sj.52.0078

[8] Wenyan Chen, Chengzhi Lu, Huanle Xu, Kejiang Ye, and ChengZhong Xu. 2026. High Throughput and Low Latency LLM Serving via Adaptive KV Caching. In *Proceedings of the 21st European Conference on Computer Systems*. 56–71.

[9] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024). https://arxiv.org/abs/2412.19437

[10] Boxiao Du, Boning Huangfu, Yizhou Luo, Chen Chen, Zijun Li, Minchen Yu, Xiaoyi Fan, and Minyi Guo. 2026. GoodServe: Towards High-Goodput Serving of Agentic LLM Inferences over Heterogeneous Resources. *arXiv preprint arXiv:2605.16867* (2026). https://arxiv.org/abs/2605.16867

[11] Shaoting Feng, Hanchen Li, Kuntai Du, Zhuohan Gu, Yuhan Liu, Jiayi Yao, Siddhant Ray, Samuel Shen, Yihua Cheng, Ganesh Ananthanarayanan, et al. 2025. AdaptCache: KV Cache Native Storage Hierarchy for Low-Delay and High-Quality Language Model Serving. *arXiv preprint arXiv:2509.00105* (2025).

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

[13] Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. Fast State Restoration in LLM Serving with HCache. In *Proc. of the Twentieth European Conference on Computer Systems (EuroSys)*. ACM, 128–143.

[14] Jiaao He and Jidong Zhai. 2024. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines. *arXiv preprint arXiv:2403.11421* (2024).

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Reza Yazdani Aminadabi, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2024. System optimizations for enabling training of extreme long sequence transformer models. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*. 121–130.

[16] Raj Jain, D. M. Chiu, and William R. Hawe. 1998. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *ArXiv* cs.NI/9809099 (1998). https://api.semanticscholar.org/CorpusID:1105820

[17] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2025. Neo: Saving gpu memory crisis with cpu offloading for online llm inference. *Proceedings of Machine Learning and Systems* 7 (2025).

[18] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[19] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 611–626.

[20] Małgorzata Łazuka, Andreea Anghel, and Thomas Parnell. 2024. LLM-Pilot: Characterize and Optimize Performance of your LLM Inference Services. *arXiv e-prints*, Article arXiv:2410.02425 (Oct. 2024), arXiv:2410.02425 pages. arXiv:2410.02425 [cs.DC] doi:10.48550/arXiv.2410.02425

[21] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[22] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. 2024. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669* (2024).

[23] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2024. Ringattention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations*, Vol. 2024. 3992–4008.

[24] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. 2025. Lmcache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665* (2025).

[25] Cheng Luo, Zefan Cai, Hanshi Sun, Jinqi Xiao, Bo Yuan, Wen Xiao, Junjie Hu, Jiawei Zhao, Beidi Chen, and Anima Anandkumar. 2025. Headinfer: Memory-efficient llm inference by head-wise offloading. *arXiv preprint arXiv:2502.12574* (2025).

[26] Zizhao Mo, Junlin Chen, Huanle Xu, and Chengzhong Xu. 2026. Serving Hybrid LLM Loads with SLO Guarantees Using CPU-GPU Attention Piggybacking. *Proceedings of the ACM on Management of Data* 4, 3 (SIGMOD) (2026), 1–26.

[27] Zizhao Mo, Jianxiong Liao, Huanle Xu, Zhi Zhou, and Chengzhong Xu. 2025. Hetis: Serving llms in heterogeneous gpu clusters with fine-grained and dynamic parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1710–1724.

[28] NVIDIA. 2023. TensorRT-LLM: A High-Performance LLM Inference Engine. https://github.com/NVIDIA/TensorRT-LLM Accessed: 2026-06-08.

[29] NVIDIA Corporation. 2025. *NVIDIA Collective Communications Library (NCCL) Developer Guide*. https://docs.nvidia.com/hpc-sdk/archive/25.11/nccl/index.html Version 2.28.7.

[30] OpenClaw Community. 2026. *OpenClaw: A Local-First Personal AI Assistant*. Retrieved April 2026 from https://github.com/openclaw/openclaw

[31] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*. 80–89. doi:10.1109/ICPP.2013.17

[32] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX conference on file and storage technologies (FAST 25)*. 155–170.

[33] Qwen Team. 2025. *Qwen2.5 Technical Report*. Retrieved April 2026 from https://qwenlm.github.io/blog/qwen2.5/

[34] Baodi Shan, Mauricio Araya-Polo, and Barbara Chapman. 2026. GICC: A High-Performance Runtime for GPU-Initiated Communication and Coordination in Modern HPC Systems. *arXiv preprint arXiv:2604.22126* (2026).

[35] Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, Yifei Zhang, et al. 2025. AIBrix: Towards Scalable, Cost-Effective Large Language Model Inference Infrastructure. *arXiv e-prints* (2025), arXiv–2504.

[36] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*. 173–191.

[37] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023). https://arxiv.org/abs/2307.09288

[38] Ashish Vaswani, Noam Shazeer, Niki Parma r, Jakob Uszkoreit, Llion Jones, Aidan N Gom ez, Łukasz Kaiser, and Ill ia Polosukhin. 2017. Attention is all you need.

[39] vLLM Team. 2024. *Automatic Prefix Caching*. vLLM Project. https://docs.vllm.ai/en/latest/features/automatic_prefix_caching.html

[40] 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*. 640–654.

[41] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2026. FastServe: Iteration-Level Preemptive Scheduling for Large Language Model Inference. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 57–74. https://www.usenix.org/conference/nsdi26/presentation/wu-bingyang

[42] Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. 2025. Strata: Hierarchical context caching for long context language model serving. *arXiv preprint arXiv:2508.18572* (2025).

[43] Amy Yang, Jingyi Yang, Aya Ibrahim, Xinfeng Xie, Bangsheng Tang, Grigory Sizov, Jongsoo Park, and Jianyu Huang. 2025. Context parallelism for scalable million-token inference. *Proceedings of Machine Learning and Systems* 7 (2025).

[44] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[45] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. *arXiv e-prints*, Article arXiv:2501.01005 (Jan. 2025), arXiv:2501.01005 pages. arXiv:2501.01005 [cs.DC] doi:10.48550/arXiv.2501.01005

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*. 521–538.

[47] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*. 144–158.

[48] Yichao Yuan, Mosharaf Chowdhury, and Nishil Talati. 2026. KAIROS: Stateful, Context-Aware Power-Efficient Agentic Inference Serving. *arXiv preprint arXiv:2604.16682* (2026). https://arxiv.org/abs/2604.16682

[49] ZeroMQ Community. 2026. *ZeroMQ: An Open-Source Universal Messaging Library*. ZeroMQ Project. https://zeromq.org/

[50] Xuanlei Zhao, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. 2024. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. *Proceedings of Machine Learning and Systems* 6 (2024), 162–172.

[51] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.