LMCACHE: AN EFFICIENT KV CACHE LAYER FOR ENTERPRISE-SCALE LLM INFERENCE

Yihua Cheng [1 *] **Yuhan Liu** [1 2 *] **Jiayi Yao** [1 2 *]
Yuwei An [1] **Xiaokun Chen** [1] **Shaoting Feng** [1] **Yuyang Huang** [1 2] **Samuel Shen** [1]
Kuntai Du [1] **Junchen Jiang** [1 2]

ABSTRACT

Today's LLM inference systems treat individual engines and queries independently for simplicity, but this causes significant resource inefficiencies. While there are proposals to avoid redundant computation by reusing KV caches across queries and to increase GPU utilization by disaggregating a single query to different engines, their promises cannot be realized without efficiently offloading and communicating KV cache across LLM inference engines and queries. We present **LMCACHE**, the first and so far the most efficient open-source KV caching solution, which extracts and stores KV caches generated by modern LLM engines (vLLM and SGLang) and shares the KV caches across engines and queries. LMCACHE exposes KV caches in the LLM engine interface, effectively transforming LLM engines from individual token processors to a collection of engines with KV cache as the storage and communication medium. In particular, it supports both *cache offloading* (prefix reuse across queries) and *prefill–decode (PD) disaggregation* (cross-engine cache transfer). LMCACHE's high performance and wide adoption stem from the following contributions: *(i)* highly optimized KV cache data movement with performance optimizations including batched data movement operations, compute and I/O pipelining; *(ii)* a modular KV cache connector component, decoupling LMCACHE from the rapid evolution of inference engines; *(iii)* a first-class control API, such as pinning, lookup, cleanup, movement, and compression, for flexible cache orchestration across GPU, CPU, storage, and network layers. Evaluation shows that combining LMCACHE with vLLM achieves up to 15× improvement in throughput across workloads such as multi-round question answering and document analysis. With a growing community, LMCACHE has seen dramatic growth in adoption by enterprise inference systems, which provides valuable lessons for future KV caching solutions. The source code of LMCACHE is at: https://github.com/LMCache/LMCache.

1 INTRODUCTION

Today, large-language model (LLM) *inference* has outpaced training in growth. LLM inference powers a wide range of applications, from interactive customer support and code generation to retrieval-based document analysis and agentic workflows. However, the LLM inference systems have not caught up—latency (both response latency and generation speed) and cost are now the bottleneck.

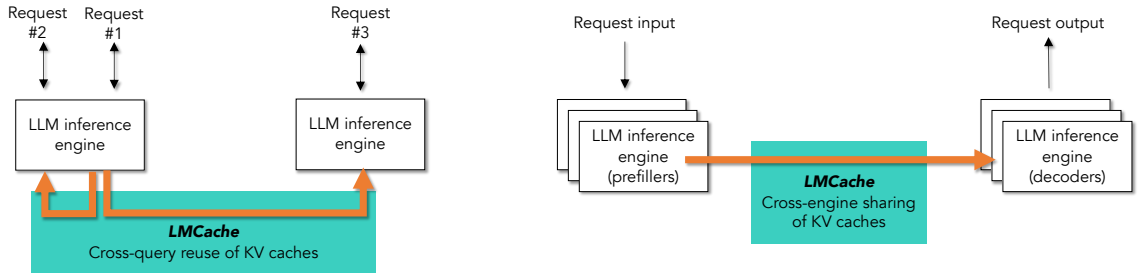
A key limitation of today's LLM inference systems is that each user query is *independently* processed by *one* instance of the inference engine, *without* any data being reused across queries or inference engines. This means that a LLM query's lifecycle, including computation and I/O operations, happens in the GPUs and the GPU memory of one inference engine. Each query takes a token sequence as input and another token sequence as output, and once a query completes, all outputs or intermediate states generated during inference are discarded.

This design, albeit simple, leads to redundant computation and resource underutilization, and two notable industry proposals attempted to address them (Figure 1).

Cross-query caching to avoid redundant compute: As each query is treated in isolation, the LLM must prefill the same context (prefix) each time the context is used, and the repeated prefill compute is expensive and causes slow response, as users cannot see the first token before the prefill completes. Yet, such *redundant computation* could be avoided if we persist *KV cache* of the prefix—which LLM created internally after it prefills the prefix—beyond the lifecycle of a query and reuse the KV cache when subsequent queries reuse the prefix (*e.g.,* the same prompt template or retrieved document chunk). By avoiding redundant computation, cross-query context (prefix) caching

[1]TensorMesh Inc., Foster City, CA, USA [2]University of Chicago, Chicago, IL, USA. Correspondence to: Junchen Jiang <junchenj@uchicago.edu>.

[*]Equal contribution



(a) Context caching: Cross-query reusing via CPU/disk offloading of KV cache

(b) PD disaggregation: Cross-engine sharing of KV caches

Figure 1. LMCACHE *supports both context caching (KV cache offloading and sharing across queries) and PD disaggregation (cross-engine transfer of KV caches.*

significantly lowers both time-to-first-token (TTFT) and overall GPU resource consumption during the prefill phase.

Prefill–decode disaggregation for higher utilization: Co-locating the prefill phase (compute-bound, throughput-oriented) and the decode phase (memory-bound, latency-sensitive) in the same inference engine causes *resource underutilization* as the GPUs must be overprovisioned to ensure a consistently low decoding latency without being interrupted by prefill. In response, the industry attempted to decouple the throughput-oriented prefill processing from latency-sensitive decoding Such prefill-decode (PD) diaggregation architecture batches all queries' prefill phase on one set of inference nodes and transfers their KV caches to another set of inference nodes that run the decoding phase only, thus mitigating tail decoding latency under high concurrency.

To make these ideas practical, the LLM inference systems must be augmented with new KV cache semantics. In particular, inference engines should support the new interface that *extracts* KV caches from a normal inference call and *reloads* KV caches into subsequent queries on demand. The system also must allow the extracted KV caches to be *stored* persistently and *transferred* across distributed inference engines. Most importantly, for such interface extensions to be practical, KV cache extraction, re-loading, storage, and transfer must be efficient, and the new interfaces must remain compatible with rapidly evolving inference engines such as vLLM (Kwon et al., 2023b) and SGLang (Zheng et al., 2024).

We introduce **LMCACHE**, the first open-source library that provides a high-performance implementation of these new KV cache semantics. With LMCACHE, KV cache can be extracted from and loaded back to inference engines efficiently, stored in a hierarchy of storage devices (CPU memory, local disk, remote disk, and Redis), and transferred over different networks (Ethernet, RDMA, NVLink).

LMCACHE makes three distinct contributions.

#1. Highly optimized performance: LMCACHE incorporates a series of performance optimizations that make storing and loading KV cache efficient and practical in real deployments. For instance, LMCACHE batches operations to pipeline the storing and loading of KV cache, as well as to pipeline GPU compute and data loading/storing (*e.g.,* loading next layer's KV cache while performing computations for the current layer). Moreover, rather than storing/loading KV cache at the granularity of the inference engine's native small page size, LMCACHE stores/loads KV cache at a configurable chunk size, often much larger than page size, to fully utilize the bandwidth between storage devices and GPU memory. LMCACHE also minimizes the copies of KV cache data when moving them among different storage tiers, by implementing zero-copy operations.

#2. Standardized interface with inference engines: LMCACHE defines standardized connector interfaces that remain compatible with fast-changing inference engine backends. On average, 15–20 new open-weight models are released every week in 2025, so to best utilize new hardware for the new models, modern LLM inference engines must evolve rapidly, potentially changing the KV cache layout in GPU memory and thus affecting the LMCACHE interface. To address this, LMCACHE designs and implements a modular KV cache connector interface that decouples LMCACHE with the inference engine backend, so LMCACHE can easily adapt to the evolving APIs in the inference engines.

#3. Flexible KV cache management interface: The interface augmentation introduced by LMCACHE exposes KV cache, a new data structure in LLM inference. LMCACHE exposes APIs that allow developers and operators to locate, move, pin, and even compress KV cache extracted from inference engines. These first-class APIs allow higher-level applications, such as query schedulers or routers, to make better decisions, such as KV cache-aware query routing.

Our evaluation demonstrates that LMCACHE consistently outperforms both built-in KV caching mechanisms in open-source inference frameworks and commercial inference APIs, delivering up to 15× higher throughput and at least 2× lower latency across diverse settings, including local prefix caching, distributed prefix reuse, and PD disaggregation. Beyond quantitative gains, LMCACHE has seen adoption across a number of enterprises and open-source projects, informing practical lessons in KV cache–driven optimization at production scale.

The remainder of this paper details the motivation (§2), LMCACHE architecture and key design choices (§3–§6), experimental evaluation (§8), and deployment experiences (§9).

2 MOTIVATION

2.1 Inference Cost and Delay as the Bottleneck

The past two years have witnessed explosive growth in the deployment of LLMs across diverse enterprise applications. As the user base and query volume of LLM-based applications increase, the cost and delay of *inference*, rather than training, emerges as the primary bottleneck to scaling LLM serving to production scale.

Inference cost: Besides the simple fact that LLMs are queried by more users, the increasing input length and output length also contribute to raising inference cost, as each input token and each output token incur an additional cost to process. Four trends are behind the increase in input and output lengths. *(i)* More interactions by each user with LLMs build a longer user history, which will be prepended to future LLM inputs as the context. *(ii)* LLMs increasingly take multimodal inputs, such as images and videos, which typically will be transformed to a long sequence of tokens before being fed to LLMs. *(iii)* Newer LLMs often have longer context windows, allowing context/prompt engineers to simply stuff more tokens into LLM inputs. *(iv)* Finally, reasoning models produce unusually long output, which might be used as input to subsequent queries in one workflow.

Even with steady improvement on GPU price and performance, serving a trillion-token-scale workload can incur millions of dollars in annual compute expenditure (Nie et al., 2024; Databricks Research, 2024). Redundant computation across queries (*e.g.,* recomputing the same prompt prefix hundreds of times) exacerbates these costs (Zhang et al., 2024; Caylent, 2024).

Inference delay: In interactive applications, tail latency (*e.g.,* 95th/99th percentile) is as critical as median latency. Slow responses in the worst case, often due to resource contention or large context processing, directly degrade user experience and can lead to user abandonment, in much the same way video streaming stalls affect viewer experience (Liu et al., 2024a).

In particular, empirical observations from industry reports and academic studies indicate that:

- Time-to-first-token (TTFT) can be dominated by the prefill phase for long-context inputs, especially when context lengths exceed 8–16k tokens.
- Inter-token latency (ITL) remains low for single-stream decoders, but rises sharply when GPU compute is shared across many concurrent sessions.
- Tail delays are disproportionately impacted by scenarios requiring large prompt recomputation or reallocation of memory-bound resources.

2.2 KV Caching as a Cross-Cutting Optimization

KV cache was originally introduced to accelerate a single inference query by storing the attention states, in the form of K and V tensors, for input tokens and previously generated tokens directly in GPU memory. KV cache effectively stores the attention information between each pair of tokens that have been seen so far in this query. In short, it is a *LLM-native* representation of knowledge.

In all popular transformer architectures, these caches are stored and used only within the lifecycle of an individual query and discarded when the query ends. Yet, KV caching can be extended to support two critical performance optimizations in production-scale serving:

1. *Context caching (i.e.,*cross-query *KV cache reuse)* persists KV cache segments from one query and reusing them for subsequent queries that share a common prefix. Examples include document analysis where the same document (chunk) remains constant across multiple queries, and multi-turn dialogues with a fixed system prompt or long preamble. Prefix caching reduces redundant computation during the prefill phase, directly lowering TTFT and GPU-hours per query (Liu et al., 2024b; Gao et al., 2024; Jin et al., 2025b; Ren et al., 2025; Qin et al., 2025a; Jin et al., 2024; Chen et al., 2024; 2025).
2. *Prefill–decode (PD) disaggregation (i.e.,*cross-engine *KV cache transfer)* splits inference into a *prefill* stage (processing the entire input prompt) and a *decode* stage (autoregrssive token generation) across different GPUs or nodes. This approach reduces tail latency by maximizing the decoding speed without being interrupted by the prefill phase (Zhong et al., 2024; Patel et al., 2024; Shi et al., 2025).

Roles of KV caching: Both optimizations critically depend

Message Size	Transfer Throughput
64KB	4GBps
256KB	13GBps
1MB	30GBps
10MB	46GBps
16MB	49GBps
100MB	49GBps

Table 1. Transfer message size vs achieved transfer throughput using RCCL transfer library (UCCL Team, 2025).

on the ability to *move KV cache segments efficiently* to and from GPU memory, CPU DRAM, NVMe storage, or over the network.

- Context reusing stores the KV cache of a context for as long as the context will be reused, so it requires a hierarchical storage of KV caches and efficient sharing of KV caches across multiple serving engine instances.
- PD disaggregation requires fast KV cache transfer between GPUs, often across PCIe, NVLink, or RDMA, in order to minimize the end-to-end delay of each query, which runs across the prefiller nodes and the decoder nodes.

2.3 Challenges of Efficient KV Caching

Despite their potential, the practical adoption of prefix caching and PD disaggregation is limited by three interrelated systems challenges:

2.3.1 Challenge #1: I/O inefficiency under paged memory

KV cache storage and transfer used to rely on PyTorch serialization (`torch.save` / `torch.load`) or primitive tensor copying, with a typical transfer speed of only sub-1GB/s. These methods introduce non-trivial delay overhead, especially when handling large data structures like KV caches, and lack zero-copy support with various storage devices (local or remote), causing extra CPU-GPU data copies.

Recent high-throughput inference engines, such as vLLM (Kwon et al., 2023b) and SGLang (Zheng et al., 2024), make KV cache storage and transfer even more challenging. They employ *paged* attention memory, dividing the attention buffer into small, fixed-size pages (typically 16–64 KB). For instance, vLLM uses 62.5-KB page in Llama-3.1-8B-Instruct model. The paged memory architecture is widely used because it improves batching and memory utilization.

However, because the pages of a KV cache are not always contiguous, the paged memory architecture dramatically increases the number of small-sized I/O operations required to persist or transfer a KV cache. Transferring such small chunks of data is known to suffer from network bandwidth *under*utilization and reduce throughput (Kwon et al., 2025; NVIDIA Developer Forums, 2020; Meta Engineering, 2024). Prior work (Table 1) has shown that, on a setup with two AMD GPU nodes connected by eight Broadcom Thor-2 400Gbps NICs, the transfer size must reach at least 16 MB to saturate the available network bandwidth (Zhou et al., 2025). Furthermore, prior work has shown that only transferring a data size of megabyte range (*e.g.,* 1–2MB) can achieve 75–80% of the theoretical PCIe 5.0 bandwidth (Xie et al., 2025).

2.3.2 Challenge #2: Compatible with fast-evolving inference engines

With the widespread use of AI, new LLMs and hardware accelerators are introduced at a rapid pace. In 2025, one prominent LLM was released on average every 4 days (bes, 2025). In response, inference engines must evolve just as quickly.

Each update to accommodate new models or hardware often changes GPU memory allocation, which in turn changes the KV cache interface. For example, when vLLM adopts a new attention kernel that produces KV caches with different dimensions, the KV caching library must be updated to translate the new kernel's output KV cache format into one compatible with the KV cache library. Keeping up with these frequent changes requires tremendous effort, given the fast-moving inference engines.

2.3.3 Challenge #3: Lack of management APIs

As KV caching becomes a first-class citizen in the LLM inference backend, various components (in addition to the LLM inference engines), as well as ML ops teams, will need to make decisions in a KV-cache-aware manner. Yet, without a unified management interface to locate, evict, pin, or compress caches, these upper-layer modules cannot make informed placement or eviction decisions. This leads to inefficient cache utilization, duplicated storage, and unpredictable eviction policies. For instance, inference query routers, which assign each query to one of the inference engine instances, need to know the locations of KV caches, in order to route queries to instances that already hold the KV cache for matched prefix tokens locally (*e.g.,* in CPU memory).

Moreover, applications now also demand such KV-cache management interfaces. In early 2025, for instance, a financial company[1] that has worked closely with LMCACHE in the production setting asked for an interface that allows users to *explicitly* pin frequently accessed financial docu-

[1]For confidentiality, we do not disclose names of enterprise users in this report.

ments in the KV caching system, for more efficient access to popular contexts. As another example, an agent company requested a series of APIs that allow them to identify the KV cache of a given content, compress the KV cache, and transfer the compressed KV cache across nodes.

2.4 Related Work and Existing Solutions

Several KV cache handling mechanisms exist, but none of them fully address the above challenges:

Inference frameworks: Since the release of vLLM Production Stack (vLLM project, 2025) in January 2025, there have been several open-source distributed inference stacks, including Nvidia's Dynamo (NVIDIA Corporation, 2025), AIBrix (Team et al., 2025), `llm-d` (llm-d Project, 2025), SGLang OME (Team, 2025), and KServe (Contributors, 2025b). They focus on easy deployment of inference engine solutions over Kubernetes, and technically, they all support various query routers based on load or prefix cache awareness and support KV caching, where LMCACHE is used in vLLM production stack, Dynamo, llm-d, and KServe.

Inference engine-native KV caching: Open-source inference engines, like vLLM and SGLang, also offer native GPU-to-CPU KV cache transfers, but it is designed for single-node inference, so they lack cross-node transfer optimization or hierarchical storage support for KV cache. We will evaluate their performance and compare it against LMCACHE in §7.

KV cache storage layers: Mooncake (Qin et al., 2025b), Redis (Redis, 2025), InfiniStore (ByteDance, 2025), and 3FS (Contributors, 2025a) provide distributed object storage or caching, but they lack an efficient "glue" layer between the inference engines to efficiently move small tensors frequently across different storage tiers, or are tightly coupled with a specific inference framework.

Proprietary implementations: Proprietary inference APIs (e.g., Fireworks AI, Together AI) implement their own prefix caching internally, but these are tied to their *closed-source* serving stacks and are not accessible to operators deploying their own infrastructure.

Source code for research: Several research proposals have open-sourced prototypes for their KV cache optimizations, including prefix caching (Kwon et al., 2023a; Zheng et al., 2024; Yu et al., 2025; Gim et al., 2024; Ye et al., 2024; Lee et al., 2024; Zhao et al., 2024; Jin et al., 2024; Gao et al., 2024; Chen et al., 2025; Jin et al., 2025a; Yang et al., 2025), PD disaggregation (Zhong et al., 2024; Patel et al., 2024; Shi et al., 2025), and KV cache compression (Liu et al., 2024c; Jegou et al., 2024; Xiao et al., 2024a;b; Li et al., 2024; Qin et al., 2025c; Tang et al., 2024; Ge et al., 2024; Li et al., 2025; Du et al., 2025; Zhang et al., 2025). However, these prototypes are typically built on research-oriented inference framworks, such as HuggingFace Transformers, not fully enterprise-ready, or are not designed to evolve alongside the rapidly changing inference engine ecosystem, such as SGLang and vLLM.

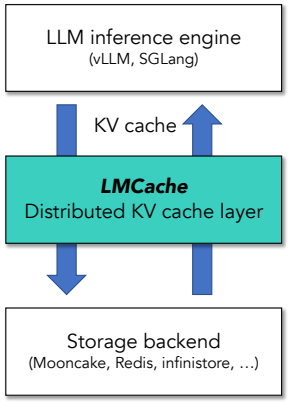


Figure 2. LMCACHE *sits between LLM inference engines and heterogeneous storage/network devices.*

3 OVERVIEW OF LMCACHE

LMCACHE addresses these challenges by a unified, high-performance KV caching layer capable of efficient storage, movement, and explicit management of KV caches for paged-memory inference engines, making prefix caching and PD disaggregation practical at enterprise scale.

As a KV caching layer, LMCACHE sits between LLM inference engines and heterogeneous storage/network devices (Figure 2). Its goal is to provide a standardized, high-performance substrate for KV cache movement and management, while remaining compatible with rapidly evolving inference frameworks such as vLLM and SGLang.

3.1 Architecture

At a high level, LMCACHE connects to inference engines to move KV cache between GPU memory and the storage backend. Figure 3 shows the end-to-end system. Below, we walk through two example workflows: storing and retrieving KV cache.

Store: When a new query arrives, it first passes through the *KV connector*, which prepares metadata such as the tokenized input prompt and GPU memory addresses of the relevant pages. The query then goes to the *token processor*, which determines how many new tokens are not yet in the backend and need to be stored. Finally, the storage manager saves the KV cache for these new tokens to the backend via the *transfer channel*, which handles the data transfer logic.

Retrieve: When a query requires loading KV cache from the backend, it also starts with the KV connector to prepare metadata. The token processor identifies the number of prefix-matched tokens already in the backend. Next, the event manager checks if the same query ID has been seen before. If so, the cached memory addresses are already tracked and can be returned directly to the *GPU connector*, which loads the KV cache back into GPU memory. The event manager also launches asynchronous, layer-wise loading events as described in §4.2. If the query ID is new, it is forwarded to the storage manager to look up the CPU memory addresses of the stored KV cache.

Lookup: When a query needs to check whether the KV cache for specific tokens exists in the backend, higher-level components such as routers query the cache controller. The cache controller maintains a token pool that records all tokens currently stored in the KV cache backend. Whenever a LMCACHE instance stores or evicts a KV cache, the LMCACHE worker inside the instance updates the token pool with the new status. This ensures the token pool always has the up-to-date information of tokens in the backend.

3.2 Key Components

LMCACHE is composed of a high-performance data plane and a flexible control plane.

LMCACHE Worker (Data Plane).: Each inference engine is paired with a LMCACHE worker. The LMCACHE worker module is responsible for moving KV caches between GPU memory and other tiers or other workers. It implements both KV offloading (to CPU or disk) and PD disaggregation (GPU–GPU transfer). To maximize throughput, workers employ kernel-optimized GPU buffers, asynchronous chunked I/O, and layer-wise pipelining, as elaborated in the next section. These techniques allow LMCACHE to sustain near-GPU-resident bandwidth even when handling small paged memory objects (16–64KB), a common configuration in vLLM and SGLang.

LMCACHE Controller (Control Plane).: The controller exposes a programmable API to system operators and higher-level schedulers. It provides cache lifecycle management primitives, such as pinning KV segments, compressing and decompressing cache tensors, migrating caches between devices, and evicting low-priority entries. The controller also maintains a virtualized namespace, enabling uniform addressing of caches across heterogeneous devices.

Together, the LMCACHE worker and controller form a modular system that integrates seamlessly into enterprise-scale serving pipelines. The worker ensures fast data movement, while the controller provides higher-level control semantics necessary for efficient resource utilization.

4 PERFORMANCE OPTIMIZATIONS

An important aspect of LMCACHE is improving the efficiency of KV cache movement across devices. In enterprise-scale LLM inference, LMCACHE addresses three key challenges:

- modern LLM inference engines manage KV cache at the granularity of pages[2], which are typically 20 KB–63 KB for popular models including Llama, Qwen, GPT-OSS etc. Such small units are inefficient for transferring, as they cannot saturate bandwidth (Xie et al., 2025; Zhou et al., 2025);
- KV cache transfers often need to run concurrently with LLM inference. This introduces overhead from two sources. First, data movement can stall inference if transfers are executed in the same CUDA stream as computation. Second, launching memory-copy CUDA functions incurs CPU overhead, as each call consumes CPU cycles and the consumption can be substantial when there are many layers and pages.
- During LLM inference, large volumes of queries generate significant amount of KV caches. Duplicating them on any storage device wastes space and introduces copy overhead, which slows down inference.

Each of these challenges arose from hard lessons in both open-source and enterprise deployments. This section describes these challenges in detail and motivates LMCACHE's design decisions.

4.1 Batched Operations

To address the I/O inefficiency caused by small KV cache units, LMCACHE introduces a set of optimizations.

Configurable Chunk Size: Rather than transferring KV cache at the page level, LMCACHE groups multiple pages from multiple layers into larger chunks, with a default size of 256 tokens per chunk[3]. This is achieved using an intermediate *GPU buffer*. For storing, multiple pages (16 by default) are first copied from GPU memory into the buffer, then offloaded collectively to lower-tier storage (e.g., CPU memory) and saved at the granularity of chunks rather than individual pages. For loading, chunks are first retrieved from the storage layer into the GPU buffer and subsequently split into pages before being placed in the serving engine's internal GPU memory. LMCACHE implements a customized CUDA kernel to accelerate the memory copy.

Batched Store/load Operations: LMCACHE supports storing and loading KV cache across multiple storage devices, including local CPU memory, local disks, and remote

[2]Each page is 16 tokens for a single layer in vLLM.

[3]The chunk size is configurable to different I/O speeds.

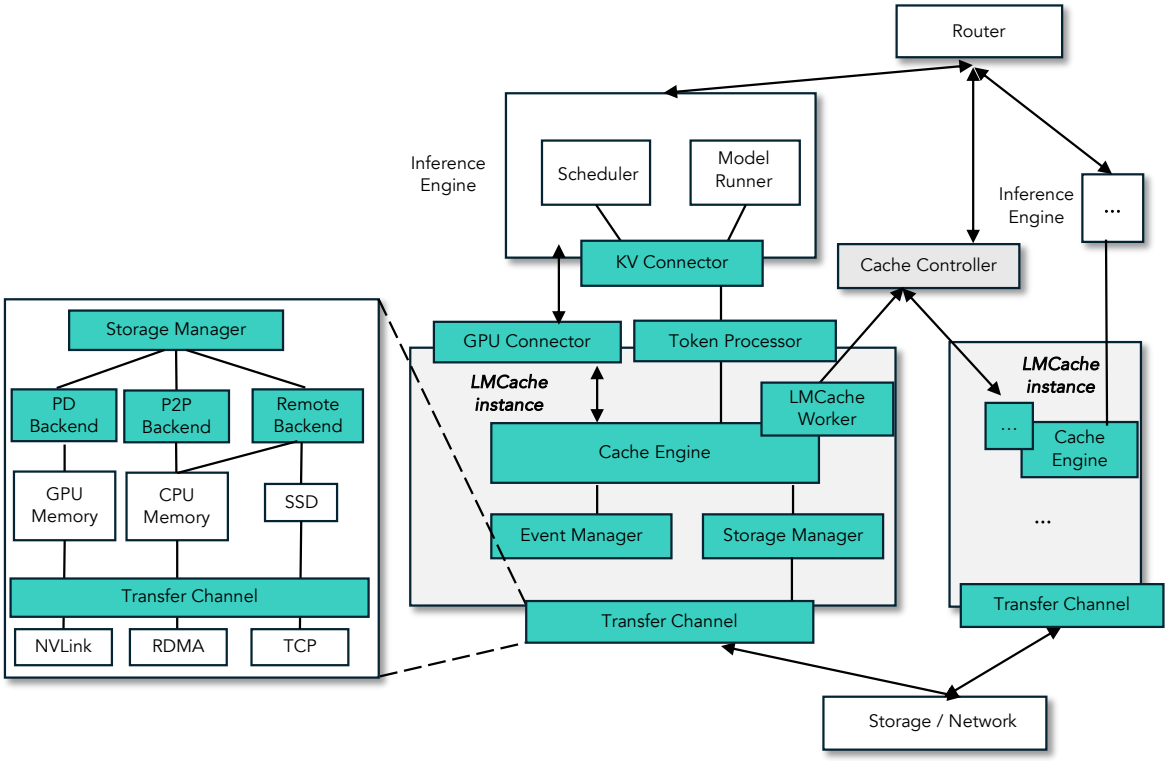


Figure 3. End-to-end system workflow for LMCACHE.

disks. In practice, KV caches often need to be transferred to different devices in parallel – for example, storing the KV cache of a hot context in CPU memory while simultaneously offloading a cold context to local disk. A naive approach that stores the two KV caches sequentially underutilizes available bandwidth, since transferring to CPU memory leaves the disk bandwidth idle. To address this, LMCACHE batches storage and loading queries across different tiers, thereby aggregating operations and fully utilizing the bandwidth between the GPU and multiple storage tiers.

Delayed Decode KV Cache Storing: LMCACHE also supports saving newly generated KV cache during decoding. In a naive design, each newly generated page is immediately offloaded to lower-tier storage, incurring frequent small writes. Instead, LMCACHE aggregates multiple pages and stores them together as a chunk, improving I/O efficiency.

4.2 Compute-I/O Overlapping

LMCACHE's data movement often happens in parallel to LLM inference computation. For example, during decoding, a KV cache is computed while being stored to storage devices at the same time. Naive data movement operations can block the LLM inference computations, or vice versa. LMCACHE employs multiple optimizations aiming for overlapping LLM inference computations with I/O.

Layer-wise pipelining: LMCACHE overlaps KV cache transfers with inference computation through layer-wise pipelining. Specifically, it assigns separate CUDA streams for inference computation and data movement within each layer. For example, before performing inference on the first layer, its KV cache is loaded into the GPU buffer and transformed into pages. While the first layer is running inference, the KV cache for the second layer is asynchronously fetched into the buffer and similarly transformed. Note that the second layer's KV cache loading is happened after the first layer's KV cache is put into the right paged memory. This design ensures that only a fixed-size GPU buffer—whose size is a single layer's KV cache—is required, while enabling overlapping between data transfer and computation.

Asynchronous compute & prefetch: In many scenarios, there is a time gap between when the inference scheduler admits a query and when the query's KV cache is actually needed for inference. For example, if 100 queries arrive simultaneously but the inference engine can only process 50 in parallel, the remaining 50 must wait in the queue before their inference begins. LMCACHE exploits this idle interval to prefetch the queued queries' KV cache from slower storage tiers into faster ones (*e.g.,* from remote disk to local CPU memory). As a result, when the actual inference computation starts, the required KV cache can be loaded directly from faster-tier storage, significantly reducing loading de-

lay. LMCACHE allows users to configure the target tier for prefetching based on their own needs in latency SLO and resource constraints.

Process separation: Co-locating LMCACHE 's data movement with the inference engine in a single process introduces 5%–10% latency overhead due to CPU resource contention. To eliminate this overhead, LMCACHE decouples data movement into a separate process, isolating it from the inference computation process. Additionally, process separation also enables memory sharing across multiple inference engine instances. When data movement is co-located with computation, each inference instance maintains its own CPU memory pool, so instances cannot access one another's KV cache without inter-process communications. By contrast, a standalone data movement process manages a unified CPU memory pool, allowing multiple inference instances to store and load KV cache from the same pool, thereby improving efficiency and reducing redundancy.

4.3 Minimum Data Copy

A naive implementation of KV cache movement would create additional copies of data at each transfer step, especially when dealing with heterogeneous storage types, leading to redundant memory usage and unnecessary overhead. LMCACHE avoids this by maintaining only the minimum required copies.

Zero-Copy Operations: When transferring KV cache to remote storage or another GPU instance (*e.g.,* in disaggregated prefill), LMCACHE minimizes data duplication through a reference counter. Specifically, when KV cache is written to multiple destinations—such as local CPU memory, local disk, or remote disk—LMCACHE increments a reference counter for the shared data instead of creating new copies. Each completed write decrements the counter, and once the count reaches zero, the data is released. This design ensures that data is shared across concurrent write operations without unnecessary replication, thus reducing memory pressure and improving efficiency. This technique is simlar to PCB counter in operating systems (Strecker, 1978).

Dynamic Offloading: Modern inference engines such as vLLM maintain a pool of *free pages* in GPU memory, *i.e.,*, pages whose KV cache is not currently used by active queries. Instead of duplicating all free pages to CPU memory, LMCACHE duplicates only a subset. This mechanism is implemented using three pointers:

- **Start pointer**: the start address of the free-page region in GPU memory.
- **Current pointer**: the index of the free pages that have already been offloaded to CPU memory.

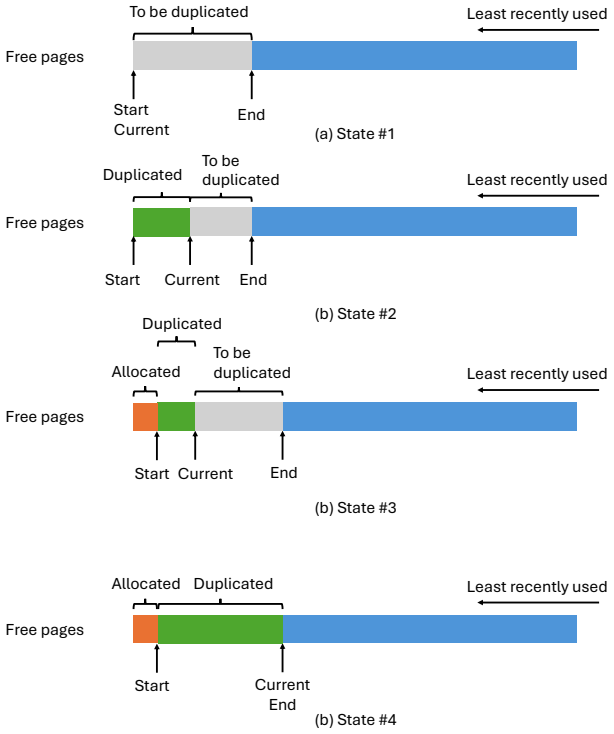


Figure 4. Illustration of dynamic offloading in LMCACHE.

- **End pointer**: the end address of the free pages that are scheduled to be offloaded.

As illustrated in Figure 4, dynamic offloading has four possible states:

- **State #1 (Initialization):** the start and current pointers overlap. The region between the start/current pointers and the end pointer marks the pages pending duplication.
- **State #2 (In-progress):** the current pointer moves toward the end pointer. Pages between the start and current pointers have already been offloaded to CPU memory.
- **State #3 (Query Arrival):** when new queries acquire some of the free pages, the end pointer is moved forward by the number of allocated pages. This ensures sufficient GPU memory is available for future active queries that need to acquire free pages.
- **State #4 (Steady state):** the current pointer overlaps with the end pointer, indicating that all scheduled pages have been duplicated.

Note that if a query attempts to allocate pages beyond the current pointer, the allocation must stall until the current pointer moves right enough to cover the required pages. Thus, a key trade-off in this design is that: the number of duplicated pages—*i.e.,*, the region defined by `end pointer { start pointer`—between GPU and CPU memory. A smaller duplication window reduces the duplication ratio

Function name	Description
get_num_new_matched_tokens(query) → matched_tokens	Returns the number of cache-hit tokens found in LMCACHE 's backend.
update_state_after_alloc(query, blocks, num_external_blocks)	Updates whether a query needs to transfer KV cache from LMCache's backend.
build_connector_meta(scheduler_output) → kv_connector_metadata	Builds metadata for KV cache transfers between LMCache's backend and GPU memory, including GPU memory addresses for KV cache pages.
start_load_kv(kv_pointers)	Starts loading KV cache from lower-tier storage into GPU memory before LLM inference begins.
wait_load_kv(kv_pointers, layer_id)	Synchronizes on KV cache loading to ensure data is available when computation requires it.
start_store_kv(kv_pointer)	Starts offloading KV cache to lower-tier storage after computation.
wait_store_kv(kv_pointer, layer_id)	Synchronizes on KV cache storing to ensure the KV cache for the current layer is offloaded.

Table 2. Functions in LMCACHE's connector.

but increases the likelihood of allocation stalls. For instance, if only one page is duplicated and an inference query requires three pages, the query must wait until the current and end pointers advance by two additional pages. On the other hand, if three pages are duplicated, the same query can proceed immediately without stalling, though at the cost of higher duplication ratio.

5 STANDARDIZED INTERFACE FOR CONNECTING THE KV CACHING LAYER AND INFERENCE ENGINE

Modern LLM inference engines, such as vLLM and SGLang, evolve rapidly to support newly released models with diverse architectures. For example, in 2025, an average of 15–20 new models are released each week. Supporting these new architectures often requires non-trivial modifications to inference engines, such as adding support for Sliding Window Attention or Multi-Head Latent Attention. These code changes frequently alter how KV cache is managed internally, making it infeasible for LMCACHE to adapt in an ad-hoc manner.

To address this challenge, LMCACHE introduces a standardized *KV cache connector* interface that decouples KV cache management from the inference engine backend. This design ensures that LMCACHE remains compatible regardless of how the underlying inference engine evolves.

Concretely, in vLLM, LMCACHE integrates through two key interfaces: 1) the *scheduler*, where the number of prefill tokens directly influences scheduling decisions and are changed by LMCACHE (*i.e.,* if there is cache hit in LMCACHE, the number of tokens that need to be newly prefilled changes); and 2) the *model runner*, where inference computation occurs, and where KV cache loading and storing must be performed before and after execution.

The remainder of this section lists all the interfaces in Table 2, discusses the design for important APIs, and then traces how a query interacts with these interfaces end-to-end.

The interfaces listed in Table 2 form the foundation of LMCACHE 's KV cache loading and storage across lower-tier storage. Among them, the first three interfaces are implemented within the vLLM scheduler, where they prepare the necessary metadata based on the number of matched tokens found in LMCACHE 's KV cache backend. The remaining four interfaces reside in the model runner, which is responsible for executing the actual KV cache transfers between the inference engine and LMCACHE 's KV cache backend.

Putting it together, when a query comes in, the scheduler first calls `get_num_new_matched_tokens` which queries LMCACHE to see cache hit tokens in the backend. Then the `update_state_after_alloc` function decides whether each page in vLLM needs to be loaded from external storage backend based on matched tokens information from LMCACHE. If the cache hit tokens are greater than zero, `build_connector_meta` function is called to prepare necessary metadata to load or store KV cache from storage devices.

Once the query reaches the model runner, in the case of layerwise pipelining, `start_load_kv` is called to start loading KV cache of the first layer to GPU memory. Then before each layer's LLM inference computation starts, `wait_load_kv` is called to synchronize the KV cache loading for this layer, and starts the KV cache loading for the next layer. After each layer's inference computation, in the layerwise case, `wait_store_kv` is called to wait until the KV cache for the previous layer has finished storing, and then calls `start_store_kv` to start the storing of KV cache for the newly generated KV cache layer.

In the case of non-layerwise pipelining, before the first

layer's LLM inference starts, `start_load_kv` is called to load the entire KV cache to GPU memory in a blocking manner. LLM inference will happen after the KV cache is put to the right GPU memory paged addresses. Then after the LLM inference has done for the current scheduling iteration, `start_store_kv` is called store the generated KV cache to lower-tier storage synchronously.

6 CONTROLLER INTERFACES

There are many scenarios where higher-level applications need to see KV cache locations and operate on that. For example, query routing can benefit from knowing where the largest number of prefix cache hit tokens is at. To support such use cases, LMCACHE exposes APIs, as shown in Table 3, for querying and manipulating KV cache, such as locating cached entries or migrating them across instances.

The KV cache controller in LMCACHE consists of two components: a centralized manager and per-instance workers. Incoming controller queries are handled by the centralized manager, which dispatches the appropriate operations to workers running within each LMCACHE instance. The remainder of this section illustrates how to use the controller interfaces through several example applications.

KV cache–aware routing: In this case, higher-level query routers would like to direct queries to instances with the largest number of prefix cache hits. The router first calls `lookup(tokens)`, where the centralized manager dispatches the lookup queries to individual LMCACHE instances, to identify the instances containing matching prefix tokens, then invokes `query_ip(instance_ids)` to map instance ids to IP addresses. Finally, the query is forwarded to the IP with the highest number of hits.

KV cache migration: When an instance holding KV cache fails or load balancing is required, the KV cache may need to be migrated across instances. The `move(source, destination, tokens)` interface moves the KV cache corresponding to `tokens` from the source instance to the destination.

KV cache clearance: Applications may clear cache when switching models or reclaiming memory. The `clear(tokens, instance, location)` function removes the KV cache corresponding to `tokens` stored at a specific storage `location` (*e.g.,* GPU memory or CPU memory) within the given `instance`.

Pinning KV cache in GPU memory: Some contexts, such as system prompts in chatbot applications, should remain in GPU memory for fast access. The `pin(instance, location, tokens)` function pins the specified KV cache in the chosen storage tier (GPU memory) of the given instance.

There are other more advanced interfaces users can call to manipulate the KV cache, such as the `compress(tokens, instance, location, compression_method)` function which compresses KV cache stored in a specific `location` in a LMCACHE instance with `compression_method` so that the KV cache takes smaller amount of space in storage. Users can freely call these interfaces to explicitly manage the KV cache in their own applications based on their needs.

7 EVALUATION

7.1 Setup

We evaluate LMCACHE under three different scenarios, as shown in Table 4. The three scenarios are representative setups that are commonly used by the users of LMCACHE.

Models: We compare LMCACHE against baseline solutions on popular open source models adopted by industry: `meta-llama/Llama-3.1-8B-Instruct`, `meta-llama/Llama-3.1-70B-Instruct`, `Qwen/Qwen2.5-Coder-32B-Instruct`, `Qwen/Qwen3-Coder-480B-A35B-Instruct-FP8`, `Qwen/Qwen2.5-72B-Instruct`.

Datasets: LMCACHE is evaluated on several datasets, including emulated multi-round question answering, long context question answering from LongBench (Bai et al., 2024), and random dataset from vLLM official benchmarking script (Kwon et al., 2023c).

Hardware: For single-node evaluation, we run LMCACHE on an 8×H100 server provided by GMI Cloud (GMI Cloud, 2025). Because different models require varying numbers of GPUs to be served, we allocate the minimum number of H100 GPUs necessary to successfully start each model in our evaluation. For multi-node evaluation, we use the same number of GPUs as in the single-node setup and configure a remote storage backend that leverages CPU memory for KV cache storage. For PD disaggregation, the prefiller and decoder instances are both set up with the number of GPUs as in single-node evaluation, and the prefiller and decoder instances are connected with NVLink.

Metrics: For each experiment, we show both time-to-first-token (TTFT), which is the prefill delay, and inter-token-latency (ITL), which is the average delay between the generation of two consecutive output tokens. For component-wise analysis which breaks down the delay for CPU offloading or PD disaggregation, we report the delay for each component separately.

Interface	Description
lookup(tokens) → {instance_id: hit_tokens}	Returns the instance names of KV caches containing prefix-match tokens with the given token list.
query_ip(instance_ids) → IP	Returns the IP addresses for the corresponding `instance_ids`
move(source, destination, tokens)	Moves the KV cache of the given token list from a source instance to destination.
clear(tokens, instance_id, storage_device)	Clears the KV cache for corresponding tokens from the `storage_device` in `instance`.
pin(tokens, instance, storage_device)	Pins the KV cache for corresponding tokens at the `storage_device` in `instance`.
compress(tokens, instance, storage_device, compression_method)	Compresses the KV cache for the corresponding tokens with a compression method and store it at the `storage_device` in `instance`.

Table 3. Functions in LMCACHE's connector.

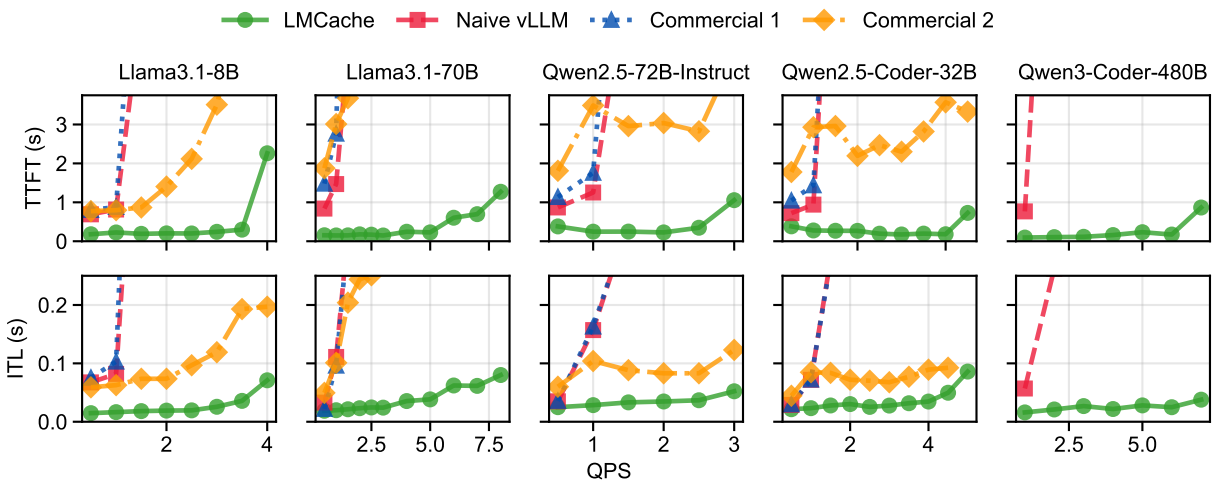


Figure 5. Single-node evaluation results.

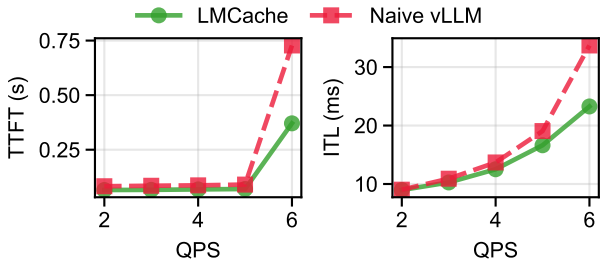


Figure 6. Real-trace evaluation results.

Baselines: We compare LMCACHE with several baselines, including 1) vLLM with prefix caching, which implements prefix caching, but only keeps limited amount of KV cache inside GPU memory; 2) commercial offering A, B, and C, which provide dedicated endpoint service that reserves GPUs for users to run a user-defined model.

Scenario Acronym	Single-node / Multi-node	Network Medium	Real-world Examples
CPU Offload	Single-node	N.A.	Single-node CPU Offloading
Central Storage	Single-node	Ethernet	Centralized Storage Server
PD	Single-node	NVLink	PD Disaggregation

Table 4. Evaluation scenarios setup.

7.2 Single-node CPU Offloading

We first evaluate LMCACHE on the CPU Offload scenario as in Table 4. In this experiment, we use multi-round Q&A workloads that emulate a typical chatbot-based document analysis scenario. By default, each LLM query contains 10K tokens, consisting of a document (roughly a 12-page PDF) used as context and a unique short question. Llama-3.1-8B-Instruct model takes 20K tokens as input, since smaller

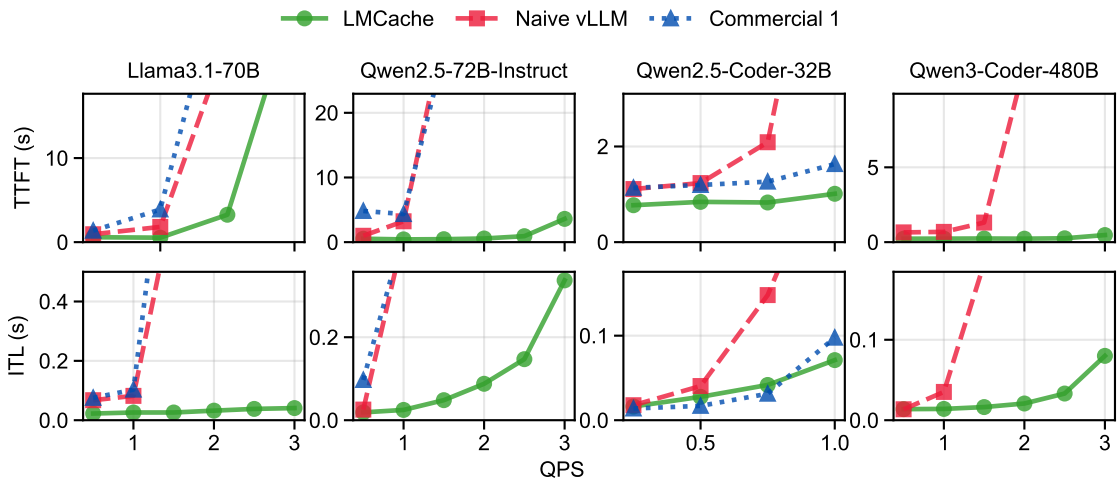


Figure 7. Remote backend evaluation results.

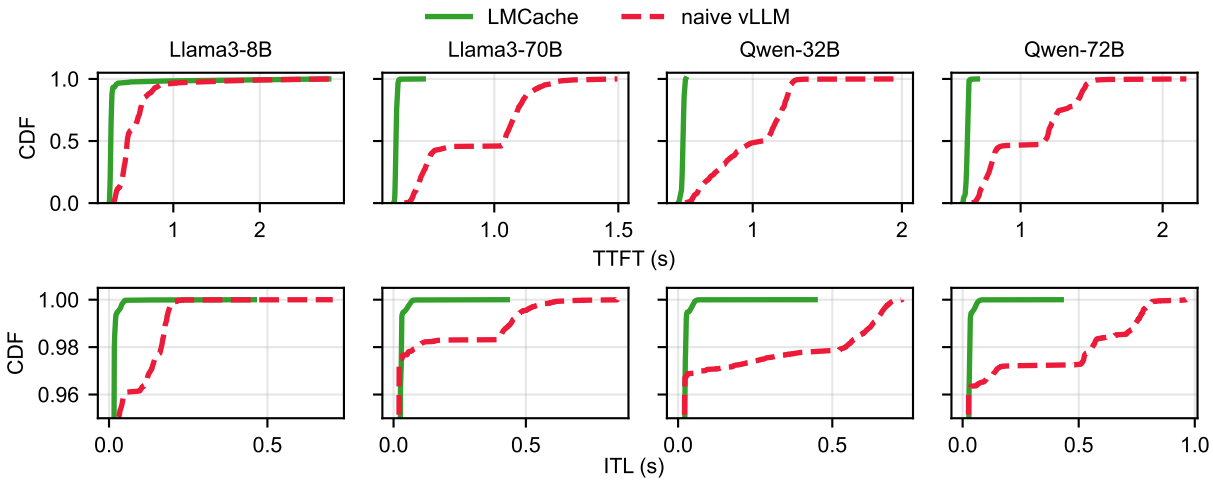


Figure 8. TTFT and ITL CDF comparison for PD.

models are generally better can handle more and longer queries. The LLM output is a short answer of 100 tokens at max. The chat session begins with 40 users, and additional users join according to a specified arrival rate (QPS). We set the maximum CPU memory LMCACHE can offload KV cache to 500 GB.

As shown in Figure 5, LMCACHE consistently outperforms all alternatives in both TTFT and ITL. For instance, under low QPS (e.g., QPS = 1), LMCACHE achieves 2.3–14× higher query processing rate (*i.e.,* throughput), at the same TTFT, than the strongest baseline across five evaluated models. In terms of ITL, LMCACHE also outperforms the baselines, as they incur a long delay before generating the first token, which in turn causes subsequent token generation to be queued. For Qwen3-Coder-480B, commercial options #1 and #2 do not provide support for hosting the model.

Understanding LMCACHE's gains: LMCACHE outperforms baselines for several reasons. Compared with naïve vLLM prefill, which caches KV data only in GPU memory, LMCACHE leverages CPU offloading. Since CPU memory can hold far more KV cache than GPU memory, LMCACHE achieves significantly higher cache hit ratios. With its efficient CPU-to-GPU loading scheme, cached data can be fetched quickly to accelerate inference. Our comparison with closed-source commercial alternatives is conducted in a black-box manner since their internal implementations are not publicly available. From the end-to-end results, we hypothesize that Commercial Option #1 lacks a KV cache

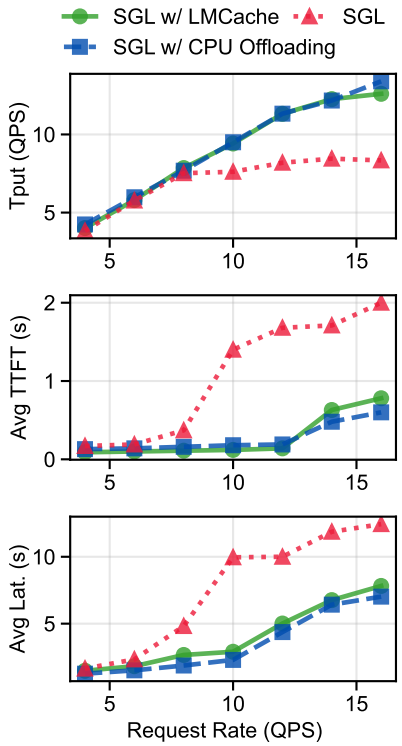


Figure 9. Comparing to SGLang CPU Offloading result.

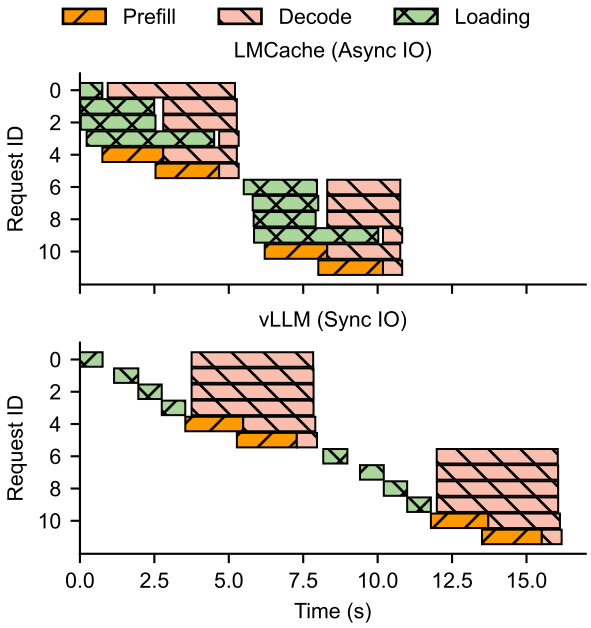


Figure 10. With request asynchronization, LMCACHE overlaps KV cache loading and inference computation (either prefill or decode).

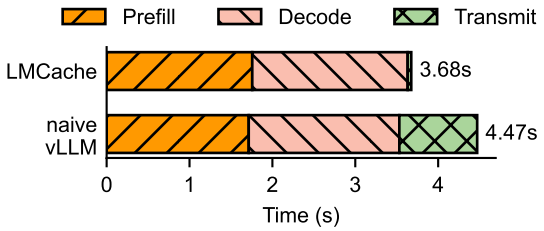


Figure 11. Latency breakdown of PD.

offloading mechanism to secondary storage. In contrast, Commercial Option #2 likely supports KV cache offloading to secondary storage, yet its performance is still worse than LMCACHE.

7.3 Centralized Storage Server

Next, we run LMCACHE for KV cache sharing through a centralized remote server, that is connected to the GPU instance with a bandwidth of 15 Gbps, following the setup of central storage in Table 4. For this experiment, we evaluate using the TriviaQA dataset from LongBench (Bai et al., 2024), a widely adopted benchmark for long-context evaluation. We follow the official vLLM benchmarking scripts (Kwon et al., 2023c), which generate inference queries according to a Poisson distribution at a specified QPS.

As shown in Figure 7, LMCACHE consistently outperforms all baselines across different QPS levels, providing 1.3–3× improvement in inference throughput. The improvement comes from the fact that the remote backend can store far more KV cache than CPU memory, thereby achieving higher cache hit ratios.

We note, however, that loading KV cache from the remote backend introduces greater latency than loading from CPU memory, since the remote backend has a much lower bandwidth. As a result, the loading delay may even surpass the prefill delay, particularly when the input context is short or model is small, as prefilling is too fast in such cases—a scenario that we will demonstrate later in 7.7.

7.4 PD disaggregation

In this experiment, we evaluate the performance in a PD disaggregation setting. Here, we compare LMCACHE with vLLM's native PD disaggregation with the official benchmarking script for random input and output workload. We use 8K tokens input and 200 tokens output. As shown in Figure 8, it presents the 95th percentile TTFT for both LMCACHE and vLLM's native PD disaggregation, showing that LMCACHE achieves significantly better tail latency. In terms of mean TTFT, LMCACHE also greatly outperforms

vLLM native PD disaggregation. Specifically, LMCACHE reduces mean TTFT by 1.53–1.84×, and reduces mean ITL by 1.12–1.66×, across the four models.

The performance gains of LMCACHE over the baseline stem from its more efficient design for PD disaggregation. Specifically, LMCACHE copies each chunk of the KV cache (generated during chunked prefill) to a buffer in the GPU memory of the prefiller instance, and then transfers it to the corresponding buffer on the decoder instance. Once received, the KV cache is copied into the paged memory of the decoder instance.

In contrast, vLLM's native PD disaggregation sends the paged KV cache generated by the prefiller directly to the decoder, using NIXL's memory copy function. This function takes as input the memory addresses of the KV cache pages on the prefiller side and copies them to the destination addresses on the decoder side. However, when the paged memory for the KV cache is scattered across the prefiller's GPU memory, the transfer is performed in a page by page manner, which leads to bandwidth underutilization, as discussed in §4.

7.5 Real-trace Driven Evaluation

We also evaluate LMCACHE on a real production trace provided by company F[4]. The trace includes query arrival timestamps and actual multi-round chat inputs. Notably, later rounds within the same user session often reuse conversation history from earlier rounds, resulting in prefix reuse. On average, the inputs in the trace are about 4K tokens long.

Since we do not have access to company F's proprietary models, we replay the trace using the Sao10K/L3-8B-Lunaris-v1 model. To make the experiment tractable, we stretch the original trace which lasts for several days and apply random sampling so that the workload we run completes within one hour.

As shown in Figure 6, LMCACHE consistently outperforms naive vLLM on the real trace on different QPS, reducing both TTFT and ITL, demonstrating its effectiveness in real-world settings. Note that for lower QPS from 2 to 5, LMCACHE still outperforms naive vLLM by 25%, while when QPS is 6, LMCACHE has improvement of 49% compared to naive vLLM.

7.6 Component-wise Evaluation

To further understand the gain brought by LMCACHE, we also perform component-wise analysis to break down the delay of each component in the end-to-end system.

PD disaggregation: Figure 11 shows the latency breakdown of LLM inference, including both prefill and decode computation, as well as the transmission of KV cache between prefiller and decoder instances. The prefill and decode computation times are the same for LMCACHE and vLLM's native PD disaggregation. However, as discussed in §7.4, vLLM's native design transmits KV cache at a much finer granularity, which results in bandwidth underutilization. In contrast, LMCACHE employs a more efficient KV cache transfer mechanism, enabling significantly faster transmission and thereby reducing the overall end-to-end delay in PD disaggregation.

Method	Achieved Bandwidth
LMCache	400 Gbps
vLLM's Native CPU Offloading	88 Gbps

Table 5. LMCACHE achieves much higher loading bandwidth when loading KV cache from CPU memory, compared to vLLM's native CPU offloading.

CPU offloading: In Table 7.6, we perform an ablation study to test the achieved loading bandwidth from CPU for LMCACHE and vLLM's native CPU offloading. The reason LMCACHE achieves higher transfer bandwidth than vLLM's native CPU offloading is due to the transfer granularity. While native CPU offloading performs data movement page by page, LMCACHE transfers data chunk by chunk. Each transfer operation triggers a CUDA memory copy, which involves preparing metadata beforehand and sending a completion signal afterward. These per-transfer operations add overhead to every memory copy kernel. By transferring larger chunks of data per copy, LMCACHE reduces the overall overhead, resulting in a much higher effective bandwidth.

Asynchronous Compute: We also show the benefit of LMCACHE's asynchronous compute in terms of reducing end-to-end delay. Figure 10 shows the timeline of queries loading and inference computation. The figure is drawn from the middle of a longer run for illustration purpose. As shown in the figure, without query asynchronization, prefill/decode computation and loading happen sequentially. With query asynchronization, the prefill/decode computation can overlap with KV cache loading, which reduces the end-to-end delay by 1.46×.

7.7 Sensitivity Study

We also perform several sensitivity evaluation to see how LMCACHE's delay changes under different context lengths and different types of remote backends.

Impact of context lengths: Figure 12 shows the prefill de-

[4]Anonymized due to NDA.

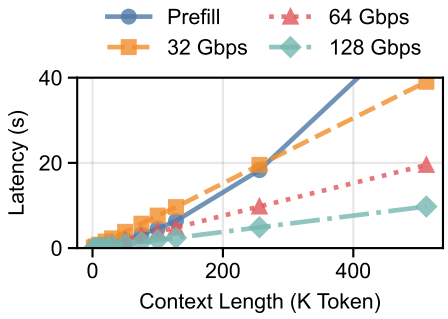


Figure 12. Impact of different context lengths.

lay on B200 machines and the loading delay under different network bandwidths. When the network bandwidth is low (*i.e.,* 32 Gbps), LMCACHE's KV cache loading outperforms naive prefilling only when the input context length exceeds 256K tokens. In contrast, when the bandwidth is higher (*i.e.,* 64 or 128 Gbps), LMCACHE's loading consistently achieves lower delay than naive prefilling across all context lengths. These results suggest that LMCACHE's KV cache loading should be adaptive: under low bandwidth, loading should be enabled only when the context length surpasses the crossover point where loading becomes faster than prefilling.

7.8 SGLang Results

Although our primary evaluation uses vLLM, we also evaluate LMCACHE integrated with SGLang. Figure 9 reports results for Qwen3-32B served on two H100 GPUs (TP=2) with LMCACHE 's CPU offloading enabled. Compared to SGLang without CPU offloading, LMCACHE achieves higher throughput and lower mean TTFT and mean end-to-end latency. Compared to SGLang's native CPU offloading, LMCACHE achieves comparable performance. These results confirm that LMCACHE is also effective on another inference engine. Although SGLang's native CPU offloading achieves performance comparable to LMCACHE 's CPU offloading on SGLang, it lacks a distributed storage backend capable of efficiently offloading data across a hierarchical set of storage devices, such as local disks and remote CPU/disk resources.

8 REAL-WORLD LESSONS AND EXPERIENCE

8.1 Scaling KV Cache Storage and Offloading

One clear trend in the adoption of LMCACHE in production systems is that more and more KV cache needs to be stored, definitely beyond the number that can be stored in GPU memory. Early deployments kept all KV caches in GPU memory, but long contexts and multi-user workloads may quickly exhaust GPU memory. Thus, many companies now need to offload KV data to CPU RAM, CPU memory pooling, or even disks when needed. This prevents evicting useful cache entries — without offloading, a follow-up query would trigger full prefill of the same input context, often long contexts, which greatly prolongs TTFT.

Interestingly, some teams, including company R, have experimented with remote storage devices. A remote data fetch is orders of magnitude slower than GPU memory access, yet real-world tests have shown it can still yield both speedups and cost savings. Retrieving previously computed KV cache over the network can be faster than recomputing them on loaded GPUs, with LMCACHE's KV cache loading optimizations (§4). While the overhead of loading KV cache from remote is large, it can be pipelined with inference computations, either prefilling requests that do not have cache-hit tokens or decoding computations. These experiences show that even slow and cheap storage can be leveraged to efficiently store and load KV caches.

Another technique used in production is the combination of CPU offloading and PD disaggregation. At company T, on the prefiller side, the KV cache is not only sent to the decoder side, but is also offloaded to the CPU memory of prefiller instance to achieve both the benefit of CPU offloading and PD disaggregation.

8.2 Emerging Use Cases and Surprising Outcomes

While chatbots were one of the first obvious beneficial usage scenarios of KV cache reuse, new use cases are rapidly emerging. Recently, large-scale recommendation systems from company L and company M have also adopted LLMs as the recommender model and becomes an important use case of KV cache reuse. In these systems, LLMs are used as the embedding models to prefill the contexts, without generating tokens, and are frequently reused by different queries of the same user. Furthermore, the contexts are often very long. Caching those KV cache can directly eliminate the expensive prefill phase, thus is a perfect use case of LMCACHE.

Although multi-round chat has long been a canonical use case for KV cache reuse, real-world deployments have revealed several important insights. Many systems initially adopted sliding context windows to fit more tokens into limited GPU memory during inference. However, this approach often degrades generation quality by discarding useful history and reduces prefix cache hit ratio, since truncated contexts are no longer prefixes. Deployment experiences at Company T and F demonstrate that keeping the full conversation history, while offloading KV caches to larger memory tiers, is a more effective practice. By avoiding truncation, the system achieves significantly higher cache hit rates on

shared prefixes, leading to less GPU memory usage per request. Moreover, in production environments, companies have observed that prefix cache hit rate is far higher than they expected.

The experience from Company B's adoption shows that industry users are accepting lossy optimizations, such as KV cache compression, for better system performance. Especially, in open-ended chatbot applications, where there is no single "correct" answer, using compressed KV cache (*e.g.,* quantized KV cache) may not harm the user experience but can dramatically reduce memory usage and I/O time. Even in finance companies, teams found that slight differences from using a compressed KV cache are tolerable if the overall system throughput improves.

8.3 Integration and Deployment Challenges

Deploying a caching layer in production LLM inference systems has its own practical challenges and lessons. One key lesson we found in the adoption of LMCACHE is that many companies preferred containerized environments, such as Docker images, when deploying the caching layer into production, as they often do not go into the source code of LMCACHE.

Another important aspect in the caching layer for LLM inference is fault tolerance and transparency. Because the caching layer sits beneath the inference engines and is supposed to be invisible to end users, robustness is important. Many companies stressed that if the cache system fails or underperforms, it should not bring down the model service. For instance, LMCACHE incorporates fault-tolerant KV cache retrieval: if errors occur during retrieval, LMCACHE reports which portions of the KV cache were successfully read, ensuring the inference engine does not break due to the unsuccessful read of KV cache and the correctness of generation is preserved.

We also observed a growing demand for a clean separation between the model serving engine and the caching engine. Many storage companies, such as company I, R, C, and W, have explicitly asked for decoupling the KV cache management and the core inference code, since they want to touch as minimum core inference logic as possible. Although LMCACHE does not solve this for now, the trend is clear: an LLM inference engine is pairing with a variety of external KV cache management services.

At last, we found that ensuring compatibility between LMCACHE and emerging inference frameworks was easier than expected. LMCACHE was originally designed and developed on top of vLLM. However, integrating LMCACHE with other frameworks, such as SGLang, has also been straightforward, as long as they adopt the paged attention mechanism. Looking ahead, we expect that LMCACHE will be equally easy to integrate with other inference frameworks based on paged attention.

8.4 Adoption, Lessons Learned, and Future Outlook

Our journey began as an academic prototype in the summer of 2024. At that time, KV caching was a niche concept mostly discussed in a few research papers. But by spring 2025, the landscape had changed — KV cache became a key technique in industry for improving the efficiency in LLM inference. The adoption of LMCACHE truly exploded in early 2025, with numerous companies starting to collaborate and use.

Interestingly, although LMCACHE has initially been an academic prototype, we found that LMCACHE has much more industry users than academia ones. One potential reason is that there is lack of an easy set of programmable interfaces. As a concrete example, a rich set of research literatures explore dropping tokens from KV cache for compressing it into smaller tensors. However, it is non-trivial to implement token dropping in LMCACHE, since LMCACHE assumes the number of tokens that is input to and output of LMCACHE should be the same. Furthermore, any intermediate states in the attention module are hidden from the LMCACHE's side, so research works that need to access or modify the attention states are hard to be implemented with LMCACHE.

Finally, we learned that programming languages for high performance are not always the best fit for building LLM inference infrastructure. Early on, some organizations (*e.g.,*, company R and I) assumed that KV cache management must be implemented in Rust or C++ to achieve high efficiency. In contrast, we developed LMCACHE in Python, which turned out to be a more effective choice. Python not only simplifies integration with existing inference frameworks but also lowers the barrier for community contributions, enabling rapid feature development and iterations. That said, CUDA-based implementations are still essential for certain high-performance data loading modules within LMCACHE.

9 CONCLUSION AND OUTLOOK

This paper presented LMCACHE, the first open-source and most widely adopted production-ready KV caching layer for enterprise-scale LLM inference. By treating KV cache as a first-class data structure rather than an internal byproduct of inference, LMCACHE transforms LLM engines from isolated token processors into a distributed ecosystem of compute and storage. Evaluation across diverse workloads and models demonstrates that LMCACHE consistently delivers significant throughput improvement and latency reduction compared to both open-source baselines and commercial inference APIs. Beyond performance, LMCACHE has already

seen rapid adoption in production environments, where enterprises leverage its CPU offloading, hierarchical storage, and PD disaggregation capabilities to keep low latency and reduce cost in trillion-token–scale deployments. Real-world deployments have also revealed new opportunities, such as KV cache reuse in recommendation systems and lossy compression in open-ended chatbots, underscoring the versatility of LMCACHE across application domains.

Looking ahead, LMCACHE points to a broader shift: **AI-native data such as KV caches will increasingly serve as the substrate for scaling LLM inference and agentic workloads**. By establishing KV cache as a standardized storage and communication medium, LMCACHE lays the foundation for future systems that treat inference not as isolated sessions but as a persistent, cache-aware computation fabric. We hope that the design, optimizations, and deployment lessons presented in this paper will inform the next generation of LLM infrastructure, where AI-native data, such as KV caches, is not merely an optimization but a core primitive for efficient, reliable, and scalable inference.

The source code of LMCACHE is at: https://github.com/LMCache/LMCache.

10 ACKNOWLEDGEMENT

We would like to thank the LMCACHE community for their invaluable support and contributions, including Baolong Mao and Chunxiao Zheng for managing remote connectors, Martin Hickey for GitHub Infrastructure, Huaizheng Zhang, Siddhant Ray, Zhuohan Gu and Hanchen Li for writing and maintaining documentation, Rui Zhang for the deployment of LMCACHE, Qizheng Zhang and Hussain Mohammad for insightful feedback.

REFERENCES

Best 44 large language models (llms) in 2025. https://explodingtopics.com/blog/list-of-llms, 2025. Accessed: 2025-09-18.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench: A bilingual, multitask benchmark for long context understanding, 2024. URL https://arxiv.org/abs/2308.14508.

ByteDance. InfiniStore: Kv cache store for distributed llm inference. https://github.com/bytedance/InfiniStore, 2025. Accessed: 2025-09-10.

Caylent. Prompt caching: Saving time and money in llm applications. https://caylent.com/blog/prompt-caching-saving-time-and-money-in-llm-applications, 2024. Accessed: 2025-09-18.

Shiyang Chen, Rain Jiang, Dezhi Yu, Jinlai Xu, Mengyuan Chao, Fanlong Meng, Chenyu Jiang, Wei Xu, and Hang Liu. Kvdirect: Distributed disaggregated llm inference, 2024. URL https://arxiv.org/abs/2501.14743.

Weijian Chen, Shuibing He, Haoyang Qu, Ruidong Zhang, Siling Yang, Ping Chen, Yi Zheng, Baoxing Huai, and Gang Chen. IMPRESS: An Importance-Informed Multi-Tier prefix KV storage system for large language model inference. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 187–201, Santa Clara, CA, February 2025. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/chen-weijian-impress.

DeepSeek AI Contributors. deepseek-ai/3fs: A high-performance distributed file system for ai training and inference workloads. https://github.com/deepseek-ai/3FS, 2025a. GitHub repository, MIT License.

KServe Contributors. kserve/kserve: Standardized distributed generative and predictive ai inference platform for scalable, multi-framework deployment on kubernetes. https://github.com/kserve/kserve, 2025b. GitHub repository, Apache-2.0 license.

Databricks Research. How long should you train your language model? https://www.databricks.com/blog/how-long-should-you-train-your-language-model, 2024. Accessed: 2025-09-18.

Dayou Du, Shijie Cao, Jianyi Cheng, Luo Mai, Ting Cao, and Mao Yang. Bitdecoding: Unlocking tensor cores for long-context llms with low-bit kv cache, 2025. URL https://arxiv.org/abs/2503.18773.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention, 2024. URL https://arxiv.org/abs/2403.19708.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms, 2024. URL https://arxiv.org/abs/2310.01801.

In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa, editors, *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/a66caa1703fe34705a4368c3014c1966-Abstract-Conference.html.

GMI Cloud. Gmi cloud: Gpu cloud solutions for scalable ai & inference. https://www.gmicloud.ai/, 2025. Provides high-performance GPU infrastructure and services for AI training, inference, and deployment. Founded in 2023, based in Mountain View, CA. Retrieved September 15, 2025.

Simon Jegou, Maximilian Jeblick, Alessio Devoto, Jiwei Liu, and David Austin. Kvpress: Efficient kv cache compression for long-context llms, 2024. URL https://github.com/NVIDIA/kvpress. Version 1.2.0.

Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation, 2024. URL https://arxiv.org/abs/2404.12457.

Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Zhuoqing Mao. Compute or load KV cache? why not both? In *Forty-second International Conference on Machine Learning*, 2025a. URL https://openreview.net/forum?id=WOyOtaO6lQ.

Shuowei Jin, Xueshen Liu, Qingzhao Zhang, and Zhuoqing Mao. Compute or load KV cache? why not both? In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=WOyOtaO6lQ.

Wook Kwon et al. Demystifying nccl: An in-depth analysis of gpu-based collective communication. *arXiv preprint arXiv:2507.04786*, 2025. URL https://arxiv.org/abs/2507.04786.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023b. URL https://arxiv.org/abs/2309.06180.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023c. Software: vLLM. https://github.com/vllm-project/vllm/tree/main/benchmarks.

Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 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)*, pages 155–172, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lee.

Junyan Li, Yang Zhang, Muhammad Yusuf Hassan, Talha Chafekar, Tianle Cai, Zhile Ren, Pengsheng Guo, Foroozan Karimzadeh, Colorado Reed, Chong Wang, and Chuang Gan. Commvq: Commutative vector quantization for kv cache compression, 2025. URL https://arxiv.org/abs/2506.18879.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024. URL https://arxiv.org/abs/2404.14469.

Jiachen Liu, Jae-Won Chung, Zhiyu Wu, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and

enhancing quality-of-experience in llm-based text streaming services, 2024a. URL https://arxiv.org/abs/2404.16283.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving, 2024b. URL https://arxiv.org/abs/2310.07240.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024c.

llm-d Project. llm-d: A kubernetes-native high-performance distributed llm inference framework. https://github.com/llm-d/llm-d, 2025. Accessed: 2025-09-10.

Meta Engineering. Roce networks for distributed ai training at scale. https://engineering.fb.com/2024/08/05/data-center-engineering/roce-network-distributed-ai-training-at-scale/, Aug 2024. Accessed: 2025-09-18.

Chengyi Nie, Rodrigo Fonseca, and Zhenhua Liu. Aladdin: Joint placement and scaling for slo-aware llm serving, 2024. URL https://arxiv.org/abs/2405.06856.

NVIDIA Corporation. Nvidia dynamo: A datacenter-scale distributed inference serving framework. https://github.com/ai-dynamo/dynamo, 2025. Accessed: 2025-09-10.

NVIDIA Developer Forums. Why is the transfer throughput low when transferring small size data (gpu host/device transfers). https://forums.developer.nvidia.com/t/why-is-the-transfer-throughput-low-when-transferring-small-size-data-from-host-to-device-or-device-to-host/153962, 2020. Accessed: 2025-09-18.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2024. URL https://arxiv.org/abs/2311.18677.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025a. USENIX Association. ISBN 978-1-939133-45-8. URL https://www.usenix.org/conference/fast25/presentation/qin.

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2025b. URL https://arxiv.org/abs/2407.00079.

Ziran Qin, Yuchen Cao, Mingbao Lin, Wen Hu, Shixuan Fan, Ke Cheng, Weiyao Lin, and Jianguo Li. Cake: Cascading and adaptive kv cache eviction with layer preferences, 2025c. URL https://arxiv.org/abs/2503.12491.

Redis. Redis enterprise software reference — redis documentation. https://redis.io/docs/latest/operate/rs/references/, 2025. Accessed: 2025-09-10.

Zebin Ren, Krijn Doekemeijer, Tiziano De Matteis, Christian Pinto, Radu Stoica, and Animesh Trivedi. An i/o characterizing study of offloading llm models and kv caches to nvme ssd. In *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '25, page 23–33, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715297. doi: 10.1145/3719330.3721230. URL https://doi.org/10.1145/3719330.3721230.

Xiaoxiang Shi, Colin Cai, Junjia Du, and Zhihao Jia. Nexus:proactive intra-gpu disaggregation of prefill and decode in llm serving, 2025. URL https://arxiv.org/abs/2507.06608.

William D. Strecker. Vax-11/780: A virtual address extension to the dec pdp-11 family. In *Proceedings of the National Computer Conference*, pages 967–980, Montvale, NJ, 1978. AFIPS Press.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference, 2024. URL https://arxiv.org/abs/2406.10774.

The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, Yifei Zhang, Yiqing Zhu, Shuowei Jin, Gangmuk Lim, Binbin Chen, Zuzhi Chen, Xiao Liu, Xin Chen, Kante Yin, Chak-Pong Chung, Chenyu Jiang, Yicheng Lu, Jianjun Chen, Caixue Lin, Wu Xiang, Rui Shi, and Liguang Xie. Aibrix: Towards scalable, cost-effective large language model inference infrastructure, 2025. URL https://arxiv.org/abs/2504.03648.

The SGLang Team. Ome: Revolutionizing llm infrastructure with model-driven architecture. https://lmsys.org/blog/2025-07-08-ome/, July 2025. Blog post, LMSYS Org.

UCCL Team. Everything you want to know about kv cache transfer engine. https://uccl-project.github.io/posts/kv-transfer-engine/, August 2025. Blog post, August 13, 2025.

vLLM project. vllm production stack: Reference system for k8s-native cluster-wide deployment with community-driven performance optimization. https://github.com/vllm-project/production-stack, 2025. Version vllm-stack-0.1.7, released Sep 3, 2025.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads, 2024a. URL https://arxiv.org/abs/2410.10819.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks, 2024b. URL https://arxiv.org/abs/2309.17453.

Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. Strata: Hierarchical context caching for long context language model serving, 2025. URL https://arxiv.org/abs/2508.18572.

Huan Yang, Renji Zhang, Mingzhe Huang, Weijun Wang, Yin Tang, Yuanchun Li, Yunxin Liu, and Deyu Zhang. Kvshare: An llm service system with efficient and effective multi-tenant kv cache reuse, 2025. URL https://arxiv.org/abs/2503.16525.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11608–11620, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.623. URL https://aclanthology.org/2024.acl-long.623/.

Lingfan Yu, Jinkun Lin, and Jinyang Li. Stateful large language model serving with pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 144–158, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696086. URL https://doi.org/10.1145/3689031.3696086.

Hailin Zhang, Xiaodong Ji, Yilin Chen, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, Weipeng Chen, and Bin Cui. Pqcache: Product quantization-based kvcache for long context llm inference, 2025. URL https://arxiv.org/abs/2407.12820.

Yan Zhang, Fei Li, Yanzhao Tang, Bingsheng He, Xiaoyong Wu, and Jeffrey Xu Yu. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024. URL https://arxiv.org/abs/2403.05821.

Yilong Zhao, Shuo Yang, Kan Zhu, Lianmin Zheng, Baris Kasikci, Yang Zhou, Jiarong Xing, and Ion Stoica. Blendserve: Optimizing offline inference for auto-regressive large models with resource-aware batching, 2024. URL https://arxiv.org/abs/2411.16102.

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. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving, 2024. URL https://arxiv.org/abs/2401.09670.

Yang Zhou, Zhongjie Chen, Ziming Mao, ChonLam Lao, Shuo Yang, Pravein Govindan Kannan, Jiaqi Gao, Yilong Zhao, Yongji Wu, Kaichao You, et al. An extensible software transport layer for gpu networking. *arXiv preprint arXiv:2504.17307*, 2025.