

LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference

Yuhan Liu^{12*} Jiayi Yao^{12*} Yihua Cheng^{1*} Yuwei An¹ Xiaokun Chen¹ Shaoting Feng¹²

Yuyang Huang¹² Samuel Shen¹ Rui Zhang¹ Kuntai Du¹ Junchen Jiang¹

¹Tensormesh Inc. ²University of Chicago

KV cache has traditionally been stored in GPU memory to accelerate the decoding phase of large language model (LLM) inference. However, it is increasingly necessary to move KV caches outside GPU devices, to enable cache reuse across different queries, and inference engines. Our real-world usage statistics confirm this trend: over time, the total KV cache stored by users has grown rapidly, far exceeding the capacity of GPU memory. Despite this need, there lacks an efficient solution for offloading and transferring KV caches. 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) out of the GPU memory and shares them across engines and queries. LMCACHE supports both *cache offloading* (prefix reuse across queries) and *prefill-decode (PD) disaggregation* (cross-engine/GPU cache transfer). LMCACHE’s high performance and wide adoption stem from the following contributions: *(i)* highly optimized KV cache data movement powered by 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. Our 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. Large-scale adoption of LMCACHE in enterprise settings provide us valuable insights, for example, fetching KV cache from remote storage has unsurprisingly benefits to prefill delay, and that context truncation, which is a widely applied technique in industry, can greatly reduce prefix cache hit ratio by half. 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 millions of applications, from interactive customer support and code generation to retrieval-based document analysis and agentic workflows. To build the systems for LLM inference, KV cache, the intermediate states of large language model (LLM) inference, has

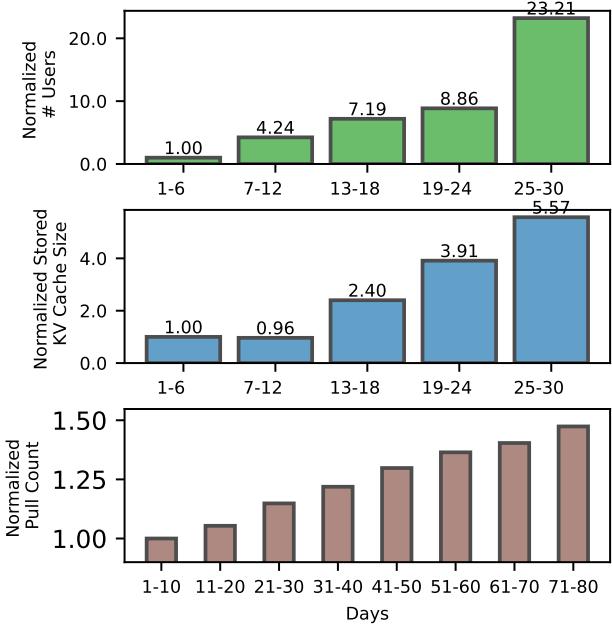


Figure 1: *Upper:* LMCACHE is being used by more and more users over time. *Middle:* LMCACHE is being used to store KV cache of larger size. *Bottom:* LMCACHE’s docker image pull counts have continued to increase.

now become a de facto optimization to make inference faster.

Traditionally, KV cache has been used during new token generation to skip re-computing the KV cache of the input prompts, *within a single query*. Thus, every query is processed independently by *one instance* of the inference engine, and a LLM query’s lifecycle, including computation and I/O operations, happens in the GPUs and the GPU memory of one inference engine.

However, recent trends have advanced to propose moving KV cache outside the GPU memory, including two emerging directions:

Cross-query caching to avoid redundant compute: KV cache can be *persisted* (*e.g.*, to lower-tier storage devices) beyond the lifecycle of a query to avoid any re-computation of the shared prefix for another query.

Prefill-decode disaggregation for higher utilization: There is an arising trend to decouple the prefill and decode phase on different GPUs, to ensure that latency-sensitive decoding phase is not affected by throughput-oriented prefill phase.

*Equal contribution

Such prefill-decode (PD) disaggregation requires KV cache produced by the prefill GPUs to be transferred to the decode GPUs.

Our real-world usage statistics confirm the trend of moving KV cache out from the GPU memory. As will be discussed in detail soon (§2.2), over time, the total size of KV cache that is stored by the users continues to grow, and it far exceeds the capacity of the GPU memory. This suggests that KV cache may need to be frequently evicted from GPU memory as more and more requests come in. Also, in order for it to be reused by another query, it needs to be offloaded from GPU memory, and then loaded back to GPU memory for reuse.

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 *re-loads* 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. \(2023\)](#) 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 \times higher throughput and at least 2 \times lower latency across diverse settings, including local prefix caching, distributed prefix reuse, and PD disaggregation.

Beyond quantitative gains, LMCACHE has seen adoption across several enterprises and open-source projects, providing useful insights and lessons in KV cache-driver optimizations at production scale. This includes surprising gain in latency brought by remote storage backend, reduction in prefix cache hit rate because of context truncation, and that the flexibility to evolve faster and integration smoothness is more important than the language performance itself.

The remainder of this paper details the motivation (§2) and challenges (§3), LMCACHE architecture and key design choices (§4, §5, and §6), deployment experiences (§9) and experimental evaluation (§8).

2 Motivation and Real-world Usage Statistics

2.1 KV Cache in LLM Inference

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.

Nowadays, the contexts have grow longer and longer, and people have started to augment inference with background knowledge. Given this trend, it is popular to share KV cache across different user queries to reduce the redundant computations for the long contexts or background knowledge.

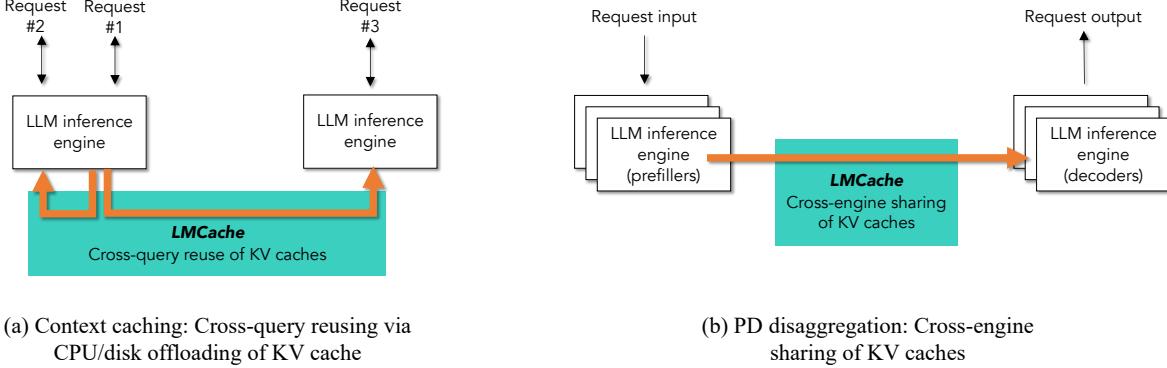


Figure 2: LMCACHE supports both context caching (*KV cache offloading and sharing across queries*) and PD disaggregation (*cross-engine transfer of KV caches*).

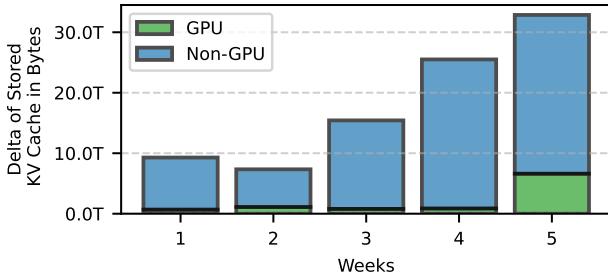


Figure 3: Weekly growth of KV cache size, including portions that fit in GPU memory and those that exceed it.

2.2 Real-world Usage Statistics

KV Cache Size Exceeds GPU memory: Although KV cache has been kept inside GPU memory for all traditional LLM inference systems, we observe that the required size of KV cache is now far exceeding the GPU memory capacity, from our real-world usage statistics, gathered by usage tracker voluntarily turned on by users.

Figure 3 shows the weekly growth of KV cache size over the past five weeks, for caches that fit within GPU memory (green) and those that exceed GPU memory capacity (blue). The portion of KV cache that no longer fits in GPU memory has increased significantly over time, showing that GPU memory alone is insufficient for storing all caches. To enable KV cache reuse across queries, especially those generated long before reuse, it becomes necessary to move KV cache out of GPU memory, for instance, by offloading it to CPU memory or other storage tiers.

Reuse per Token has Greatly Increased: We also observe that reuses per token has greatly increased over time.

As shown on the left side of Figure 4, where we plot the ratio between reused tokens and all stored tokens, beyond the GPU memory, plotted with top-10 users. We denote this ratio as reuses per token. The reuses per token has grown significantly over the past several weeks, which indicates that

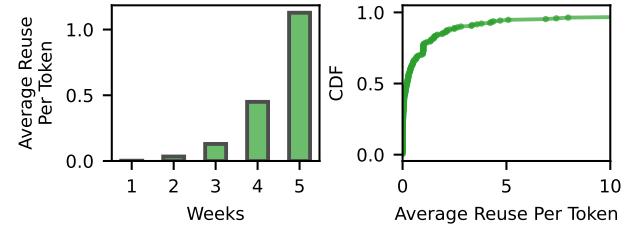


Figure 4: Left: Average reuse per token for top users. Right: Distributions of average reuse per token across different users.

tokens that cannot fit inside GPU memory are being more and more frequently reused by inference. This suggests that more and more tokens need to be loaded back to GPU memory.

On the right hand side, the figure shows the distribution of reuse per token for different users over the past week. More than 19% of users reuse stored tokens for more than 1.5 times, suggesting the trend of users accessing a token multiple times after it is stored.

2.3 Need an Efficient KV Caching Layer for Moving KV Cache

From the above observations from the statistics gathered in real-world deployments, we find two important trends of KV caching. First, the KV cache that cannot simply fits in GPU memory keeps growing, potentially due to growing length of the contexts or larger amount of user traffic. Second, reuses per token stored beyond the GPU memory has also increased over time. Both trends suggest that we need to move KV cache out of GPU memory. Specifically, in the current industry, two scenarios which move KV cache out from GPU exist:

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

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 \(2025\)](#).

long preamble. Prefix caching reduces redundant computation during the prefill phase, directly lowering TTFT and GPU-hours per query [Chen et al. \(2024, 2025\); Gao et al. \(2024\); Jin et al. \(2024, 2025\); Liu et al. \(2024a\); Qin et al. \(2025a\); Ren et al. \(2025\)](#).

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 (autoregressive token generation) across different GPUs or nodes. This approach reduces tail latency by maximizing the decoding speed without being interrupted by the prefill phase [Patel et al. \(2024\); Shi et al. \(2025\); Zhong et al. \(2024\)](#).

However, there lacks a library to support efficient extraction and loading from and to the GPU memory due to the system challenges as discussed soon (3).

3 Challenges of Efficient KV Caching and Related Work

3.1 Challenges of Efficient KV Caching

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

3.1.1 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. \(2023\)](#) 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 *underutilization* and reduce throughput [Kwon et al. \(2025\); Meta Engineering \(2024\); NVIDIA Developer Forums \(2020\)](#). 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. \(2025\)](#).

3.1.2 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.

3.1.3 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 company¹ that has worked closely with LMCACHE in the

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

production setting asked for an interface that allows users to *explicitly* pin frequently accessed financial documents 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.

3.2 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), l1m-d ILM-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, l1m-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 §8.

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 Chen et al. (2025); Gao et al. (2024); Gim et al. (2024); Jin et al. (2024, 2025); Kwon et al. (2023); Lee et al. (2024); Yang et al. (2025); Ye et al. (2024); Yu et al. (2025); Zhao et al. (2024); Zheng et al. (2024), PD disaggregation Patel et al. (2024); Shi et al. (2025); Zhong et al. (2024), and KV cache compression Du et al. (2025); Ge et al. (2024); Jegou et al. (2024); Li et al. (2025, 2024); Liu et al. (2024b); Qin et al. (2025c); Tang et al. (2024); Xiao et al. (2024a,b); Zhang et al. (2025). However, these prototypes are

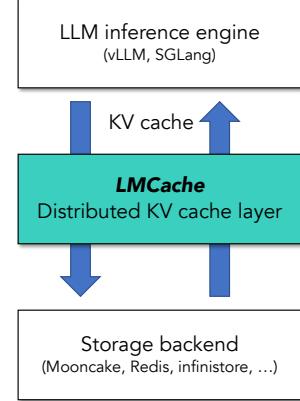


Figure 5: LMCACHE sits between LLM inference engines and heterogeneous storage/network devices.

typically built on research-oriented inference frameworks, 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.

4 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 5). 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.

Figure 6 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 §5.2. 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.

5 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 pages², 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.

5.1 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 chunk³. This is achieved using an intermediate *streaming GPU buffer*. For storing, the KV cache are first copied from the scattered paged GPU memory into a contiguous streaming buffer with a customized CUDA kernel, then offloaded collectively to lower-tier storage (e.g., CPU memory) with DMA engines at the granularity of chunks rather than individual pages. For loading, chunks are first retrieved from the storage layer into the GPU buffer with DMA engines and subsequently split into paged memory with CUDA kernels.

Parallel Store/load Operations: LMCACHE supports parallel storage and retrieval of KV caches across multiple storage tiers, including local CPU DRAM or disks, remote CPU DRAM or disks, and object storage (e.g., S3). In practical LLM serving workloads, KV caches often need to be migrated across devices concurrently—for instance, transferring the KV cache of a hot context from GPU to CPU memory while simultaneously offloading a cold context from CPU memory to local disk. To maximize link utilization, LMCACHE’s store and load APIs accept multiple source and destination devices, enabling concurrent data movement across heterogeneous links. Moreover, these operations can be executed in parallel when the interconnect supports full-duplex communication (e.g., PCIe).

Delayed Decode KV Cache Storing: LMCACHE also supports storing newly generated KV caches during decoding. Instead of offloading each token’s KV cache immediately—a naive approach that triggers frequent small writes—LMCACHE buffers KV caches and performs batched storage once a predefined number of tokens (i.e., a chunk) have been generated. This chunk-based delayed storing strategy reduces write frequency, minimizes I/O overhead, and significantly improves overall storage throughput.

5.2 Compute-I/O Overlapping

LMCACHE employs multiple optimizations aiming for overlapping LLM inference computations with I/O to maximize GPU utilization.

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

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

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

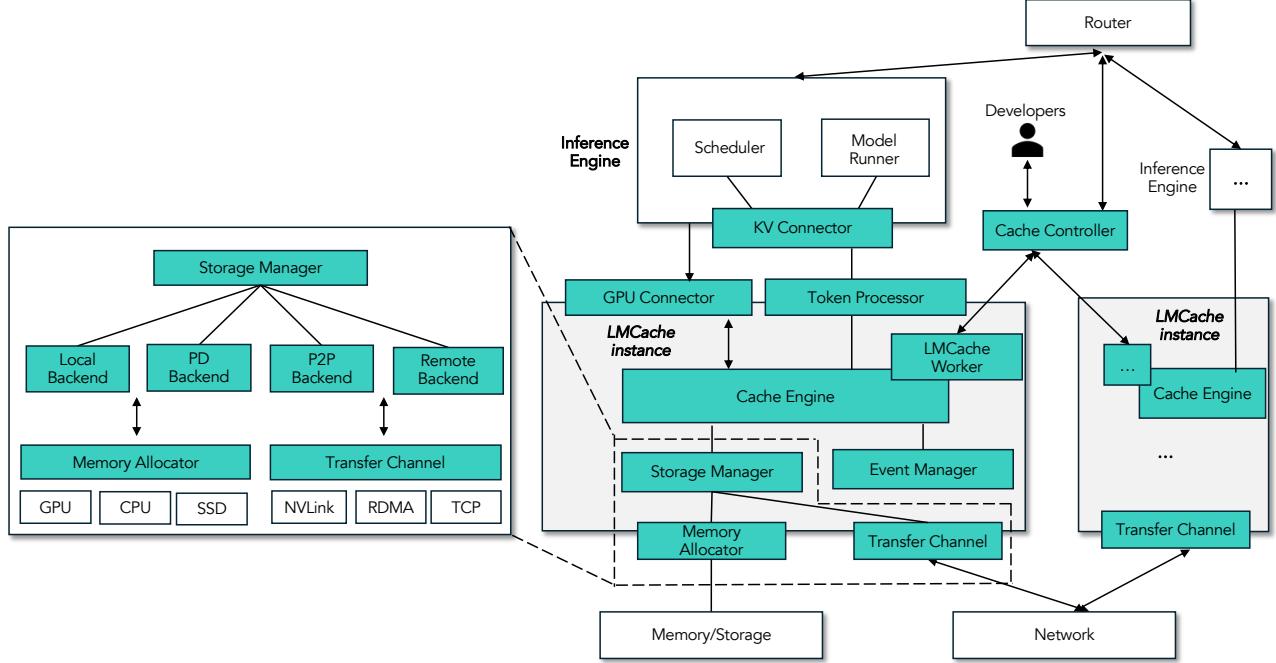


Figure 6: End-to-end system workflow for LMCACHE.

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 a query (with cache hit) arrives when the inference engine is processing other queries, the arriving query has to wait in queue. 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 or GPU memory). As a result, when the actual inference computation starts, the required KV cache can be loaded or used directly from faster-tier storage, significantly reducing loading delay. LMCACHE allows users to configure the target tier for prefetching based on their own needs in latency SLO and resource constraints.

5.3 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 multiple devices simultaneously, LMCACHE minimizes data duplication through a reference counter. Specifically, when KV cache is written to multiple destinations—such as writing from local CPU memory to local disk and remote object

storage at the same time—LMCACHE increments a reference counter on the shared data for each transfer instead of creating new copies. Each completed read or write decrements the counter, and once the count reaches zero, the data is released. This design ensures that data is shared across concurrent read and write operations without unnecessary replication, thus reducing memory pressure and improving efficiency. This technique is similar 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.
- **End pointer:** the end address of the free pages that are scheduled to be offloaded.

As illustrated in Figure 7, 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.

Function name	Description
get_num_new_matched_tokens(query) → Optional[matched_tokens]	Returns the number of cache-hit tokens found in LMCACHE’s backend. Returns None if the LMCACHE decides to let vLLM process other requests first and put this request back to waiting queue.
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.

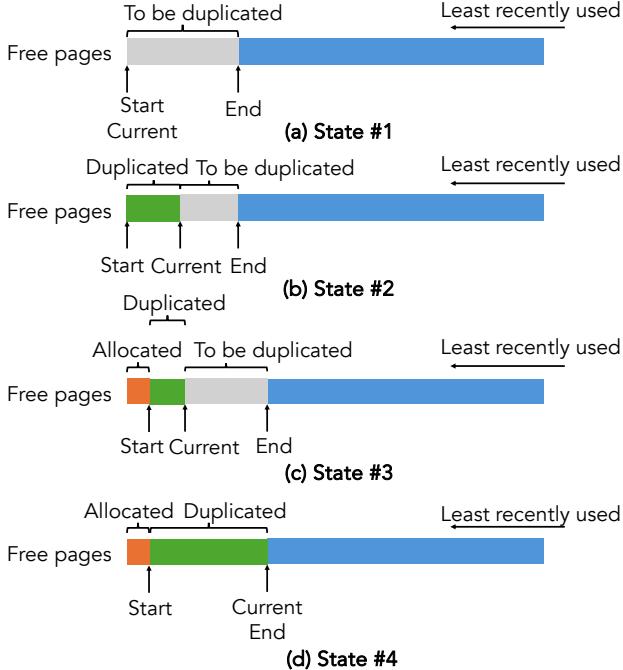


Figure 7: Illustration of dynamic offloading in LMCACHE.

- **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 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. Though not supported, the same dynamic offloading strategies can also be extended to other storage tiers beyond CPU and GPU.

6 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 upstream inference engine evolves.

We note that the design of this API is initiated by LMCACHE team, but the implementation and the maintainance of this API are the collaborative effort of both LMCACHE team and vLLM team.

Design objectives: The key design objectives are:

- Maximum flexibility: it enables as much KV cache operations as possible.
- vLLM-native: it aligns with the design direction of vLLM, including strict scheduler — worker separation, prefix caching as the first-class citizen, and piece-wise CUDA graph, where vLLM only captures CUDA graph for non-attention operations.
- Friendly to out-of-tree connector: it allows integrating with out-of-tree connectors without vLLM-side code modification.
- Minimum API-level overhead: it does not introduce overheads (*e.g.*, inter-process communication) at API level.

To be compatible with the scheduler — model runner separation philosophy in vLLM, the connector API contains two sets of interfaces: 1) the *scheduler*, where the extra cache-hit tokens from the connector are treated as normal prefix-cached tokens in vLLM and 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 we add hooks before and after model execution, and also before and after attention computation, to enable both bulky KV cache offloading and layer-wise KV cache offloading.

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. The function can return `None` if LMCACHE decides to let vLLM put the current request back to waiting queue and process other requests first, overlapping this request’s I/O with other requests’ computation. 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 to store the generated KV cache to lower-tier storage synchronously.

Impact: This API is out for over six months in vLLM. During these six months, we see open-source adoptions, including NVIDIA dynamo project, Ilm-d project from RedHat, AIBrix project from ByteDance, and vLLM production stack project. We also see multiple proprietary connectors from different companies that use the KV connector API.

7 Controller Interfaces

LMCACHE operates as a distributed caching system built around a centralized KV cache controller responsible for global metadata management, cache manipulation, and request routing. To support these functionalities, LMCACHE provides two categories of APIs: (1) external APIs, which are directly accessible to users or system operators, and (2) internal APIs, which are used by individual LMCACHE instances.

Mechanically, the KV cache controller consists of two layers: a centralized controller manager and per-instance workers. The controller manager runs as a standalone process and serves as a global coordination point, while per-instance workers collocate with each peer LMCACHE instance and handle local operations or issue global requests to the manager. External API calls are handled by the centralized manager, which, if necessary, dispatches the appropriate operations to each worker. Per-instance workers can also proactively interact with the centralized manager via internal APIs for metadata update or lookup.

The KV cache controller underpins a series of advanced optimizations, including cross-node KV cache sharing, cache-aware request routing, and dynamic KV cache migration. The remainder of this section demonstrates how these optimizations could leverage the controller interfaces through concrete examples.

KV cache-aware routing: In this case, higher-level routers aim to direct requests to the instance with the highest expected cache hit rate. Each LMCACHE instance reports its cache admission and eviction decisions to the controller manager via the `batched_admit` and `batched_evict` interfaces.

Internal APIs		Description
batched_admit/batched_evict(hashes, inst_id, device)		Send the KV admission/eviction messages from an LMCACHE instance to the controller manager.
batched_p2p_lookup(hashes) → list[inst_id, device, hit_chunks]		Lookup peer KV cache existence from an LMCACHE instance based on the given hashes.
External APIs		Description
lookup(tokens) → list[inst_id, device, hit_tokens]	Lookup the global KV cache existence of the given tokens.	
move((src_inst_id, src_device), (dst_inst_id, dst_device), tokens)	Moves the KV cache of the given tokens from source location (src_inst_id, src_device) to destination location (dst_inst_id, dst_device).	
clear(tokens, inst_id, device)	Clears the KV cache for corresponding tokens from the storage device device in instance inst_id.	
pin/unpin(tokens, instance, storage_device)	Pins/unpins the KV cache for corresponding tokens at location (inst_id, device).	
compress/decompress(tokens, instance, device, method)	Compresses/decompresses the KV cache for the corresponding tokens at location (inst_id, device) with a specified compression/decompression method.	

Table 3: APIs in LMCACHE Controller.

The controller manager aggregates these updates and maintains a global in-memory view of the KV cache state across all instances. When the router calls `lookup(tokens)`, the controller consults its in-memory global KV cache states and returns a list of `(instance_id, storage_device, hit_tokens)`, indicating where and how many of the requested tokens are currently cached.

KV cache migration: When an instance holding KV cache is about to be scaled down or load balancing is required, the KV cache may need to be migrated to another instance. The controller manager handles such operations through `move((src_inst_id, src_device), (dst_inst_id, dst_device), tokens)` API call by dispatching the request to the source instance. The source instance will try to establish connection to the destination instance if one does not exist, and transfers the specified KV cache from the source storage device `src_device` to the destination location indicated by `(dst_inst_id, dst_device)`.

P2P KV cache sharing: LMCACHE supports peer-to-peer KV cache sharing, allowing an instance to fetch KV cache from another peer when a local cache miss occurs. Upon a cache miss, the instance’s local worker can query the centralized controller manager via `batched_p2p_lookup`. The manager will return a list of `(inst_id, device, hit_chunks)`, representing the number of hit chunks and the location that hold these chunks. The instance can then choose to, for example, load KV cache from a peer with maximum `hit_chunks`.

KV cache clearance: Applications may clear cache when switching models or reclaiming memory. Upon receiving a `clear(tokens, inst_id, location)` call, the controller manager dispatches the operation to the corresponding instance identified by `inst_id`. The instance’s worker then removes the KV cache associated with tokens stored from a

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.

specific storage device device.

Some APIs are not covered in the above applications, such as `compress/decompress(tokens, inst_id, device, compression_method)` which compresses/decompresses KV cache stored in location `(inst_id, device)` with a specified `compression_method` and `pin/unpin(tokens, inst_id, device)` which can pin/unpin the specified KV cache at a certain location `(inst_id, device)`. Users can freely call these APIs to explicitly manage the KV cache in their own applications based on their needs.

8 Evaluation

8.1 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, Sao10K-L3-8B, meta-llama/Llama-3.1-70B-Instruct, Qwen/Qwen2.5-Coder-32B-Instruct,

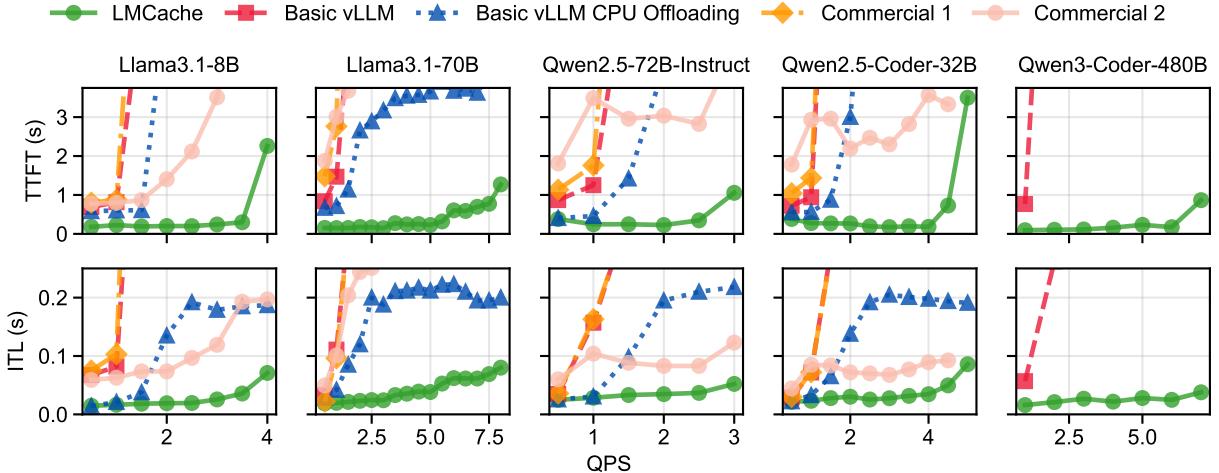


Figure 8: Compared to basic vLLM, basic vLLM CPU offloading, and two commercial alternatives, LMCACHE has 1.9–8.1× smaller TTFT, and supports 2.3–14× higher inference throughput. Basic vLLM CPU offloading fails to run on Qwen3-Coder-480B, and commercial alternatives do not have the option to deploy Qwen3-Coder-480B.

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. (2023).

Hardware: For single-node evaluation, we run LMCACHE on an $8 \times$ 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 centralized remote storage backend that leverages CPU memory for KV cache storage. For PD disaggregation, the prefller and decoder instances are both set up with the number of GPUs as in single-node evaluation, and the prefller 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.

Baselines: We compare LMCACHE v0.3.6 with several baselines, including:

- **Basic vLLM:** vLLM v0.10.2 which enables prefix caching by default, but only keeps KV cache inside GPU memory, so only a small portion of it can be kept;

- **Basic vLLM CPU Offloading:** vLLM v0.11.0 with its own implementation of CPU offloading;
- **Commercial offerings #1 and #2:** provides dedicated endpoint service that reserves GPUs for users to run a user-defined model. We ran these baselines accessed on September 10th.

8.2 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 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 8, LMCACHE consistently outperforms all baselines in both TTFT and ITL. For instance, under low QPS (e.g., QPS = 1), LMCACHE has 1.9 to 8.1× smaller TTFT. 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. Specifically, compared to the best baseline, LMCACHE has 7% to 92% smaller ITL, at QPS=1. For Qwen3-Coder-480B, commercial options #1 and #2 do not provide support for hosting the model.

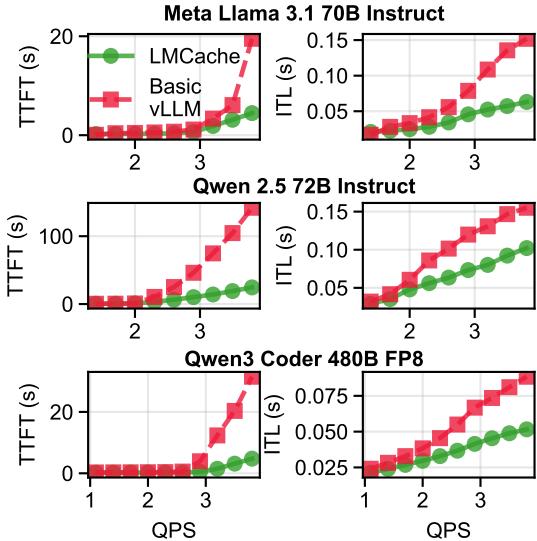


Figure 9: Comparing LMCACHE and basic vLLM on three different models based on real trace drawn from company F’s input and output distributions. LMCACHE has at least 4.4–6.6× smaller TTFT, and 34–58% smaller ITL, at high QPS.

Understanding LMCACHE’s gains: LMCACHE outperforms baselines for several reasons. Compared with basic vLLM, 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. Compared to basic vLLM CPU offloading, which employs per-layer and per-16-token transmission and is unable to fully utilize the loading bandwidth, LMCACHE has more efficient implementation of the transmission module as it loads KV cache at chunk level with high-performance data loading CUDA kernels. 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 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.

8.3 Real-trace Driven Evaluation

We evaluate LMCACHE on a real trace drawn from company F and company G’s distribution of input and output tokens. Since we do not have access to company F’s proprietary models, we run the trace using five different models. To make the experiment tractable, we stretch the original trace which lasts for several days so that the workload we run completes within one hour. LMCACHE is set to use 500 GB of CPU DRAM at maximum, and we compare it with latest basic

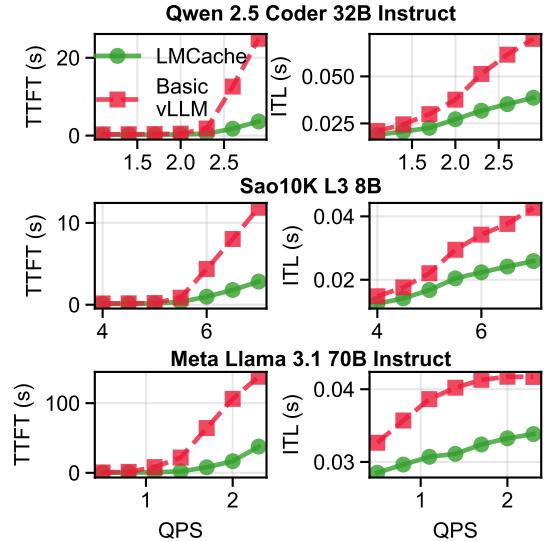


Figure 10: Upper and middle figures: comparing LMCACHE and basic vLLM on Qwen 2.5 Coder 32B Instruct and Sao10K L3 8B models based on real trace drawn from company F. Lower figure: comparing LMCACHE and basic vLLM on Llama 3.1 70B Instruct on company drawn from company G’s input/output data distribution. LMCACHE achieves 3.7–6.8× smaller TTFT and 19–44% smaller ITL.

vLLM with GPU prefix caching.

As shown in Figure 9 and Figure 10, LMCACHE consistently outperforms basic vLLM on the real trace on different QPS across the five models. Specifically, LMCACHE reduces TTFT by at least 3.7–6.8×, and reduces ITL by at least 19–58%, across five models at high QPS.

8.4 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. (2023), which generate inference queries according to a Poisson distribution at a specified QPS.

As shown in Figure 11, 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

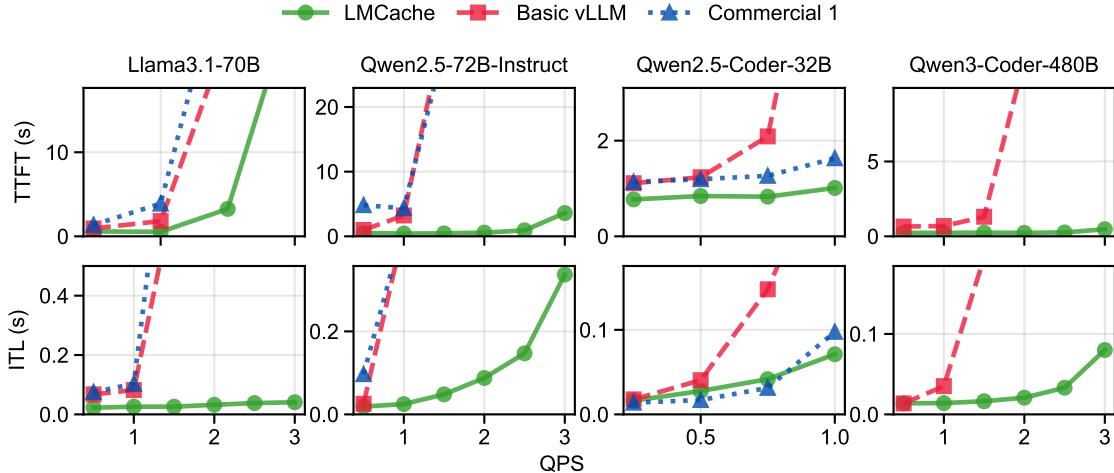


Figure 11: Compared to basic vLLM, LMCACHE with remote backend offloading has 1.3 to 3× improvement in inference throughput, under the same TTFT.

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 §8.7. Thus, adaptive decisions between KV cache loading and prefilling need to be made when KV cache resides in a remote storage server.

8.5 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 12, 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 prefller 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 prefller 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 prefller side and copies them to the destination addresses on the decoder side. However, when the paged memory for

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.

the KV cache is scattered across the prefller’s GPU memory, the transfer is performed in a page by page manner, which leads to bandwidth underutilization, as discussed in §5.

8.6 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 14 shows the latency breakdown of LLM inference, including both prefill and decode computation, as well as the transmission of KV cache between prefller and decoder instances. The prefill and decode computation times are the same for LMCACHE and vLLM’s native PD disaggregation. However, as discussed in §8.5, 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.

CPU offloading: In Table 5, 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

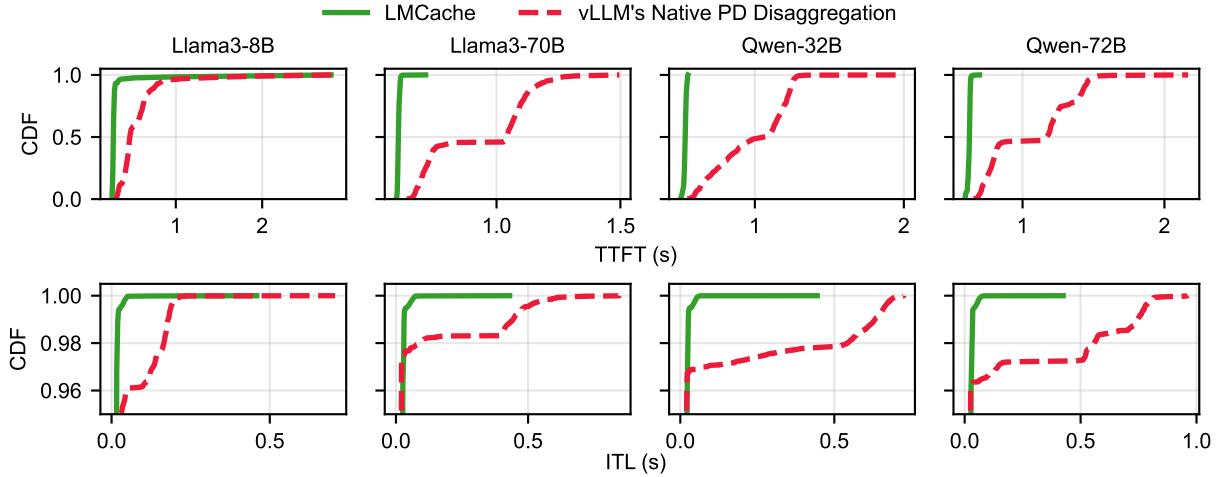


Figure 12: Compared to vLLM’s native PD disaggregation, LMCACHE’s PD disaggregation has significantly lower tail latency, and achieving 1.5–1.8× lower mean TTFT, and 1.1 to 1.7× lower mean ITL.

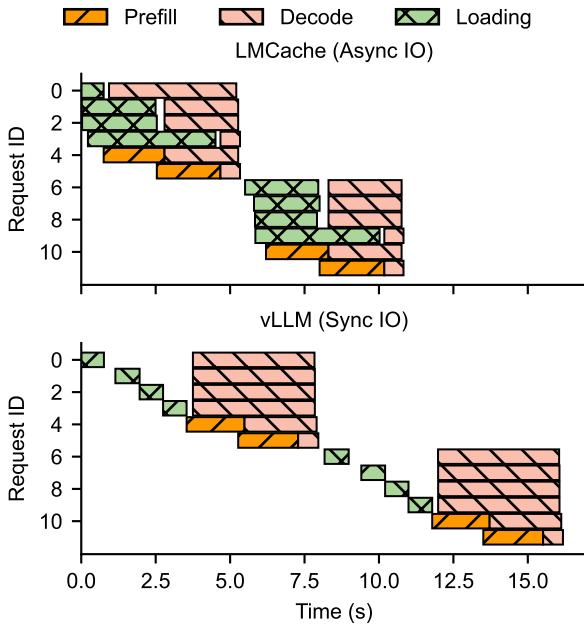


Figure 13: With request synchronization, LMCACHE overlaps KV cache loading and inference computation (either prefill or decode).

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.

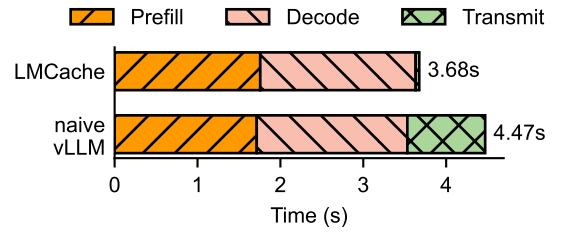


Figure 14: Compared to vLLM’s native PD disaggregation, LMCACHE achieves much smaller transmission latency, thus reducing end-to-end delay.

Asynchronous Compute: We also show the benefit of LMCACHE’s asynchronous compute in terms of reducing end-to-end delay. Figure 13 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 synchronization, prefill/decode computation and loading happen sequentially. With query synchronization, the prefill/decode computation can overlap with KV cache loading, which reduces the end-to-end delay by 1.46×.

8.7 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 15 shows the prefill delay 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

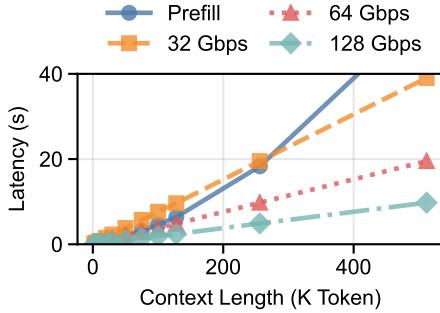


Figure 15: At network bandwidth of 32Gbps, LMCACHE’s KV cache offloading only outperforms basic vLLM’s prefill when input length is more than 256K tokens. At network bandwidth of 64 or 128Gbps, LMCACHE’s KV cache offloading is better than prefill across all input lengths.

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.

8.8 SGLang Results

Although our primary evaluation uses vLLM, we also evaluate LMCACHE integrated with SGLang. Figure 16 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.

9 Real-World Lessons and Experience

Loading from remote storage is faster than prefill: Traditionally, it was believed that loading KV cache from remote storage primarily improves cache hit rates and reduces storage costs by leveraging cheaper storage devices, but at the cost of increased inference latency, since loading data from remote devices was thought to be slower than performing a full prefill. This assumption was largely due to the historically low throughput of remote object stores such as Amazon S3, which

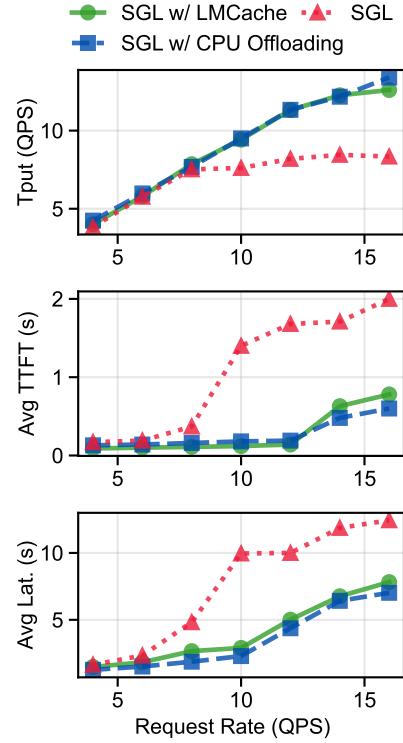


Figure 16: On Qwen3-32B model, LMCACHE’s CPU offloading achieves comparable performance as SGLang’s native CPU offloading.

offered loading speeds as low as 100 MBps. Recently, however, there has been a major boost in remote storage performance, for example, with Amazon S3 Express, the throughput has increased from 100 MBps to nearly 1 GBps. Our users, such as Company C, have adopted LMCACHE to load KV caches from their own remote object store, achieving 22–32% lower TTFT compared to full prefill. This insight suggests that remote backends can simultaneously improve cache hit ratios and reduce TTFT.

Context truncation lowers prefix cache hit rates: Many industry users employ a sliding-window mechanism to handle long-context inputs constrained by limited model context windows or GPU memory. For instance, when input tokens exceed the context window limit, some companies truncate the input to keep only the most recent tokens. However, this approach significantly reduces prefix cache hit ratios, since truncated inputs no longer match the prefixes of previously cached contexts. In practice, using real traces from Company F, we observe that prefix cache hit ratios drop from roughly 85% to 45% when truncating input contexts to keep only the latest tokens. Other studies have also discussed this phenomenon, emphasizing that dynamically adding or removing context tokens should be avoided, as it invalidates prefix KV cache reuse [Ji \(2025\)](#).

Containerized code is preferred: With the growing scale of LLM inference, most production environments now rely on Kubernetes to manage GPU clusters. Consequently, deploying inference engines (*e.g.*, vLLM or SGLang) and LMCACHE through containerized environments, typically via Docker images, has become the standard practice among industry users. Interestingly, many users rely solely on the official Docker images without going deep into LMCACHE’s source code.

Unexpected high cache hit rate in production systems: Our customers did not expect such a high prefix cache hit ratio, such as 50% hit rate for company G in their production environments, until they deploy LMCACHE inside their systems. Previously, people thought KV caches could only be reused for fixed system prompts. However, modern applications increasingly exhibit “dynamically reusable contexts”, such as conversation histories in coding assistants, chat applications, and retrieval-augmented generation (RAG) pipelines. These emerging patterns have significantly increased overall cache hit ratios in real-world deployments.

Industry vs. academia users: LMCACHE was designed as a unified prototype framework in May 2024, where we can put our research works into it to gain more impact. However, we found that at that time, industry wants a highly efficient KV cache offloading solution as the size of KV cache and concurrent users keep growing. Then, our attention has been moved towards improving the performance, stability, and compatibility. Since most companies are less concerned with customizing attention algorithms, we deprioritized designing flexible APIs for integrating specialized attention mechanisms, such as selective token dropping. This makes LMCACHE less popular in academia, since the academia users often focuses on modifying attention mechanisms for their research prototypes. As next step, LMCACHE is going to design more flexible APIs such that it is easy to use by both industry and academia.

Flexibility vs. performance for programming language: Python has always been the de facto language in ML. However, the current industry focus is gradually shifting towards higher efficiency instead of broad compatibility. Specifically, many companies are rewriting ML libraries in high-performance languages such as Rust or C++, or carefully optimizing Python-based systems to hide its runtime overhead while preserving flexibility. Although some alternatives have been writing their ML libraries in Rust, we continue to use Python with carefully designed optimizations. This approach allows us to evolve faster, with more community contributions, and still maintain performance on par with alternatives.

LMCACHE now a community effort: One key reason LMCACHE rapidly evolved from a research prototype to a widely adopted industry framework is the active involvement of community contributors. About a year ago, LMCACHE supported only local CPU, local disk, and Redis backends integrated with vLLM on NVIDIA GPUs. Today, it supports eight more

storage backends (NFS, WEKA, GPU-Direct Storage, Mooncake Store, NIXL, S3, InfiniStore, and Valkey) across four processor types (NVIDIA, AMD, Ascend, and TPU), and two inference engines (vLLM and SGLang). All of these contributions were made by industry partners who actively upstreamed their code to stay aligned with ongoing development and avoid divergence from the latest LMCACHE updates.

10 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>.

11 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, Qizheng Zhang and Hussain Mohammad for insightful feedback. We also thank GMI cloud for providing us GPU clusters to run the experiments.

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. Long-bench: 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.
- 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.
- 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.
- Yichao “Peak” Ji. Context engineering for ai agents: Lessons from building manus, 7 2025. URL <https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus>. Blog post.
- 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*, 2025. URL <https://openreview.net/forum?id=W0y0tao61Q>.
- 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, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL <https://doi.org/10.1145/3600006.3613165>.
- 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>.

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. Cachegeen: Kv cache compression and streaming for fast large language model serving, 2024a. 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*, 2024b.

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.

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/osts/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>.

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. Sqlang: 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.