Building py-kvcache: A Performance Characterization of External KV Caching for vLLM with NVMe SSDs

Joseph Kanichai
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Tiziano De Matteis
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands

Animesh Trivedi
IBM Research Zurich
Zurich, Switzerland

Abstract

Prefix caching can reduce the time to first token (TTFT) of long-context LLM requests by reusing previously computed key-value (KV) states, but for short prefixes or fast GPUs, recomputation can be faster than loading from an external cache. We characterize this tradeoff in *vLLM* across GPU, CPU, and NVMe tiers using synthetic workloads, long-context benchmarks, production traces, and find that cache performance depends on transfer granularity, intermediate memory use, and when transfers enter the request schedule, not only on device bandwidth.

These findings motivate *py-kvcache*, a *vLLM KV Offload* connector with asynchronous direct I/O, bounded shared staging, and scheduler-aware preloading, which starts disk reads while requests are still waiting, overlapping with compute. At 80k tokens, *py-kvcache* loading from disk is 2.0× faster than *LMCache*, with preloading contributing 1.34×. With GPU, CPU, and disk caching enabled, it is 1.23× faster than *LMCache* and within approximately 4% of the native *vLLM KV Offload* implementation.

LongBench and *SCBench* show that these benefits extend to irregular prefix chains and multi-turn workloads. *Bailian* trace replays improve TTFT on a weaker GPU, but on an H100 the average request falls below the break-even point and GPU memory alone retains enough prefixes. External KV caching should therefore be treated as a setup specific admission decision.

The *py-kvcache* implementation is available at github.com/atlarge-research/py-kvcache.

This work is partially supported by Netherlands-funded projects NWO MLS (OCENW.KLEIN.561). This work used the Dutch national e-infrastructure with the support of the SURF Cooperative using grant no. EINF-15777. This work was carried out between January and July 2026, supervised by Tiziano De Matteis and Animesh Trivedi.

1 Introduction

Modern large language model (LLM) serving is increasingly a systems problem as much as a model problem. Transformer attention becomes increasingly expensive as context length grows, as during prefill, each token pays "attention" to all earlier tokens, making attention computation scale quadratically with prompt length. To avoid recomputing attention for previously processed tokens, each layer's key and value tensors can be stored in a key-value (KV) cache [33]. This cache can be essential for efficient decoding, but it grows linearly with context length, batch size, model width, and number of layers. As a result, long-context workloads can become limited by both quadratic prefill computation and KV cache capacity and data transfer rates. Recent work on *PagedAttention*, KV cache quantization, and KV cache offloading all treat KV memory as a serious bottleneck in LLM inference systems [13, 17, 21].

Prefix caching is a particularly attractive optimization because many real serving workloads contain repeated context. Examples include querying the same long document, multi-turn chat with a system prompt, retrieval-augmented generation (RAG) over repeated passages, and batch evaluation where many questions share a common prefix. In these cases, an inference engine can cache the KV blocks produced during prefill and reuse them for later requests, reducing time-to-first-token (TTFT) as well as the total request time. *vLLM* is an open source LLM serving engine. Its core abstraction, *PagedAttention*, manages the KV cache in blocks, allowing requests with variable length prompts and generations to share GPU memory efficiently while being scheduled together [13]. *vLLM* exposes repeated prefix reuse through automatic prefix caching and manages KV blocks on the GPU using *PagedAttention* [13, 34]. However, GPU memory is finite, and once the reusable working set exceeds video RAM (VRAM) capacity, the system must either evict useful KV blocks or move KV data to slower but larger tiers.

External KV caching extends prefix caching beyond GPU memory by storing KV blocks in CPU DRAM, local NVMe storage, shared filesystems, or even remote KV stores such

as Redis. Systems such as *LMCache*, *vLLM*'s own external caching mechanisms, and *llm-d* offloading components make the KV cache a distributed multi-tier storage platform rather than a simpler GPU-only data structure [7, 18, 19, 35]. This design is appealing because modern NVMe SSDs offer far better cost per gigabyte when compared to GPU VRAM or traditional DRAM. External caching is not automatically beneficial, since a cache hit replaces GPU prefill computation with cache lookup, disk or host memory reads, and a CPU $\leftrightarrow$ GPU transfer, not to mention scheduling, synchronization, etc. For short prompts or prompts with low cache hit rates, or when using fast GPUs, these costs can exceed the recomputation time that caching is meant to avoid. This creates a gap in existing systems: they provide mechanisms for storing and retrieving KV blocks from larger tiers, but pair them with a fixed policy that treats every prefix hit as a win. What is missing is a policy that decides when a lookup pays for itself, and writing one requires knowing where the dominant overheads arise and how scheduling decisions interact with I/O and transfer placement.

This work studies that tradeoff in *vLLM*, in various tiers and external KV caching systems. Our starting point is an implementation and measurement effort around the *vLLM KV Transfer API*, *LMCache*, and the newer *vLLM KV Offload API*. A significant part of the work is code path tracing, where we followed KV blocks as they move through *vLLM* scheduling logic, KV connector and transfer interfaces, CPU memory, GPU memory, and filesystem/NVMe storage to determine which operations lie on the TTFT critical path. The early experiments showed that API structure matters: blocking transfer paths, the granularity of KV chunks, and whether copies overlap with model forward passes can substantially change TTFT. These are not fixed properties that can be tuned once, since the same configuration wins or loses depending on prompt length, hardware, model, and how long a request waits before execution. Later experiments therefore shifted the focus from raw I/O bandwidth to request scheduling and critical-path placement. In particular, our notes show that reading KV data only after a request is scheduled leaves disk I/O on the TTFT critical path, while preloading data for waiting requests can leave only the final CPU-to-GPU transfer when the request begins service.

Based on these observations, we built *py-kvcache*, an external Python KV-cache backend designed for *vLLM* and filesystem/NVMe storage. The design emphasizes four principles. First, the cache should be simple to deploy and share across multiple *vLLM* instances by using an ordinary filesystem layout. Second, the I/O path should be efficient for the large, contiguous KV objects produced by block-based serving engines, avoiding excessive worker-thread coordination when a smaller number of outstanding large reads and writes is sufficient. Third, intermediate CPU memory should be explicitly bounded rather than growing with request size. Fourth, cache policy should be hardware and workload aware: external KV reuse should be bypassed below the measured break-even point instead of treating every prefix hit as a win.

The main question we ask is therefore not merely whether external KV caching can work, but *when it should be used*. Our experiments compare *LMCache*, *vLLM KV Offload*, and *py-kvcache* across synthetic long-document workloads and trace-derived workloads. The results preview a nuanced answer. For long repeated documents, *py-kvcache* with preload reduces TTFT substantially relative to *LMCache* and to *py-kvcache* without preload. In the tiered configuration at 80k-token document sizes, *py-kvcache* is 1.23$\times$ faster than *LMCache* and stays within 1.04$\times$ of the native *vLLM KV Offload* implementation. Preload is essential: in the disk-only configuration it halves TTFT relative to *LMCache* at 80k tokens, and against *py-kvcache* without preload it lowers TTFT by 1.34$\times$ at 80k tokens and by 1.66$\times$ at 40k tokens. However, *Bailian* trace experiments indicate that some real workloads have average request sizes below the break-even point for our setup, so external KV caching offers no TTFT benefit over GPU prefix caching alone despite cache hits.

This work makes the following contributions:

- We trace and characterize the performance behaviour of *vLLM*'s *KV Transfer API* and *KV Offload API* and related KV-cache systems, including how scheduling, copy granularity, and copy/compute overlap affect TTFT.
- We design *py-kvcache*, a Python external KV cache engine for *vLLM* for use with a shared filesystem, utilizing *io_uring*.
- We introduce and evaluate a preload mechanism that reads KV data for waiting requests before they enter execution, shifting disk I/O off the request critical path, and improving overlap of copy and compute.
- We show that external KV caching has hardware and workload dependent break-even points, motivating cache admission and bypass policies instead of unconditional loading and storing.

2 Background

This section provides the background needed to understand external KV caching, and how they work in *vLLM*. We first review how transformer inference uses the KV cache during prefill and decoding, why the cache becomes a major memory object for long-context workloads, and how prefix reuse changes the cost of repeated requests. We then describe why serving systems move KV data beyond GPU memory and discuss the *vLLM* interfaces that allow this.

2.1 KV Cache in LLM Serving

Autoregressive transformer models generate sequences of data, essentially predicting the next element based on all pre-

viously generated elements. Autoregressive transformer inference has two phases: *prefill* and *decode*. During prefill, the model processes the input prompt and computes a vector representation for each token at each layer, with each vector formed by attending to the tokens that came before it. For a prompt of length L, because each token is compared against all previous tokens, this work grows quadratically with L [33]. During decode, the model generates one new token at a time based on the previous prefill. Without additional state, each decode step would need to recompute attention over the entire prefix, making generation expensive for long prompts.

The KV cache avoids this recomputation. In each transformer layer, the attention mechanism projects hidden states into queries, keys, and values. Once a token has been processed, its key and value tensors can be stored and reused by later tokens. This reuse applies only to tokens in the prefix, since newly generated tokens depend on the model's previous outputs, so their KV tensors do not exist until generation reaches them. This also means that even though two prompts share identical text in the middle of a prompt, they cannot be interchanged unless the preceding text is also identical. A decode step then computes the query for the new token and attends over the cached keys and values from the prefix, rather than recomputing those keys and values from the original tokens.

The KV cache grows linearly with the number of cached tokens and is calculated like so: bytes per token = layers $\times$ 2 $\times$ KV heads $\times$ head dim $\times$ datatype size. As such, even a small model like Llama 3.2 3B, the bytes per token is 112 KiB, so a context of 80k tokens takes up 8.75 GiB. This can quickly overwhelm the limited GPU VRAM. It also creates a data movement problem since loading, storing, evicting, transferring KV tensors can directly affect request latency if those operations sit on the critical hot path for inference.

Most LLM serving systems manage the KV cache by placing groups of tokens in blocks rather than as one tensor per request. Block management makes it easier to allocate memory for variable length prompts and generations and evict or transfer cache regions at a useful granularity. *vLLM*'s *PagedAttention* is a popular implementation of this approach, using fixed size KV blocks to reduce fragmentation and support high throughput batched serving [13]. This block abstraction is the basis for the external cache systems studied in this work.

To identify reusable KV blocks, *vLLM* computes a hash for each full token block. The hash uses both the tokens in the current block and the hash of the preceding block, forming a chain from the start of the prompt. Two requests can therefore reuse a block only when their token sequences match from the beginning of the prompt through that block. Identical text appearing later in otherwise different prompts does not produce a reusable match. Cache lookup proceeds block by block until the first mismatch, and only the contiguous matching prefix is loaded. This makes prefix reuse and cache granularity dependent on the configured block size, since a partially matching block must still be recomputed.

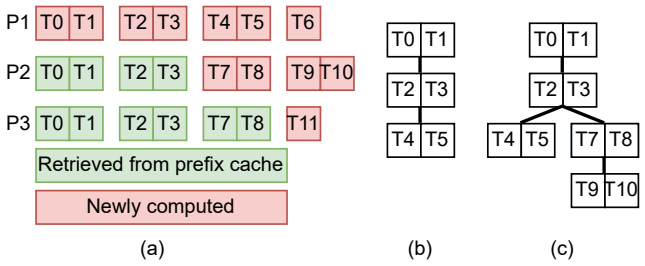


Figure 1: Prefix-hash chaining.

Figure 1 illustrates this process with two-token blocks. In panel (a), P1 has no reusable prefix and computes every token. Its three full blocks form the hash chain in panel (b), while the partial block containing T6 is not cached. P2 then reuses the first two blocks before branching to T7–T8 and T9–T10, producing the prefix tree in panel (c). P3 can follow that second branch through T7–T8 and needs to compute only its unmatched partial block, T11. Blocks before a branch can therefore share KV data, whereas blocks following different token sequences receive different chained hashes.

2.2 Extending KV Cache Beyond GPU Memory

GPU memory is the best place to keep KV blocks, since it offers the highest bandwidth, but it is also the smallest and most expensive tier. Once the active KV working set exceeds GPU memory, the serving engine must either evict blocks and recompute them later or move the KV cache to a larger but slower storage tier. External KV caching does just this, by storing on other media outside the GPU and loading back when a later request reuses the same prefix.

The closest external tier is CPU DRAM. DRAM has much larger capacity than GPU memory and can be shared by the host process, but using it still requires CPU $\leftrightarrow$ GPU transfers over PCIe. Because of the general design of all modern computer hardware, it is not possible to escape the PCIe link, and all secondary tiers for a KV cache must pass through the PCIe bus (excluding tightly coupled chips such as Nvidia's Grace Hopper platform). Each secondary tier used for storing KV cache adds copy, metadata, synchronization costs.

Using storage as a secondary tier provides far greater capacity thanks to the far lower cost per gigabyte offered by modern NVMe SSDs when compared to DRAM or GPU VRAM. Shared or distributed filesystems take this idea further allowing whole clusters and deployments to share their secondary cache, which should result in better cache hit rates.

LMCache adds external KV cache storage for *vLLM* and supports moving KV data across GPU, CPU, local storage, and remote or shared backends such as object stores like S3,

or Redis [7, 19]. *vLLM*'s *KV Offload API* provides a native interface for offloading KV blocks from GPU memory and overlapping transfer work with model execution [35], and offers its own DRAM and filesystem tier. Numerous other external KV cache stores exist such as *llm-d* [18] and Mooncake [24]. These systems make external KV caching practical, but they also motivate characterization of external KV caching, to understand tradeoffs and how they react to changes in setup and workload.

2.3 vLLM KV Cache Interfaces

vLLM exposes external KV caching through interfaces that let a connector load KV blocks for a request and store newly produced KV blocks for later reuse. The lower-level interface, called the *KV Transfer API*, separates cache management into scheduler and worker operations. The scheduler methods run in the *vLLM* scheduler process and operate on request metadata rather than tensor data, by computing token or block hashes, checking whether a prefix is present in the external cache, and deciding how many prefix tokens can be loaded. The worker methods run in the GPU worker process and perform the actual data movement, both loading matched KV tensors into GPU cache slots and storing newly generated KV tensors into the external tier. This split is at the core of how *vLLM* is designed, since the scheduler does not run GPU code, or have control over the GPU tensors.

The *Transfer* connector was synchronous until v0.9.0[1], so *vLLM* could block while a connector loaded or stored KV data. v0.9.0 added asynchronous support, but the API still requires the connector to manage request completion. In practice, this makes the *KV Transfer API* complex to implement, and requires intimate knowledge of data flow and system state, resulting in numerous methods that need to be implemented, all with minimal documentation. Most existing external KV cache systems like *LMCache* use the *Transfer* connector as shown in Figure 2b.

The *KV Offload API* introduced in v0.11.0 provides a smaller abstraction for the same general problem. Instead of requiring a connector to implement many scheduler and worker hooks, *Offload* exposes a few simpler operations: `prepare_load` and `prepare_store` on the scheduler side, and `transfer_async` on the worker side. Internally, *vLLM* implements this API as a wrapper over the *KV Transfer API*. An *offloading_connector* translates the simpler offload operations into the lower-level *Transfer* interface, as shown in Figure 2a. *Offload* allows implementors to only think about data movement and cache management, without worrying about async state management, request lifecycle and other similar actions required when implementing the *KV Transfer API*. The important systems difference is that *Offload* is designed around asynchronous movement of KV blocks. In particular, it defers store work from engine iteration N to iteration $N+1$, so GPU $\rightarrow$ CPU copies for the previous iteration can overlap with the model's forward pass for the next iteration. Figure 2c shows the execution sequence of *Offload*.

Unlike the *KV Transfer API*, *vLLM* provides its own implementation of connectors using *Offload*, by providing a DRAM tier, with a cache manager running either least recently used (LRU) or adaptive replacement cache (ARC) eviction for management. *vLLM* have also introduced "Secondary Tiers", one of which offloads to a filesystem, described further in subsection 3.3.

2.4 Metrics

The main latency metric in this work is time-to-first-token (TTFT), the time from when a request is submitted until the first output token is produced. TTFT is especially important for interactive workloads because it includes queueing, scheduling, prefill or cache loading, GPU transfer, and the first decode step. External KV caching primarily affects TTFT because a cache hit replaces some or all of prefill with cache lookup and KV movement. In benchmarks where multiple requests are sent at the same time, other requests can spend time waiting, changing the TTFT.

A system with high storage bandwidth can still perform poorly if cache reads are placed late in the request path, and a slower tier can be useful if its work is started early enough. We therefore report not only end-to-end TTFT, but where applicable we also analyze total benchmark runtime, cache lookup time, CPU $\leftrightarrow$ GPU transfer times, disk read/write time and model execution. We also analyze bandwidth of these operations in GB/s. These measurements let us identify whether an optimization reduces total work, moves work off the critical path, or merely shifts overhead from one component to another.

For cache behaviour, we report cache hit rate and break-even point. Cache hit rate measures what fraction of requests, tokens, or KV blocks can be served from an existing cache entry rather than recomputed. We also discuss a break-even point, which captures the minimum prefix size where loading cache KV data is faster than recomputing the same prefix on the GPU.

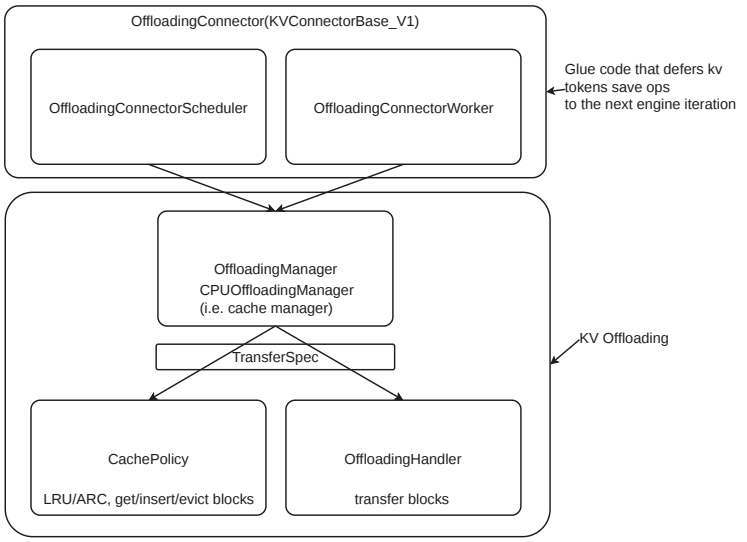
3 Systems Compared

We compare *py-kvcache*, our own external KV cache, against three existing systems. Table 1 summarizes their versions, *vLLM* interfaces, and storage designs, and the subsections below describe three existing systems, while *py-kvcache* is described in section 6.

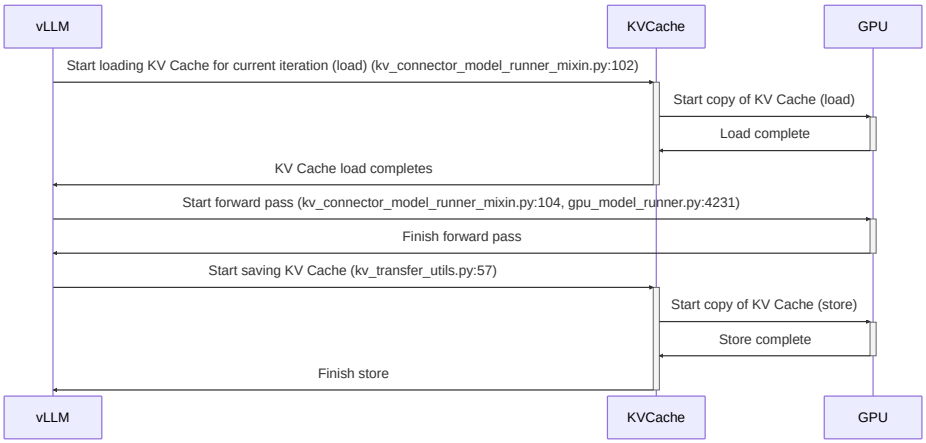
3.1 LMCache

For our experiments, we identified *LMCache* (v0.4.7) as the primary external KV cache with a disk tier [7, 19]. *LMCache*

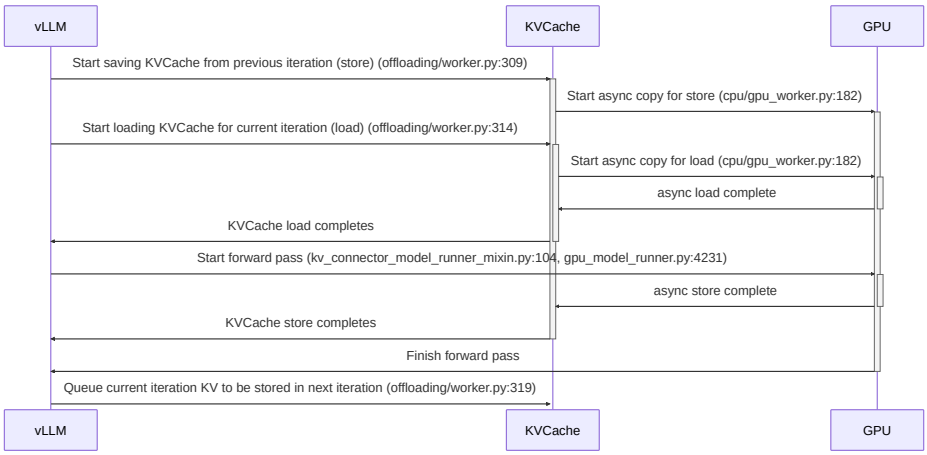
[1] https://github.com/vllm-project/vllm



(a) *KV Offload API* wrapper classes.



(b) Synchronous *KV Transfer API* execution sequence.



(c) *KV Offload API* execution sequence.

Figure 2: *vLLM* KV cache interfaces and execution sequences (v0.22.0). *Offload* defers stores to the next engine iteration so they overlap the following forward pass, while *Transfer* waits for them after each pass.

	LMCache	*llm-d*	Native *vLLM Offload*	*py-kvcache*
Version	v0.4.7	Our direct I/O fork, based on v0.8 #118c1ab	Our instrumented fork #d6eadf4	#790074a
Tested with *vLLM* version	v0.16, v0.22	v0.18, v0.19, v0.22	v0.16, v0.22	v0.22
vLLM KV Cache API	*KV Transfer API*	*KV Offload API*	*KV Offload API*	*KV Offload API*
Supported KV Cache tiers	CPU DRAM, disk, remote object stores, P2P	CPU DRAM (for staging), disk	CPU DRAM, disk (remote object stores and P2P only after v0.22)	CPU DRAM, disk
Disk I/O	POSIX, 4-thread pool	`O_DIRECT` in our fork, thread pool with read or write preferring workers	`O_DIRECT`, thread pool with read or write preferring workers	*io_uring*, single thread with bounded I/O depth
Staging memory	CPU tier's pinned pool doubles as staging	One pinned buffer per I/O thread	CPU tier doubles as staging	Shared with CPU tier
Filesystem layout	Flat directory, hashed filenames	Hierarchical, hash-addressed	Same scheme as *llm-d*	Same scheme as *llm-d*
Language	Python	C++	Python	Python

Table 1: Summary of the compared external KV cache systems. *py-kvcache* is this work, and is described in section 6.

is a popular external KV Cache, supporting multiple store types, GPUs, inference engines besides *vLLM*, distributed P2P deployments, integration with Kubernetes and much more. It supports being run as only a CPU DRAM KV Cache, and also with a secondary slower tier such as with disk or remote stores. For the disk tier, we used *LMCache*'s default disk tier (although they have recently introduced a native disk tier as well). The disk tier uses a small thread pool and standard POSIX file operations to write all cache files to the same directory, by hashing them with the parameters of the model and the prefix hash.

3.2 llm-d

For comparative purposes, we also picked *llm-d*, which provides a filesystem KV Cache [18].[2] However, this was deprecated while our experiments were ongoing, in favour of the secondary filesystem disk tier introduced directly into *vLLM*. That tier is functionally identical to what *llm-d* provided, keeping the same on-disk format and the same direct I/O thread-pool design, so it replaces *llm-d* rather than changing what is being measured. As a result, some of our Pareto fronts were conducted in *llm-d*. *llm-d*-kv-cache is a KV cache written in C++, that uses a staging CPU memory pool as an intermediate point for FS operations. It used a thread pool, where workers were reserved for read or write operations. *llm-d* uses standard POSIX file read and write operations, going through the page cache. However, for our tests this is unfavourable, and is likely to distort results, especially when cached data is re-read. To prevent this, we built a fork of *llm-d* with direct I/O support (based on v0.8.0-fs-v0.20, #118c1ab)[3]. *llm-d* also maintains a hierarchy of folders and subfolders based on model parameters and the prefix hash, with the aim of reducing filesystem contention by limiting the number of files in each directory.

3.3 Native vLLM Offload Implementation

vLLM's *KV Offload API* includes a native Python implementation with a CPU DRAM cache managed by an ARC or LRU eviction policy, and optional secondary tiers for filesystem, object stores or P2P transfer [35]. Multi-tier offloading was introduced in v0.22.0, and is orchestrated by the `TieringOffloadingManager`. The CPU tier acts as a gateway between secondary tiers and the GPU. On a store, *vLLM* first copies a completed GPU KV block to the CPU primary tier and then asynchronously propagates it to the secondary tiers. The CPU block remains protected from eviction until the filesystem write completes. On a lookup, *vLLM* checks the CPU tier first. A secondary tier hit is promoted into a reserved CPU slot, after which it can be copied to the GPU. While this promotion is in progress, the scheduler defers the request and retries in a later scheduling iteration rather than blocking on the operation.

The filesystem tier uses the same storage format as *llm-d*. It features two thread pools, giving reads and writes separate preferred priority, while allowing idle workers to process the other queue. This is functionally identical to *llm-d*'s staged memory and filesystem design, although the tier manager and cache policy are integrated into *vLLM*.

4 Experimental Setup

This section describes how we evaluate external KV caching systems and why each experiment was chosen. Our goal is

[2] https://github.com/llm-d/llm-d-kv-cache
[3] https://github.com/t348575/llm-d-kv-cache

not only to compare TTFT, but also to break down where the latency comes from. We therefore combine benchmarks with code tracing, to dive deeper into the causes. This work was carried out alongside upstream *vLLM* development, so the experiments span several *vLLM* versions rather than a single release. The evaluation of *py-kvcache* (and comparisons with Native *vLLM KV Offload API*, *LMCache*) in section 7 uses our *vLLM* fork based on v0.22. Most of the characterization in section 5 predates that fork. The interface comparison in Figure 4 and Figure 5 was measured on v0.16, the *llm-d* connector tracing on v0.18, and the break-even frontiers in Figure 6 and Figure 7 on v0.22. Table 1 lists the versions used for each system. Experiments use Llama 3.2 3B Instruct and Qwen3 4B Instruct 2507 [22, 27].

vLLM changed substantially over the course of this work, and the version spread follows that development rather than a choice on our part. Our experiments span v0.16 through v0.22. The *KV Offload API* was still new when we began, and was later rewritten to support multiple tiers, so we moved our fork forward as those changes landed. The same development also removed the need for *llm-d*. Its filesystem KV cache was deprecated in favour of the secondary filesystem tier built directly into *vLLM*, which is functionally identical to it, reusing the same on-disk format and disk I/O design. *llm-d* therefore appears in the characterization, where it was the filesystem cache available at the time, while the native *vLLM* filesystem tier takes its place as the architectural comparison point in section 7. Because the two occupy the same role and the connector has since been deprecated, we stopped testing *llm-d* rather than carrying it through the later experiments.

The experiments are designed to separate three questions:

- When cached KV reuse is faster than recomputation
- How different *vLLM* cache interfaces perform
- Where current KV caches leave performance unexploited

4.1 Benchmarks and Workloads

For evaluating external KV caches, we used both synthetic and trace driven workloads, representing the current state-of-the-art for testing LLM serving systems as well as workloads for specifically testing and benchmarking the performance of KV caches. These workloads were chosen based on their popularity, as well as their prominence in literature. The synthetic workloads give us maximum configurability, allowing us to control prefix length (how much of a prompt prefix is reused across requests), reuse rate (how many requests use the prefix), concurrency, among numerous other parameters. These synthetic workloads allow understanding the performance of KV caches under highly specific scenarios. Trace driven workloads on the other hand are meant to test whether the same conclusions and observations from running synthetic benchmarks hold under more realistic request patterns.

Measurement protocol

Document and prompt sizes are quoted in binary units throughout this work, so 1k denotes 1,024 tokens and 80k denotes 81,920 tokens. Each benchmark configuration was repeated three times, and we report the mean across repetitions unless otherwise stated. All tests were run with a single output token unless stated otherwise. Output tokens are generated by the LLM after prefill. We have set output tokens to 1 since any greater number would involve more GPU forward passes. This ensures that our performance results focus on the KV cache and not on long decode steps. All experiments were also run with FP16 KV data, to test the worst case scenario for KV cache size.

Long-document

The long-document benchmark is our main synthetic workload. Versions are included in the *vLLM* benchmark suite[4] and the *LMCache* benchmark suite[5] and are largely identical. We created our own version with additional control parameters. Some of the parameters supported by this benchmark are: document size, prefix reuse percent, prefix size, concurrency, arrival rate, warmup requests. Each run has a pre-warmup phase that sends a few small requests to *vLLM*. This is to warm up the code path, as well as to soak up any initialization costs. In our tests, we noticed a 2×–5× higher TTFT for the first request sent to *vLLM* after startup. After this, a warmup phase populates the cache and a query phase sends the requests that reuse it. In our experiments we sweep through a number of prefix reuse rates and document lengths. Our benchmark also supports providing a range of values for almost all parameters.

To evaluate scheduling and overlap, we use mixed workloads with both fresh requests and prefix reuse requests. These runs vary request concurrency, reuse fraction, and the gap between warmup and query traffic. They are used to test whether KV caches can move disk I/O out of the TTFT critical path and whether the benefit remains when the system is serving other requests at the same time.

Pareto capture

Because SSDs are the slowest form of memory in our external KV cache chain, it is important to understand when the cost of storing KV on an SSD is worthwhile. To do this, we have created a program that runs experiments using the long-document benchmark on a range of document sizes with KV cache disabled to understand pure GPU computation cost. Next, we re-run the same experiments while using our KV

[4] https://github.com/vllm-project/vllm/blob/d6eadf416bb5234047760bf55d532f2f038cf697/benchmarks/benchmark_long_document_qa_throughput.py

[5] https://github.com/LMCache/LMCache/blob/v0.4.7/benchmarks/long_doc_qa/long_doc_qa.py

cache. We can do the same for a pure CPU KV cache as well. With the experiment data, we can then interpolate TTFT times for values not explicitly measured (i.e. in-between values).

We define the terms used by the Pareto capture tool as follows:

$$\begin{aligned} D &= \text{total document size in tokens,} \\ P &= \text{cached prefix size in tokens,} \\ f(x) &= \text{cold TTFT (full compute),} \\ g(x) &= \text{cache hit TTFT.} \end{aligned}$$

Caching is worthwhile when:

$$g(P) + f(D-P) < f(D)$$

Calculating the Pareto frontier is essential, since it allows us to not only calculate a break-even Pareto front, but also allows us to calculate the minimum break-even data transfer bandwidth for prefix caching to be beneficial.

Filesystem metadata

External KV caches that use filesystems do not only depend on sequential read and write bandwidth. They also perform metadata operations such as checking whether a cache object exists, opening files, publishing newly written cache entries, evicting old entries. To isolate these costs, we built a minimal filesystem metadata microbenchmark that measures throughput and latency of various filesystem operations under various configurable parameters.

Each run prepares a populated workspace, performs an unmeasured warmup, and then runs the selected operation for a fixed duration across multiple worker threads. Payload-bearing operations use `O_DIRECT` to bypass buffered file-data caching, with optional file and directory `fsync()`. Some of the operations performed include `stat`, `access`, `open`/`close`, `create`/`unlink`, `rename`, `readdir`. These are measured both individually and as the transactions a cache connector actually performs, categorized as *lookup* or *publication* operations. This microbenchmark was deployed to understand if the standard metadata operations in a filesystem can turn into a bottleneck at scale.

ShareGPT

ShareGPT is a widely used collection of anonymized, real world ChatGPT conversations [2]. We use *vLLM*'s *ShareGPT* benchmark using `vllm bench`. The benchmark samples prompts and target response lengths from the dataset and replays them at a configurable request rate or concurrency. It therefore captures a realistic mix of short and medium requests, rather than a single fixed prompt length as in our synthetic workloads.

Unlike the long-document workload, *ShareGPT* does not intentionally construct a shared prefix across requests, and conversations are sampled independently and reuse arises incidentally. We use it as a general serving baseline and to quantify the overhead of enabling an external KV cache when hits are infrequent. The dataset has a low level of prefix reuse, and consists mostly of small conversations and prompt sizes. This distinction is important when interpreting the results, since *ShareGPT* measures behaviour under representative, mostly cold-cache traffic, whereas the controlled workloads isolate the potential benefit of prefix reuse. As a result, *ShareGPT* is only done to observe how external KV caching behaves in these scenarios, and not to establish any definitive performance baseline. As with our other experiments, output tokens is set to 1.

Bailian

Bailian is a set of anonymized traces created from samples of production traffic to a Qwen serving cluster on Alibaba Cloud *Bailian* [1, 36]. Similar to *ShareGPT*, this allows us to test realistic requests in size, prefix reuse and arrival. We replay the Coder, interactive (A), and API-driven (B) traces. Our replay driver reconstructs a synthetic prompt, similar to the *ShareGPT* benchmark.

SCBench

SCBench (Shared Context Bench) is a multi-turn benchmark to evaluate long-context scenarios, and is specifically designed for evaluating KV caching performance [15, 23]. Each session contains a context followed by a number of questions. The questions from each session share a prefix, making it possible to test how KV caches handle realistic multi-turn workloads. We do not directly use *SCBench*, but instead only use its dataset. We replay the dataset with our own driver which supports replaying the messages in round-robin format which allows evaluating a worst case scenario when the dataset is much larger than available GPU VRAM and CPU DRAM.

LongBench

LongBench is a long-context benchmark based on multiple-choice questions over long documents [5, 32]. Unlike *SCBench*, *LongBench* focuses on evaluating LLM output performance and quality. We therefore use *LongBench* only for its dataset and prefix reuse.

Benchmark Orchestration & Other scripts

We use a Python orchestration script to run all benchmark configurations consistently. The script starts *vLLM* with the selected KV cache configuration, executes the requested workload, and collects aggregate and individual request results. Benchmark and server parameters are defined in JSON configuration files, which can expand into parameter sweeps



Figure 3: Sample *py-kvcache* execution trace. Blue blocks are I/O reads and pink blocks are transfers to the GPU.

over specified configurations. This provides a common execution path for all systems. The benchmark orchestration script, driver code for all the benchmarks, along with all the other scripts and plotting for this work are available at github.com/atlarge-research/kvcache-experiments.

Tracing & Profiling

To understand how the two APIs work, as well as some of the internals of *vLLM*, we implemented simple-profiler[6], a profiling/tracing system in Python and a web viewer for analyzing the recorded traces. The tracing library records the start and end times of functions and events, together with supplementary data. Entire functions can be instrumented with `@profile`, while individual code blocks use `@profile_scope`. Asynchronous events that outlive a function call are timed manually and recorded with `add_event`. Traces are saved as JSON when the program exits and can then be loaded into the web viewer. Figure 3 shows an example subsection of a *py-kvcache* trace rendered by this viewer. Here `load_e2e` covers the cache work for one request, and `forward` only begins once it completes. The smaller blue blocks indicate I/O read operations, and the pink blocks are transfers to the GPU.

4.2 Hardware & Software Configuration

Our experiments were run across two machines. A local node equipped with less powerful GPUs, and GPU nodes on Snellius, the Dutch national supercomputer [30], used to represent more powerful systems. The local node was used for development, debugging, testing, and benchmarks, while the Snellius nodes were used only for benchmarks. Snellius nodes and our local node were allocated exclusively, so no other job shared the GPUs, the offload SSD, or the PCIe links during a measurement. Although the nodes have multiple GPUs, all our experiments use a single GPU. The two systems differ enough in GPU speed and VRAM capacity to fall on opposite sides of the measured break-even point, which section 7 uses to test when external caching stops being beneficial.

Component	Configuration details
Server	Supermicro SYS-221H-TNR
CPU	Intel(R) Xeon(R) Silver 4514Y
DRAM	256 GiB, DDR5
GPUs	2× Nvidia RTX 4000 Ada (PCIe 4.0), 20 GiB GDDR6
Offload SSD	Kioxia CM7-R 1.92 TB (PCIe 5.0), Samsung PM9A3 1.92 TB (PCIe 4.0)
OS & Kernel	Ubuntu 24.04 with kernel 6.8.0
Runtime stack	CUDA 13.2, Python 3.12, PyTorch 2.11

Table 2: Local node configuration.

Component	Configuration details
Server	ThinkSystem SD665-N V3
CPU	Dual socket AMD EPYC 9334
DRAM	768 GiB, DDR5
GPUs	4× Nvidia H100 (PCIe 5.0), 94 GiB HBM2e
Offload SSD	Samsung PM1743 7.5 TB (PCIe 5.0)
OS & Kernel	RHEL 9.6 with kernel 5.14.0
Runtime stack	CUDA 12.8, Python 3.13, PyTorch 2.11

Table 3: Snellius GPU node configuration.

[6] https://github.com/t348575/simple-profiler

Because of the limited software availability on the Snellius nodes, certain software versions are not identical when compared with our local node. The configuration of the local node can be seen in Table 2 and the Snellius node configuration in Table 3. The performance impact of these version differences is minimal for our purposes, since our measurements concern GPU copy time and the architecture of external KV caches.

The SSDs on the test systems are not identical, and have slightly differing write performance. However, since our workload primarily stresses the read performance, this is not a problem. Further, the sustained read performance we measured for the offload drive used on each system, the Kioxia CM7-R locally and the Samsung PM1743 on Snellius, is similar at 13.5 GB/s.

5 Characterization of Existing KV Cache Systems

To motivate and inform the design of our own external KV cache, *py-kvcache* (section 6), we characterize existing KV cache implementations to identify sources of latency and which parts of their design can be improved. A particular focus was placed on the hot path to reduce TTFT, as well as overlap between copy and compute. In these experiments, the terms "document length" and "prompt length" are used interchangeably to refer to the number of tokens in a request.

Unless otherwise specified, all experiments in this section were performed on the local node using Llama 3.2 3B.

Unless otherwise specified, *Offload* refers to the default *vLLM* implementation of the *KV Offload API* for a CPU cache. All experiments in this section were run with GPU prefix caching disabled, to isolate CPU and disk effects. These experiments were run while *vLLM* was actively changing both cache interfaces, so they span several *vLLM* versions.

5.1 GPU ↔ CPU Transfer Path

Our first experiment aims to understand whether the choice between the *KV Transfer API* and *KV Offload API* has an impact on TTFT, and whether this effect changes with document length. We compare *LMCache*, which uses the *KV Transfer API*, against *vLLM*'s default CPU cache implementation with the *KV Offload API*. We use the long-document benchmark over a range of document lengths, on *vLLM* v0.16. Each run first sends N unique prompts of length D to populate the cache, then repeats each prompt R times in random order, resulting in $N + (N \times R)$ requests. The first phase measures requests that compute and store KV data, while the second measures requests that load an existing prefix. All requests generate one output token so that the measurement is dominated by prefill or cache loading rather than decoding.

Figure 4 shows that both external caches substantially reduce cache-hit TTFT compared with recomputing the full prompt. The benefit grows with document length, from 2.2× at 1k tokens to 32.8× at 80k tokens, since the recomputation being replaced grows faster than the KV data being transferred. The two interfaces are far closer to each other than either is to the baseline. *Offload* is 1.03× faster at 1k tokens, 1.05× at 40k tokens and 1.04× at 80k tokens, while *LMCache* is 1.02× faster at 10k tokens. In absolute terms these gaps are at most 33 ms. On this benchmark the choice of interface therefore has little effect on cache-hit TTFT, because the same KV data must cross the same CPU → GPU path in both cases and that transfer dominates the measurement. This does not establish that the interfaces are equivalent in general, only that repeated long documents do not separate them. The next experiment therefore moves to shorter, mostly fresh traffic, where connector overhead is a larger fraction of TTFT.

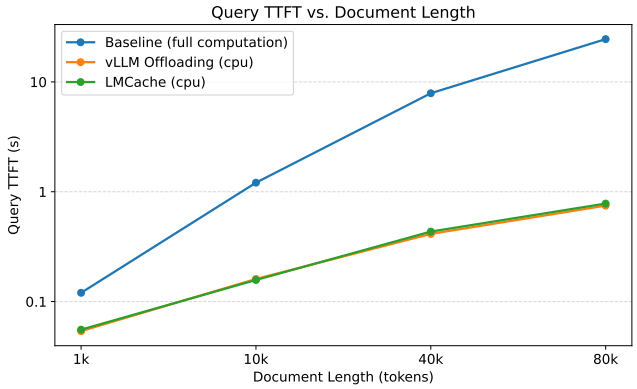


Figure 4: Long-document benchmark TTFT (*vLLM* v0.16, local node). Both external caches cut cache-hit TTFT by 2.2–32.8× relative to recomputation.

The first experiment uses repeated long prompts, where a large transfer dominates every cache hit. We therefore use ***ShareGPT*** to test whether the two interfaces also match under shorter, mostly fresh traffic. We replay more than 10,000 *ShareGPT* prompts at 32 requests/s using the same two cache implementations. These results are not plotted. *LMCache* has a TTFT of 206 ms, compared with 128 ms for the *Offload* implementation, a 1.61× difference. Because *ShareGPT* contains very little prefix reuse, this experiment does not measure the benefit of cache hits. Instead, it showcases connector overhead, and specifically focuses on the store path. This can affect TTFT even when most requests do not load a reusable prefix, and shows that the two interfaces do differ, on the store path rather than on the cache-hit path measured in Figure 4.

Tracing the GPU ↔ CPU Transfer Path

To identify where the two connectors differ, we instrument both cache paths and record cache loads, stores, GPU → CPU and GPU ← CPU copies, and model forward passes for a 40k prompt. The default *Offload* implementation uses asynchronous DMA copies between GPU and CPU memory. En-

queuing a copy through `transfer_async` has negligible cost in our traces, whereas the corresponding `wait_for_save` call in *LMCache* takes approximately 400 μs. *LMCache* also performs its copies using a CUDA kernel, which consumes GPU compute resources.

Offload spent 3.67 s in total on GPU $\leftrightarrow$ CPU copies across the benchmark and issued 21 transfers per warmup request, with each transfer taking an average of 9.08 ms. *LMCache* spent 4.13 s on the same stage, but divided each request into 640 transfers averaging 310 μs each. *Offload* transfer size follows *vLLM*'s `max_num_batched_tokens`, while *LMCache* divides the same KV data into smaller cache chunks. The larger *Offload* transfers are interleaved with subsequent forward passes, allowing much of the copy time to overlap with model execution. This experiment therefore identifies transfer granularity and execution overlap as possible causes of the TTFT difference, but the different copy mechanisms mean that the aggregate timings alone cannot determine whether either system has higher raw transfer bandwidth.

The transfer trace shows that *Offload* copies overlap with later forward passes, but not why this overlap occurs. To determine the cause, we trace successive *vLLM* engine iterations while varying `max_num_batched_tokens`. The scheduler greedily fills each execution batch up to this limit and submits the next iteration immediately after the current one. The *Offload* connector queues KV data produced in iteration N and performs the corresponding store work during iteration $N+1$, allowing the copy to overlap with a later forward pass. The *Transfer* path used by *LMCache* does not defer stores in the same way and instead waits for the store operations after each forward pass. This experiment confirms that the observed TTFT difference is partly a consequence of where store work is placed in the engine cycle, rather than only the cost of the copy itself.

To test whether the TTFT difference is caused by raw GPU $\leftrightarrow$ CPU copy bandwidth, we vary both document length and `max_num_batched_tokens` and calculate the effective bandwidth of the measured copies. This changes the size and number of transfers issued by the *KV Offload API* while keeping the underlying GPU and CPU unchanged.

Despite their different copy mechanisms and granularities, Figure 5 shows that both systems sustain approximately 24–26 GB/s of GPU $\leftrightarrow$ CPU bandwidth. Changing `max_num_batched_tokens` has little effect on bulk transfer bandwidth once transfers are sufficiently large. We therefore conclude that the TTFT advantage of *Offload* comes mainly from issuing fewer transfers and placing them asynchronously alongside model execution, rather than from moving data at a substantially higher raw bandwidth. This motivates retaining *Offload*'s asynchronous execution model in *py-kvcache* while reducing coordination and I/O overhead elsewhere in the cache path.

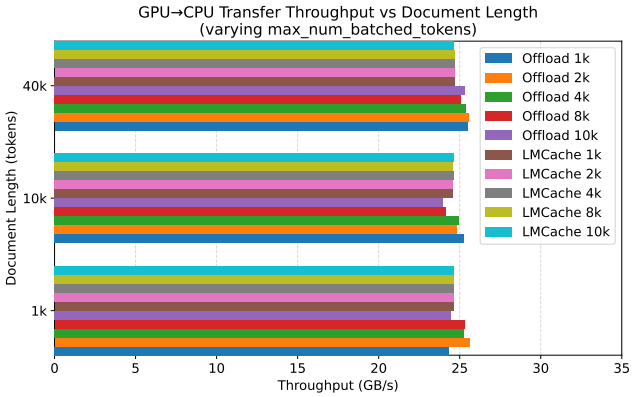


Figure 5: GPU $\leftrightarrow$ CPU transfer bandwidth for different `max_num_batched_tokens`. 1k–10k in the legend denotes the `max_num_batched_tokens` value (*vLLM* v0.16, local node).

5.2 NVMe SSD KV Caching

Having investigated the GPU $\leftrightarrow$ CPU transfer path, we next examine the additional costs introduced when KV data is stored on an NVMe SSD. These experiments focus on the conditions under which disk KV caching is useful and on whether the overhead comes from the storage device, the filesystem, or the cache connector.

Our first disk experiment identifies when loading a cached prefix is faster than recomputing it on the GPU. This is critical, since widely varying GPU and disk performance could swing results either way. Using the Pareto benchmark described in section 4.1, we measure cold TTFT (compute + store op) and cache-hit TTFT (load op) across a sweep of document sizes. We then interpolate between the measured points to obtain the boundary at which both choices have equal TTFT. We validate this boundary by running additional configurations immediately above and below it. The experiments were conducted with *llm-d* as the filesystem KV cache, and as described in subsection 3.2, the tests were run on our fork with direct I/O support to avoid page-cache effects. The frontiers discussed below were measured on the same v0.22 fork used in section 7.

The resulting Pareto frontiers in Figure 6 show that a cache hit is not sufficient for caching to be beneficial. The line in each subplot marks the minimum reusable prefix fraction at which the TTFT for a cache hit will match recomputation, and the background shades the TTFT speedup or slowdown predicted for the same path against recomputing the whole prompt, at each document size and prefix fraction. On our Snellius node, using Llama 3.2 3B with *llm-d*, an 8k prompt requires a 77.8% prefix reuse for breaking even, while at 80k tokens the break-even prefix falls to 7.8% of the prompt. The frontier shifts between GPUs, models, SSD. Faster GPUs recompute a prefix more quickly, whereas larger or slower

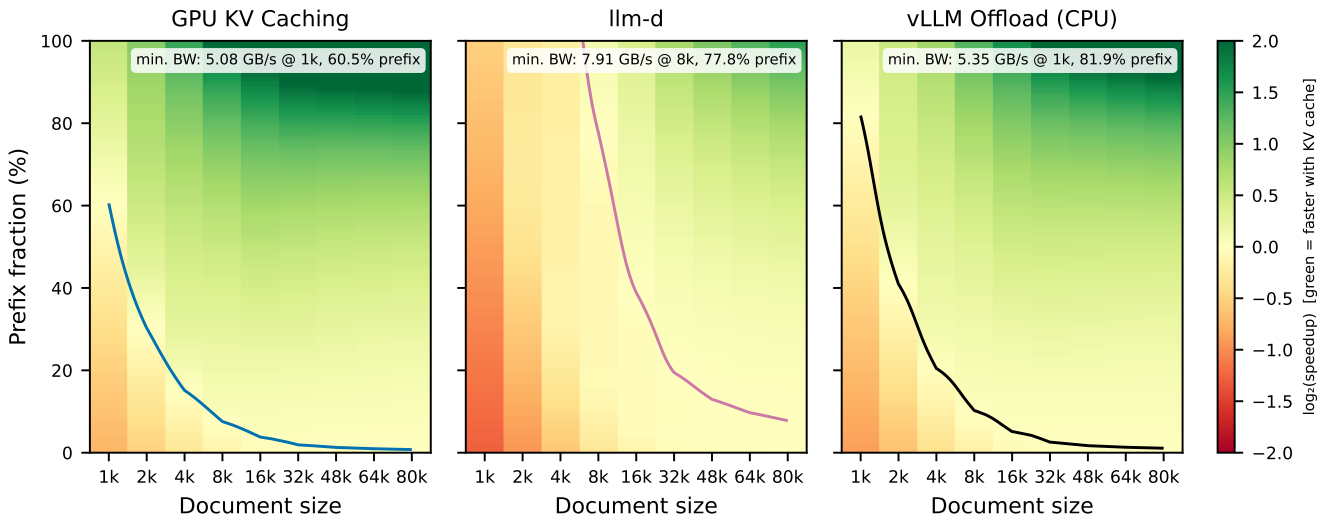


Figure 6: TTFT break-even frontiers (*vLLM* v0.22, Snellius node). Above each line, loading the cached prefix is faster than recomputing it.

models leave more time in which cached KV data can be loaded.

In Figure 7 we answer the question of how fast a storage platform (DRAM, SSD, etc.) must be for caching to beat recomputation. Based on section 4.1, for a document of D tokens we subtract the non-transfer part of the cache hit from the time needed to compute that document, which leaves the time budget available to move the KV data:

$$\text{max_io}(D) = f(D) - (g(D) - t_{\text{copy}}(D)),$$

where $g(D)$ is the cache-hit TTFT and $t_{\text{copy}}(D)$ is the time spent moving the KV data into the GPU, taken from the profiler traces described in section 4.1 as the median duration of the recorded transfer events for that path. In *llm-d* this is disk to GPU time, and covers the disk read as well as the GPU copy. Dividing the document's KV bytes by this budget gives the throughput a cache path must sustain for a fully cached prompt to match recomputing it. Llama 3.2 3B stores 114,688 bytes per token, so the requirement falls sharply as prompts grow, from 23.2 GB/s at 1k tokens to 10.4 GB/s at 8k and 3.5 GB/s at 80k, because prefill cost grows faster than the KV data that prefill produces.

The GPU $\leftrightarrow$ CPU copy sustains 54–56 GB/s and therefore clears the requirement at every document size. *llm-d* moves 9.0 GB/s at 1k tokens and 11.2 to 12.0 GB/s at larger sizes, which leaves it below the requirement at 1k and 2k but above it from 8k onward. The reference lines represent the ideal throughput of drives in our setup, as well as the required throughput for break-even. The practical consequence is that storage bandwidth sets a minimum document size. Below roughly 8k tokens the prefill is short enough that no storage device we measured can load the prompt faster than the GPU

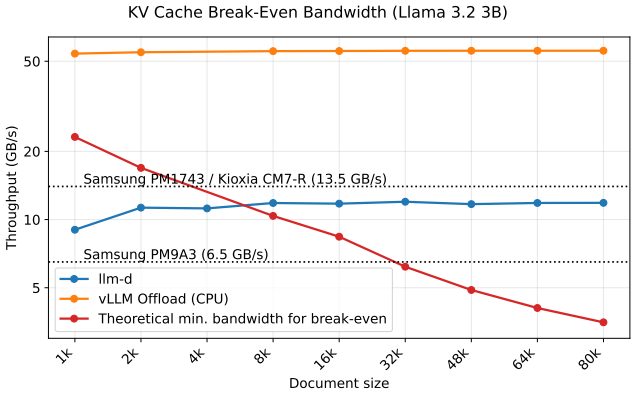
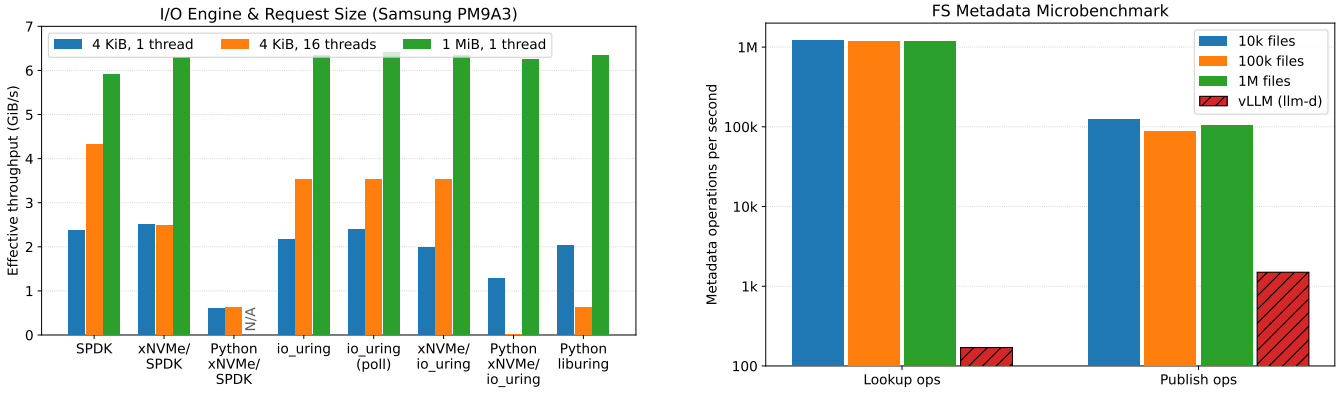


Figure 7: Break-even storage bandwidth (*vLLM* v0.22, Snellius node). This is a bandwidth representation of the Pareto Frontiers in Figure 6. The dashed lines show ideal measured throughput of our SSDs.

rebuilds it, and on a mid-range drive that threshold moves out to 32k. Therefore, external cache admission must depend on the model, GPU, SSD, and reusable prefix length rather than treating every hit as beneficial.

The Pareto model predicts the best TTFT attainable from the measured compute and transfer costs. Initial *llm-d* load measurements remained slower and more variable than this prediction even when disk reads approached the raw throughput measured by *fio* (v3.36) [3]. To identify the source of this gap, we instrumented the *llm-d* filesystem connector and traced its memory copies and block I/O requests.

The trace showed that *llm-d* copied and submitted data separately for each cache chunk. More importantly, its mini-



(a) I/O-engine throughput.

(b) Filesystem metadata throughput.

Figure 8: Storage microbenchmarks.

mum staging-buffer allocation was larger than the KV data required by our test model. A request containing 4.4 GB of useful KV data consequently caused approximately 10 GB to be copied and read or written. We reported this allocation problem, and a separate *vLLM* integration incompatibility, upstream as *llm-d* issues #454 and #389; the allocation bug was fixed in #589. After removing the minimum allocation, TTFT fell approximately 50% below *LMCache* in the same test. The connector's reads could reach more than 12 GB/s, but writes remained around 2 GB/s. Device bandwidth alone is therefore an incomplete predictor of cache performance: staging-buffer policy and data amplification can dominate the storage path. This motivates exact staging allocations and explicit bounds on how much intermediate memory an operation may reserve.

A separate native *vLLM* offloading issue observed during *SCBench* is discussed alongside its execution trace in Figure 12.

I/O engine and request size

We next test whether a more scalable asynchronous I/O engine or a larger worker pool improves SSD KV caching on our local node. As reference points, *fio* measured 13.5 GB/s on the Kioxia drive, and 6 GB/s from the Samsung PM9A3, while `nvbandwidth` measured 25–26 GB/s between CPU and GPU memory on our local node, connected via a 16× PCIe 4.0 link. Our Snellius node measured 54–56 GB/s over a 16× PCIe 5.0 link. In a standalone storage benchmark on Snellius, *io_uring* reached 338k operations/s with one thread, compared with 13k operations/s for the tested POSIX path, and reached 2.59 million operations/s with 16 threads. However, replacing the I/O path did not measurably improve TTFT in the LLM benchmark. Block tracing explains this result: the filesystem issued mostly 512 KiB requests, with some 1 MiB requests. As shown in Figure 8a the tested engines vary considerably on 4 KiB requests, but converge near the drive's bandwidth limit with 1 MiB requests and a single thread.

The figure compares *SPDK*'s userspace NVMe driver, native *io_uring* with and without submission-queue polling, the *xNVMe* wrappers over both backends, and the Python bindings for *xNVMe* and for *liburing*. With one thread at 4 KiB the engines land close together, but at 16 threads only the native engines gain throughput while the Python paths lose it, which we attribute to the Python runtime rather than to a property of the device. At 1 MiB the difference disappears and every engine clusters near the 6 GB/s that *fio* measured on this drive, except the Python *xNVMe SPDK* binding, which did not run at that request size.

The LLM workload is therefore limited by bandwidth and the placement of large transfers rather than small-I/O operation rate. High small-I/O IOPS and additional worker threads can improve a synthetic microbenchmark without improving end-to-end inference. This finding motivates using a small number of asynchronous workers with bounded queue depth, rather than relying on a large thread pool to obtain storage parallelism. Since KV cache files are much larger than 1 MiB, a Python implementation will reach the same storage throughput as a native one (section 6).

Filesystem metadata scalability

Because *py-kvcache* is intended to use an ordinary filesystem and to be shared by multiple *vLLM* instances, we test whether filesystem metadata becomes a bottleneck as the cache grows. We populate workspaces containing 10k, 100k, and one million files and separately measure the lookup and publication transactions defined in section 4.1. As summarized in Figure 8b, lookup throughput remains nearly constant at 1.21, 1.19, and 1.17 million operations/s, respectively. Publication operations achieve 122.7k, 87.4k, and 104.5k operations/s.

For comparison, the instrumented *vLLM* workloads issue only approximately 170 lookups/s and 1,500 publications/s. A standalone connector test with up to 64 instances likewise showed no filesystem-specific performance collapse before

CPU capacity became the limiting factor. We conclude that a hierarchical filesystem layout can support the metadata rate required by the evaluated serving workloads. This does not make individual metadata operations free, but it shows that the filesystem namespace itself is unlikely to be the throughput bottleneck at the tested scale.

Placement of disk reads

We finally use the request traces to determine whether disk I/O overlaps with useful work or remains on the cache-hit critical path. In the existing disk paths, a request waits for the disk → CPU transfer before the remaining CPU → GPU transfer can be performed. Thus, even when the SSD reaches its expected bandwidth, the read remains visible in TTFT if it begins only after the request enters execution. This observation motivates investigating whether disk reads for queued requests can begin earlier i.e., through preloading.

Characterization Takeaway

These experiments identify the requirements carried into the design of *py-kvcache*. The cache should avoid loading or storing prefixes below the measured break-even point, and allocate staging memory in proportion to the useful KV data. Its disk path should target large, bandwidth limited I/O with bounded concurrency rather than trying to achieve high IOPS with numerous threads and small requests. A regular hierarchical filesystem is sufficient for the observed metadata load, and disk reads should begin before request execution if possible so that storage latency does not remain entirely on the TTFT critical path. The next section describes how *py-kvcache* implements these requirements.

6 Design of py-kvcache

The characterization of existing KV cache systems identified various problems. Based on these findings, we built *py-kvcache*, a Python *vLLM KV Offload* connector for shared filesystem storage, built on the *KV Offload API* and its asynchronous GPU transfers. The design has four goals: to share cached prefixes across *vLLM* instances through an ordinary filesystem layout; to reach the bandwidth of current-generation NVMe devices without a large I/O thread pool; to bound the intermediate CPU memory any operation may reserve; and to avoid cache operations that are unlikely to improve latency.

6.1 Selecting the Storage Interface

The final I/O path was reached through several prototypes. We first investigated *xNVMe* because it provides a common interface to Linux I/O engines and the *SPDK* userspace NVMe driver. *SPDK* was attractive because bypassing the kernel storage stack could reduce per-operation overhead. However, the *xNVMe SPDK* backend uses only *SPDK*'s userspace NVMe driver, rather than its reactor, threading model, or application framework [40]. It also requires the device to be detached from the kernel NVMe driver and accessed by its PCI address and namespace. This did not fit our goal of allowing multiple cache instances to exchange hash-addressed objects through a mounted filesystem.

The prototype also exposed practical API problems. Our first *xNVMe* implementation aimed to use *io_uring*, but the *xNVMe* file API ignored this setting and executed the operations through POSIX I/O. After moving to the asynchronous *xNVMe* interface, each file operation required its own buffer and ring. The *SPDK* path additionally required the NVMe command interface, making it different from the filesystem backends, while the *xNVMe SPDK* integration did not expose all of the functionality needed by the prototype. Correcting the backend improved storage microbenchmarks (Figure 8a), but did not improve our *vLLM* TTFT. This result agreed with our prior characterization, since the workload consists of large, bandwidth limited reads and writes, so reducing overhead of I/O operations does not necessarily shorten the request's critical path. Further, this initial implementation was only able to maximize disk throughput in a traditional multithreaded worker design, similar to *llm-d*. In a single threaded design, the overhead of maintaining individual rings and separate polling queues for each file led to significant overhead for a single Python thread and as a result we could not utilize all of the available disk throughput.

We separately tested whether GPUDirect Storage could remove the intermediate CPU transfer. We implemented a simple prototype using KvikIO, which provides Python bindings to `cuFile` and can access GPU buffers through GPUDirect Storage [28]. In our configuration, this path was slower than all our other implementations.

py-kvcache therefore uses ordinary files with direct I/O and *io_uring* through the *liburing* Python library (v2026.3.30) [29]. *io_uring* provides asynchronous submission and completion queues shared between userspace and the kernel, and permits several requests to be submitted together [4]. This preserves our need for asynchronous I/O and bounded queue depth while also allowing the implementation to remain in Python, because at the request sizes used for KV blocks, we were easily able to maximize the throughput available on our test setups.

6.2 System Architecture

Figure 9 shows the resulting architecture. Solid arrows show request and KV-transfer execution, while dotted arrows show construction and fallback control paths. *py-kvcache* is loaded by *vLLM* as an `OffloadingSpec`, which constructs components in both the scheduler and worker processes. On

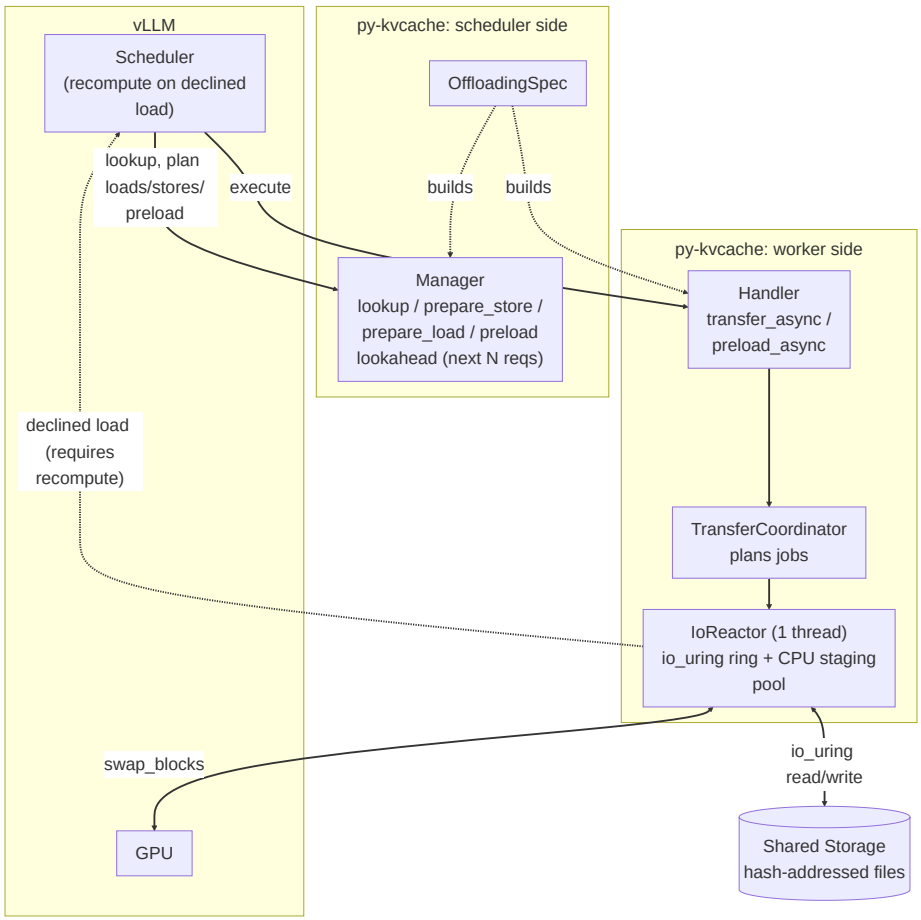


Figure 9: *py-kvcache* architecture.

the scheduler side, the `Manager` performs cache lookups and prepares load, store, and preload plans. It can decline a load, in which case *vLLM* recomputes the corresponding prefix, but it does not move KV data itself. On the worker side, the `Handler` receives the plan and invokes the `TransferCoordinator`. The coordinator divides the operation into file jobs, and a single `IoReactor` schedules them subject to the available staging memory and I/O depth. The reactor owns the *io_uring* queues, CPU staging pool, and CUDA streams, and uses *vLLM*'s batched block-copy operation to move data between the GPU KV tensors and staging slots.

This separation keeps scheduling decisions close to *vLLM*'s request state and storage execution close to the model worker. It also makes the control path independent of the data path: only compact transfer plans cross between the scheduler and worker, while KV tensors move directly between the worker's GPU, its staging pool, and shared storage.

6.3 Shared Filesystem Layout

To share cached prefixes across *vLLM* instances, *py-kvcache* adopts the hierarchical hash-addressed layout used by *llm-d*. A cache entry's path is derived from the model configuration and prefix-block hash, and intermediate directories distribute entries across the namespace. Our metadata characterization showed that this organization supports substantially more lookups and publications than the serving workload requires, including with one million populated files.

File data is accessed using direct I/O so that transfers do not depend on the state of the OS page cache. Direct I/O alone does not make an entry safe to share. Stores therefore write to a uniquely named temporary file in the target directory. After the write completes, *py-kvcache* creates the final hash-addressed name with a hard link and removes the temporary name. Link creation is atomic and fails if another instance has already published the same hash [31], giving concurrent writers a first-writer-wins protocol without replacing a valid entry. Readers only open the final name, so an entry is either absent or complete.

6.4 Asynchronous Transfers and Bounded Staging

py-kvcache stores KV data in blocks that group several of *vLLM*'s GPU blocks, with one file per block. With the 256-token block size used in our experiments, one Llama 3.2 3B cache block occupies approximately 28 MiB. Existing filesystem backends use pools of blocking worker threads to obtain I/O concurrency, which is not beneficial at this object size. *py-kvcache* instead uses one reactor thread to submit and reap multiple outstanding operations. The configured I/O depth, rather than the number of worker threads, controls storage concurrency.

By having a configurable software I/O depth, we can bound memory allocation. A single load or store covers the whole prefix of one request, which usually spans many storage blocks. *py-kvcache* therefore splits it into one job per storage block, each of which occupies one slot of a fixed CPU staging pool, and returns the slot once its I/O and GPU transfer complete. This minimizes CPU memory pressure and avoids deadlocks or similar issues that could arise when different scheduling is used, e.g., through deferral of requests. A request larger than the pool is processed incrementally rather than reserving memory for the entire KV object. Loads, stores, and preloads use the same pool, preventing each path from independently reserving its required capacity. The staging pool is one contiguous aligned, pinned CPU allocation. The CPU pool can optionally retain completed slots with LRU or ARC replacement, making it a DRAM cache in addition to a transfer buffer. By using the same CPU slots for disk I/O and DMA copies to our GPU, we avoid issues such as the write amplification observed in *llm-d* (subsection 5.2).

Transfers are pipelined per storage block (i.e. each file) rather than executing as a single operation. On a load, the reactor first issues asynchronous `openat` operations through *io_uring* up to a separate lookahead depth. A slot is acquired only when I/O depth is available and the read can be submitted. When a disk read is completed, its block mapping is immediately queued for CPU→GPU transfer. All load mappings that become ready during one reactor iteration, including mappings from different requests, are fused into a single `swap_blocks_batch` launch on a dedicated CUDA stream. While that DMA is in flight, the reactor can reap other disk completions and submit more reads. The staging slot is released only after its CUDA event reports completion. This pipelining can be observed in the trace in Figure 3. The pool reserves additional copy headroom beyond the configured I/O depth so that slots held by CUDA transfers cannot drain the disk pipeline.

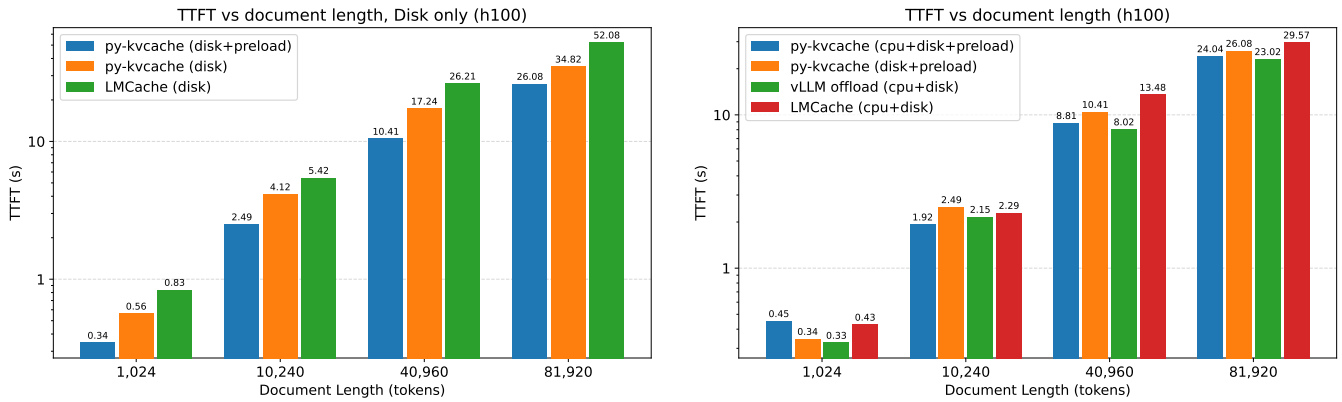
Stores use the reverse pipeline. Each staging slot has its own CUDA stream, which waits on that slot's CUDA event and copies one storage block from GPU memory without a device-wide synchronization. As soon as a slot's GPU→CPU copy completes, the reactor submits its I/O write. A store consumes one I/O-depth budget entry from copy launch until the write completion is reaped. The completed file is then published, and the slot is released or retained in the optional DRAM cache. Together with *Offload*'s deferred store semantics, this permits GPU copies and disk writes from one engine iteration to overlap work in later iterations, although this was never observed in our experiments because of our high performance NVMe drives, and was also implicitly limited by `max_num_batched_tokens`.

6.5 Preloading

Although asynchronous I/O reduces blocking inside the transfer engine, an ordinary demand read still begins after the request is selected for execution. *py-kvcache* therefore uses scheduler information to preload queued requests. The manager examines a bounded lookahead of waiting requests and sends preload plans to the worker when the storage path has available capacity. The reactor begins moving matching prefixes from disk into the existing CPU staging pool before those requests are scheduled. If a request later begins execution, it joins an in-flight read or claims the staged blocks, leaving only CPU → GPU promotion on its request path. This can lead to significant reductions in TTFT.

This required a small set of changes to our *vLLM* fork. The original *KV Offload API* only asks the worker to load KV data after the scheduler has selected a request, while the worker itself has no visibility into the waiting queue. We added a bounded `on_preload_candidates` callback to the scheduler and propagated the resulting preload identifiers and block metadata to the worker, where a new connector hook can start the read. This scheduler information stays within *vLLM*, and is handled in the translation layer between the *KV Transfer API* and the *KV Offload API*. The later demand load identifies and claims the same staged operation. Without this scheduler-worker path, *py-kvcache* could not begin disk I/O before request admission, and the full read would remain on the TTFT critical path.

Preloading work is deliberately subordinate to demand traffic. The reactor schedules ready demand read ops first, then new demand load and store ops, and only then speculative preloads. It reserves a number of staging slots equal to the configured I/O depth for foreground work, and a foreground load may reclaim either a retained cache slot or an unclaimed preload slot. If a demand load op appears after a speculative file has been opened or is in progress, the reactor closes that descriptor and requeues the preload instead of letting it consume read bandwidth. Identical prefixes requested by multiple preload candidates share one disk read and one reference-counted staging slot, while later demand loads join the in-flight operation rather than issuing duplicate reads. These rules were introduced after observing that concurrent speculative promotions in *vLLM*'s native secondary tier (subsection 3.3) could exhaust CPU memory, evict re-



(a) Disk-only query TTFT .

(b) Disk + DRAM query TTFT.

Figure 10: Isolated long-document evaluation (*vLLM* v0.22, Llama 3.2 3B, Snellius node).

cently promoted blocks, and force the scheduled request to recompute them, which we trace in Figure 12.

6.6 Break-even

Finally, *py-kvcache* does not treat every matched prefix as a useful load. The Pareto front procedure described in section 4.1 is run once per node and model as an offline calibration, and produces a small file holding the break-even prefix length for each source tier. Nothing is measured or fitted while serving. At lookup, the manager compares the reusable prefix with the threshold for its source tier. If loading is predicted to cost more than recomputation, the manager declines the load and lets *vLLM* recompute the prefix. The gate therefore protects against a loss rather than producing a gain. Below the break-even point the prompt is short enough that its TTFT is small in absolute terms, so declining the load avoids wasted transfer work without materially improving latency. Its value is that it keeps external caching from becoming a regression on workloads such as the *Bailian* traces.

7 Evaluation of py-kvcache

We evaluate whether *py-kvcache* improves end-to-end serving performance, which parts of its design provide observed improvements, and whether its behaviour remains useful under realistic long-context workloads. We first use the controlled long-document workload to isolate disk and preload behaviour, then use *LongBench*, *SCBench*, and the *Bailian* traces to evaluate the complete system. Experiments in this section were performed on our local node and on the Snellius node, with Llama 3.2 3B or Qwen3 4B. All measurements in this section use *vLLM* v0.22. *llm-d* is not evaluated here, since it was deprecated during this work in favour of *vLLM*'s native filesystem tier, which is functionally identical (subsection 3.2).

7.1 Isolated prefix cache tests

Disk-only comparison

We first ask whether *py-kvcache* improves cache-hit latency when all reusable KV data must be obtained from disk. We compare *LMCache*'s disk backend with *py-kvcache*, both with and without preload, using 50 concurrent requests on the Snellius node. GPU prefix caching and the break-even gate are disabled so that every matched prefix exercises the disk path.

Across the tested document sizes, the query TTFT results in Figure 10a show that *py-kvcache* without preload is consistently about 1.5× faster than *LMCache*, isolating the benefit of its asynchronous transfer engine. With preload enabled the total speedup over *LMCache* reaches 2.0–2.5×; subsection 7.2 separates the two contributions.

Tiered KV caches

We next compare complete "production-like" configurations with CPU DRAM, disk. As shown in Figure 10b, *py-kvcache* is 1.19×, 1.53×, and 1.23× faster than *LMCache* at 10k, 40k, and 80k tokens respectively. It stays within 1.10× and 1.04× of the native *vLLM* offloading implementation at 40k and 80k tokens, and is 1.12× faster than it at 10k. Relative to the same *py-kvcache* configuration without a CPU tier, adding DRAM improves TTFT by 1.30× at 10k tokens, 1.18× at 40k, and 1.08× at 80k. At 1k tokens all four configurations fall within 0.12 s of each other and the ordering reverses, which is expected below the break-even point established in subsection 5.2. Thus, *py-kvcache* approaches the integrated native *vLLM* implementation at the document sizes where external caching is worthwhile.

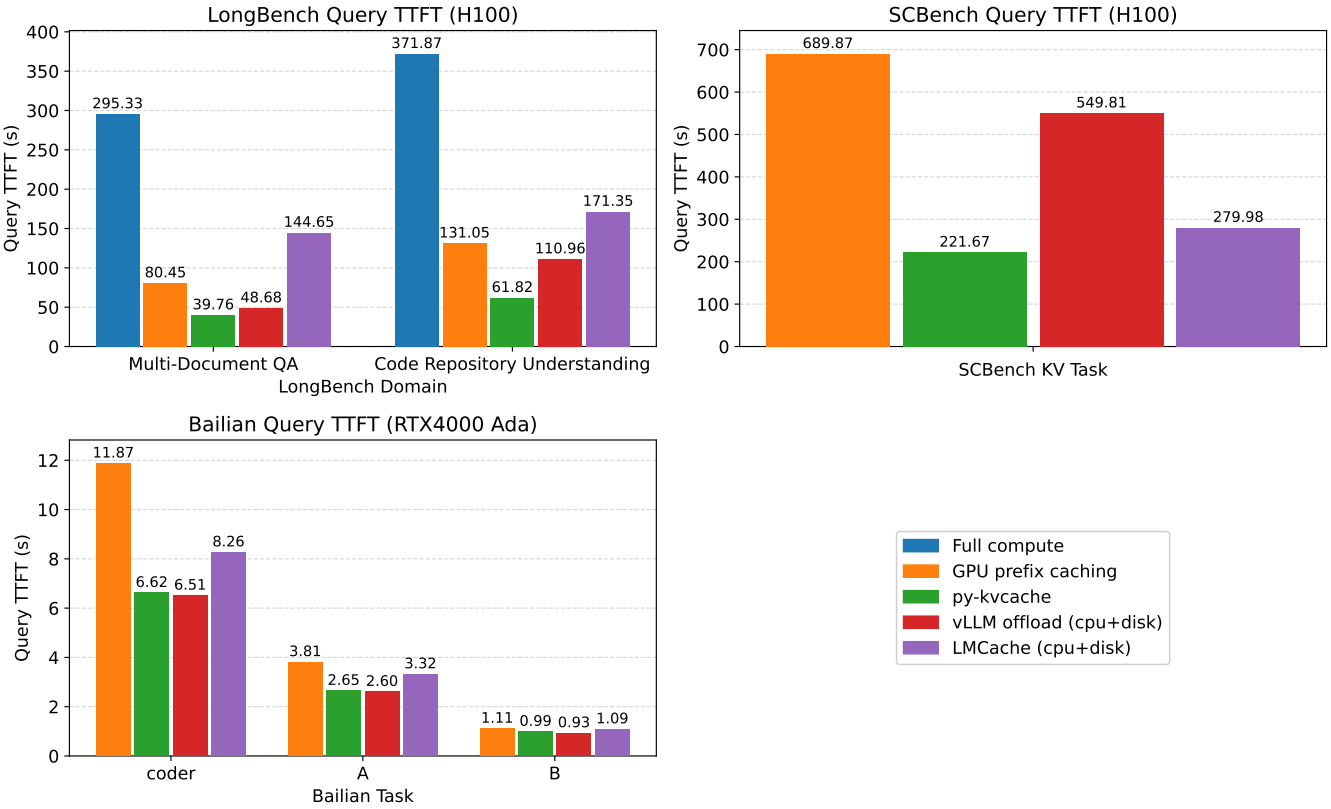


Figure 11: Query TTFT for *LongBench* and *SCBench* workloads on the Snellius node, and *Bailian* traces on the local node (*vLLM* v0.22, Qwen3 4B, Snellius node). *py-kvcache*, *LMCache*, and native *vLLM KV Offload* run with GPU prefix caching enabled.

7.2 Effect of Preloading

The preceding results compare complete systems, so we next isolate our preload mechanism using identical disk-only *py-kvcache* configurations with preload enabled and disabled. The query TTFT results in Figure 10a show that preload provides a 1.66× improvement at 40k tokens and a 1.34× improvement at 80k tokens. The smaller gain at 80k tokens is because the CPU → GPU transfer is significantly larger than at 40k tokens, and this time dominates the total TTFT, effectively "hiding" our preload gains.

Separately, we run a mixed workload containing 50 requests of 80k tokens, with 50% of requests reusing a cached prefix and a maximum concurrency of eight. For this complete benchmark run, preload reduces the query-round wall time by 6.8%, from 73 to 68 s. Within the same run, the per-request mean TTFT falls by 0.4 s across all requests and by 1.3 s for prefix-reuse requests alone. Varying the output length between 128 and 1k tokens produces little change in the preload benefit, as expected because the mechanism affects prefill rather than the subsequent decode phase.

Tracing explains where the improvement originates. Without preload, the first file read begins approximately 6 ms after cache lookup starts, after which the request must still wait for the disk → CPU transfer followed by a CPU → GPU transfer. Preload pays the same I/O startup cost earlier, while the request is waiting in the scheduler. When the request enters execution, some or all of the disk stage has already completed.

7.3 Representative Long-context Workloads

LongBench

The controlled workload always constructs reusable prefixes, so we next test whether the same advantage appears in a less regular request sequence. We replay the "Multi-Document QA" and "Code Repository Understanding" domains from *LongBench* on the Snellius node. As shown in Figure 11, *py-kvcache* achieves the lowest mean query TTFT in both domains: it is 6.02–7.43× faster than recomputation, 2.02–2.12× faster than GPU prefix caching, 2.77–3.64× faster than *LMCache*, and 1.22–1.79× faster than the native *vLLM KV Offload* implementation. Unlike the controlled benchmark, these workloads contain prefix chains of different lengths coupled with high concurrency. The result shows that preloading and *py-kvcache* as a whole is very effective in more realistic workloads where the reusable prefix distribution is not uniform. The performance uplift *py-kvcache* offers is largely due

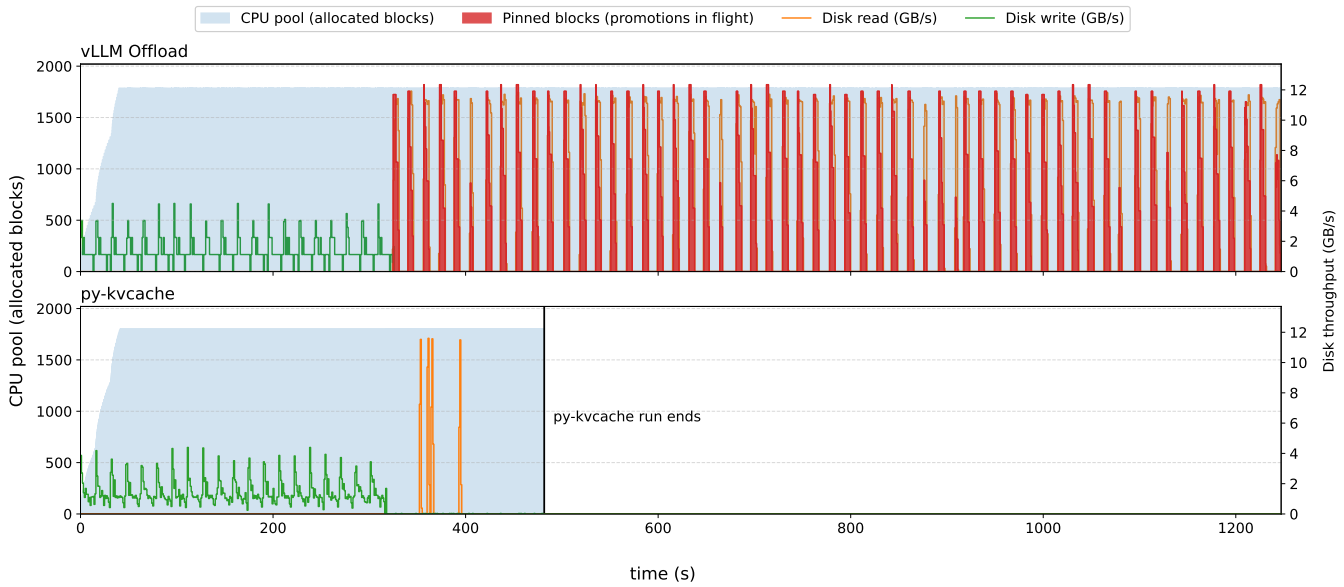


Figure 12: *SCBench* CPU-pool occupancy and storage throughput (*vLLM* v0.22, Qwen3 4B, Snellius node). Unbounded promotions keep the native CPU pool full and read 3.4 TB from disk, whereas *py-kvcache*'s bounded staging reads 85 GB and finishes serving all requests by 480 s.

to where the transfer is placed. Requests queue behind one another at this concurrency, and *py-kvcache* preloads from the scheduler's waiting list rather than on a cache lookup (subsection 6.5), so a reusable prefix can be read while its request is still waiting, leaving only the CPU → GPU promotion on the critical path.

SCBench

We use the *SCBench* KV workload to test a larger multi-turn working set in round-robin order. It is similar to our *LongBench* workloads, but represents a worst case scenario for KV caching. As shown in Figure 11, *py-kvcache* achieves a 3.11× speedup over GPU prefix caching, a 2.48× speedup over native CPU-and-disk offloading, and a 1.26× speedup over *LMCache* on the Snellius node.

The unexpectedly poor native-offload result led us to trace promotions, stores, demand loads, memory reservations, and evictions. Figure 12 compares native *vLLM* filesystem offloading in the top panel with *py-kvcache* in the bottom panel. The shaded area tracks CPU-pool occupancy, while the lines show disk read and write throughput. Native *vLLM* begins a promotion whenever a scheduler lookup finds KV data on disk. Several lookups can occur in quick succession, causing multiple promotions to reserve most of the CPU cache concurrently. Stores can then fail to obtain staging memory, and completed promotions may be evicted immediately to make room. When the request is eventually scheduled, its demand load may again find no free CPU memory and fall back to recomputing the prefix. The native trace repeatedly pins most of the CPU pool and continues beyond 1,200 s. In the traced run, only two requests ultimately transferred the promoted data from CPU to GPU, and a total of 3.4 TB of data was read from disk, while *py-kvcache* only read 85 GB, and the total cache size on each system was 465 GB. We reported this behaviour upstream as *vLLM* issue #49902. The issue has been acknowledged, and maintainers expect a policy or mechanism to detect back pressure, discussed in #50031 and #50014.

py-kvcache instead permits only one speculative preload at a time and stops issuing preload work when a demand load arrives. It also bounds the memory reserved by each load or store to the configured I/O depth. In this configuration, *py-kvcache* needs only 576 MB of free CPU memory even when the complete context is much larger. The lower panel of Figure 12 shows that this bounded staging avoids the repeated occupancy spikes and allows the run to finish after approximately 480 s. This experiment shows that moving reads earlier is not sufficient by itself, and that speculative work should be treated for what it is: speculative, and it must also be bounded and yield to demand operations.

7.4 Bailian Production Traces

Finally, we replay the Coder, interactive (A), and API-driven (B) traces from Alibaba Cloud *Bailian*. As shown in Figure 11, *py-kvcache* is 1.12–1.79× faster than GPU prefix caching and 1.10–1.25× faster than *LMCache* on the local node (RTX 4000 Ada). It remains close to the native *vLLM KV Offload* implementation, which is only 1.02–1.06× faster

across the three traces. This is largely due to the smaller prompt sizes and lower prefix reuse of this workload, compared to *LongBench* and *SCBench*.

The same traces show a different outcome on the Snellius node. Their average request length is below the measured 6,203 token SSD break-even point for Qwen3 4B, on our setup, and the H100's larger GPU memory retains a large portion of the working set. As a result the TTFT of only using GPU prefix caching is identical to that of *py-kvcache*, while the native *vLLM KV Offload* implementation is 0.2 s higher. Whether external caching is useful depends on both the workload's reusable-prefix distribution and the GPU's compute and memory capacity. It motivates *py-kvcache*'s model and hardware-specific break-even gate instead of loading every matched prefix.

7.5 Evaluation Takeaways

The evaluation shows that *py-kvcache* improves both the transfer path and the placement of disk reads. In the controlled disk-only benchmark, it halves TTFT relative to *LMCache* at 80k tokens. Preload provides a further 1.34–1.66× improvement over the same engine without lookahead. *LongBench* and *SCBench* show that these gains extend to chained and multi-turn contexts, while the *SCBench* trace demonstrates why bounded staging and demand prioritization are required under memory pressure. The *Bailian* traces show the boundary of these benefits, where external KV caching helps on the smaller GPU, but on the H100 many prefixes are below break-even and should remain in GPU memory or be recomputed.

8 Discussion

The results suggest that external KV caching should be understood as a critical path optimization, rather than only as an extension of the memory hierarchy. Peak storage bandwidth determines how quickly KV data can be moved, but it does not determine when that movement begins, how much intermediate memory it reserves, or whether loading is faster than recomputation. Across the evaluated systems, these scheduling and resource management decisions were often as important as the underlying storage medium. This distinction explains why a slower source can occasionally produce a lower TTFT.

8.1 Importance of the Critical Path

The storage microbenchmarks reinforce this interpretation. The tested I/O engines differed substantially for 4 KiB requests, especially when additional threads were used, but converged near the device bandwidth limit for 1 MiB requests with one thread, as shown in Figure 8a. The KV-cache workload generated mostly large requests, so increasing small-I/O operation rate or replacing the I/O engine did not by itself improve end-to-end TTFT. For this workload, the useful role of asynchronous I/O is therefore not to maximize IOPS. It is to maintain enough outstanding large operations to use the SSD bandwidth.

The GPU transfer results lead to a similar conclusion. *LMCache* and the native *KV Offload* implementation achieved comparable bulk GPU ↔ CPU bandwidth, but the *Offload* path issued fewer, larger transfers, crucially hiding store work alongside model forward passes. Its advantage therefore came from transfer granularity and overlap rather than a substantially faster physical copy path. *py-kvcache* retains this *Offload* execution model and applies the same principle to storage reads. Moving a read earlier can reduce visible latency even when the number of transferred bytes is unchanged. In the controlled experiments, preload improved TTFT by 1.34–1.66× over the same *py-kvcache* engine without preload. This is evidence that optimizing when work executes can be more valuable than optimizing the isolated speed of that work.

Preloading converts demand work into speculative work. It is beneficial only when the selected request is likely to execute and remains queued long enough for useful I/O to complete. An unlimited preloader could consume storage bandwidth, staging memory, and CPU ↔ GPU transfer capacity for requests that are delayed or never scheduled. The *SCBench* experiment demonstrates this risk, where several native promotions reserved most of the CPU tier concurrently, displaced recently promoted blocks, and left later demand operations without sufficient memory. *py-kvcache*'s policy of allowing one preload, sharing its buffers with demand transfers, and yielding when a demand load arrives is intentionally conservative. The result suggests that successful preloading requires admission control and resource bounds, not merely earlier submission. Unlike native *vLLM Offload*, which performs promotions based on lookups to KV blocks, *py-kvcache*'s access to the scheduler list allows it to perform smarter choices about which requests to preload.

8.2 Comparison with Existing Systems

LMCache supports multiple serving engines, storage backends, and distributed deployments, whereas *py-kvcache* is specialized for *vLLM*'s *KV Offload* and shared filesystem storage [7, 19]. In the evaluated configuration, *py-kvcache* benefits from *Offload*'s coarser, deferred transfers but this does not make *Offload* universally superior or replace *LMCache*'s broader functionality and extensive support for other storage media and distributed deployments. The native *vLLM Offload* implementation is the closest comparison because it uses the same API and integrated CPU and filesystem tiers [35]. *py-kvcache* achieves similar performance to *vLLM*'s native *Offload* implementation in our tiered cache tests and performs better on *LongBench* and *SCBench*. The *SCBench* result reflects *py-kvcache*'s more conservative preloading policy, and its more robust design when under demanding worst-case conditions.

py-kvcache and *llm-d* both use hierarchical, hash-addressed files for sharing and atomic publication [18]. *py-kvcache* uses transfer jobs bounded by I/O depth and a shared staging pool, instead of the separate buffers used by *llm-d*.

GPU prefix caching remains preferable when the working set fits in VRAM because it avoids any transfer or copy, since *vLLM* just swaps pointers.

8.3 When External Caching Should Be Used

The Pareto frontiers in Figure 6 show why a cache hit alone is an insufficient policy signal. External reuse is most attractive when a request has a long reusable prefix, the reusable working set exceeds GPU capacity, and the request waits long enough for transfers to overlap with other work. It becomes less attractive for short prefixes, low reuse, faster GPUs, or lightly loaded systems with little queueing time. The appropriate threshold also changes with model architecture, KV datatype, GPU compute rate, PCIe bandwidth, storage bandwidth, and source tier. Consequently, the numerical thresholds measured in this work are properties of specific configurations.

The *Bailian* traces illustrate both sides of this operating envelope. On the local node, external caching improved TTFT because the smaller RTX 4000 Ada retained less of the working set and recomputation was comparatively expensive. On the H100 system, the larger VRAM capacity retained more prefixes and the average request was below the measured SSD break-even point for Qwen3 4B. External transfers consequently offered little benefit even when reusable data existed. A practical multi-tier policy should therefore use CPU or SSD only when their predicted transfer cost is lower than recomputation, and preserve recomputation as a valid fallback rather than treating it as a cache failure or miss.

Break-even-aware decisions should also include the expected placement of work. A static threshold based only on prefix length and device bandwidth cannot distinguish a demand read from a preload that may be hidden by queueing. Conversely, a preload that appears profitable from transfer time alone may be wasteful when the request is unlikely to run soon. A stronger policy would combine prefix size, source tier, current staging capacity, measured transfer rates of each medium, expected queue time. The bounded policy evaluated here is a first step toward a more complex scheduler and cache controller.

9 Related Work

GPU-resident Prefix Reuse

PagedAttention makes non-contiguous KV blocks practical inside an LLM serving engine and underpins *vLLM*'s automatic prefix caching [13, 34]. SGLang's RadixAttention organizes reusable prefixes in a radix tree and integrates cache state with request scheduling [41]. These systems avoid external transfers when the reusable working set fits in GPU memory. *py-kvcache* addresses the complementary case in which reusable prefixes exceed that capacity.

External and Disaggregated KV Caches

LMCache provides reusable KV storage across serving engines and multiple local or remote backends [7, 19]. MemServe separates KV storage from inference workers through an elastic memory pool and includes a cost model based on request and cache state [9]. Mooncake similarly treats KV transfer as a central part of disaggregated serving and coordinates prefill, decode, and cache resources around a KV-centric data path [24]. DualPath extends this line by observing that the storage NICs attached to prefill engines saturate while those attached to decode engines stay idle, and adds a second load path that pulls KV state into decode engines and forwards it to prefill over the compute network, reporting up to 1.87× higher offline inference throughput on agentic workloads [39]. These systems target broader distributed deployments than *py-kvcache*.

SSD KV Caching

IMPRESS states the premise this work measures: when prefix KV state must be stored on disk, reusing it does not always reduce TTFT, because disk latency can exceed the prefill it saves [6]. It answers by loading less, restoring only the tokens it judges important from the similarity of their index sets across attention heads. *py-kvcache* answers the same observation by deciding whether to load at all rather than which parts to load. The two are compatible: selective loading moves fewer bytes and therefore shifts the boundary, but something must still decide when even the reduced transfer is not worth issuing.

Bidaw is the closest published relative of the break-even gate. It targets the same two-tier arrangement of host memory and SSDs evaluated here, and weighs storage footprint against computational saving when deciding what to retain [10]. Bidaw selects what to keep, while *py-kvcache* decides whether a load already matched in the cache is worth placing on the critical path against the cost of recomputing the same prefix. Tutti is concurrent work on the same tier, eliminating CPU intervention from the I/O control path through a GPU-centric object store and slack-aware scheduling [25]. Tutti is also built on *vLLM*, so the difference from *py-kvcache* is the I/O path rather than the serving engine: *py-kvcache* keeps the CPU-staged path and asks instead whether the transfer is worth issuing.

Several systems converge on transfer granularity as the dominant SSD-side concern, whether by consolidating KV pairs into larger blocks, adopting a single granularity for pruning and prefetching alike, or spreading co-activated entries

across multiple devices [37,42,43]. KV compression helps by increasing effective tier capacity, and shifting the measured break-even frontier [16, 17]. Where memory pressure is severe enough to make page-cache behaviour unpredictable, DUAL-BLADE abandons the filesystem altogether for an NVMe-direct path [12].

Prefetching and Lookahead

Prefetching recurs as the mechanism for moving KV transfers off the critical path before a request is scheduled. CachedAttention performs layer-wise preloading and asynchronous saving with scheduler-aware fetching and eviction [8]. Other systems also perform prefetching, by looking at the scheduler queue [38] or by speculating required KV entries [14]. HyMCache performs a similar action, but uses a CXL hybrid memory device to stage KV in device-side DRAM ahead of the foreground read [11]. *py-kvcache* takes the signal directly from the scheduler's waiting queue, so nothing is predicted.

10 Limitations and Future Work

10.1 Limitations

The experiments cover only two hardware classes, a small number of models, and a limited set of SSD and storage configurations. Notably, they exclude networked and RAID configurations. Our runs only use FP16 KV data and one output token in order to isolate prefill and cache movement. Although varying output length in the mixed workload did not appear to change much in our tests, the study does not establish effects on sustained decode throughput or multi GPU configurations. All experiments use a single KV block size of 256 tokens, so the results do not show how staging, file sizes and transfer granularity behave at other block sizes.

The trace workloads broaden the evaluation beyond fixed synthetic prefixes, but replay does not reproduce every aspect of a live production deployment.

Every measurement in this work uses a CPU dependent I/O path. Data is staged in host memory when transferring in/out of the GPU. Some research has explored direct GPU access to CPU memory or storage for both KV and the forward pass, and can dramatically change our conclusions [20,26]. A faster path does not remove the break-even between loading and recomputing, but it does move it, so the frontier we measure belongs to the path we evaluate.

Finally, *LMCache* and *vLLM* are rapidly evolving systems. The results describe the versions and configurations evaluated here. The comparative conclusions should therefore be read as evidence about design of such systems rather than as permanent rankings of projects.

10.2 Future Work

A dynamic serving engine could continuously estimate prefill time, tier bandwidth, queue delay, and promotion success, then update store, load, and defer decisions as the workload changes. Preload selection could similarly move beyond a single request, to a small priority queue governed by explicit memory and I/O budgets. The lookahead signal could also be pushed below the serving engine. A prefix-cache lookup already yields the ordered list of blocks that a request will consume, and recent work shows that a memory device can consume that ordering directly to stage data ahead of the foreground read [11]. The same signal could drive readahead in the storage layer rather than in the connector. Other useful directions include combining external caching with KV quantization or compression, testing remote and distributed filesystems, and evaluating multi node, multi GPU and small cluster configurations to test cache sharing under failures and contention. A broader evaluation should include tail TTFT, throughput, fairness, energy, and cost. Together, these extensions would turn the central result of this work into a general policy determining when and how KV data moves or whether it should move at all.

11 Conclusion

External KV caching is useful only when a reusable prefix can be moved sooner than it can be recomputed. Our characterization shows that this comparison depends not only on storage bandwidth but also on overlap between compute and copy operations. These findings led to *py-kvcache*, a *vLLM KV Offload* connector that uses asynchronous direct I/O, bounded shared staging, scheduler-aware preload, and setup-specific break-even decisions.

In the evaluated configurations, *py-kvcache* halves disk-only query TTFT relative to *LMCache* at 80k tokens and remains within 4% of native *vLLM* in the multi-tier comparison. *LongBench* and *SCBench* show benefits when reusable contexts exceed GPU capacity, while the *Bailian* traces on the H100 show the opposite boundary, where short prefixes and a large GPU leave little reason to use an external tier. External caching pays off when admission selects useful reuse and scheduling removes transfer work from the request's critical path.

Acknowledgment

We thank Zebin Ren and the AtLarge group at VU Amsterdam for their support.

References

[1] ALIBABA EDU. Qwen Bailian usage traces. https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon, 2025. Accessed 2026-07-21.

[2] ANON8231489123. ShareGPT Vicuna Unfiltered Dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2023. Accessed 2026-07-21.

[3] AXBOE, J. fio: Flexible i/o tester, version 3.36. https://github.com/axboe/fio/releases/tag/fio-3.36, 2023. Accessed 2026-07-21.

[4] AXBOE, J. io_uring(7): Linux manual page. https://man7.org/linux/man-pages/man7/io_uring.7.html, 2025. Accessed 2026-07-21.

[5] BAI, Y., TU, S., ZHANG, J., PENG, H., WANG, X., LV, X., CAO, S., XU, J., HOU, L., DONG, Y., TANG, J., AND LI, J. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks, 2024.

[6] CHEN, W., HE, S., QU, H., ZHANG, R., YANG, S., CHEN, P., ZHENG, Y., HUAI, B., AND CHEN, G. 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)* (2025), USENIX Association.

[7] CHENG, Y., LIU, Y., YAO, J., AN, Y., CHEN, X., FENG, S., HUANG, Y., SHEN, S., DU, K., AND JIANG, J. LMCache: An efficient KV cache layer for enterprise-scale LLM inference, 2025.

[8] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Cost-efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (2024), USENIX Association.

[9] HU, C., HUANG, H., HU, J., XU, J., CHEN, X., XIE, T., WANG, C., WANG, S., BAO, Y., SUN, N., AND SHAN, Y. MemServe: Context caching for disaggregated LLM serving with elastic memory pool, 2024.

[10] HU, S., ZHANG, G., ZHOU, Y., WEI, Y., ZHONG, Z., AND CHEN, J. Bidaw: Enhancing key-value caching for interactive LLM serving via bidirectional computation–storage awareness. In *24th USENIX Conference on File and Storage Technologies (FAST 26)* (2026), USENIX Association.

[11] JANG, H., SONG, I., NOH, S. H., AND KIM, J. HyMCache: A KV cache framework for multi-turn LLM serving with CXL-hybrid memory, 2026.

[12] JEONG, B., BYUN, H., KIM, Y., YU, W., LEE, K., YANG, J., AND PARK, S. DUAL-BLADE: Dual-path NVMe-direct KV-cache offloading for edge LLM inference, 2026.

[13] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J. E., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles* (2023), pp. 611–626.

[14] LEE, W., LEE, J., SEO, J., AND SIM, J. 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)* (2024), USENIX Association.

[15] LI, Y., JIANG, H., WU, Q., LUO, X., AHN, S., ZHANG, C., ABDI, A. H., LI, D., GAO, J., YANG, Y., AND QIU, L. SCBench: A KV cache-centric analysis of long-context methods, 2024.

[16] LIU, Y., LI, H., CHENG, Y., RAY, S., HUANG, Y., ZHANG, Q., DU, K., YAO, J., LU, S., ANANTHANARAYANAN, G., MAIRE, M., HOFFMANN, H., HOLTZMAN, A., AND JIANG, J. CacheGen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference* (2024).

[17] LIU, Z., YUAN, J., JIN, H., ZHONG, S., XU, Z., BRAVERMAN, V., CHEN, B., AND HU, X. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Proceedings of the 41st International Conference on Machine Learning* (2024).

[18] LLM-D PROJECT. Native KV cache offloading to any filesystem with llm-d. https://llm-d.ai/blog/native-kv-cache-offloading-to-any-file-system-with-llm-d, 2026. Accessed 2026-06-29.

[19] LMCACHE PROJECT. Architecture overview. https://docs.lmcache.ai/developer_guide/architecture.html, 2026. Accessed 2026-06-29.

[20] LUO, S., AND SHEN, H. No buffer, no bottleneck: Efficient zero-copy KV cache offloading for long-context LLMs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)* (2026), USENIX Association.

[21] MENG, W., LEE, B., AND WANG, H. Understanding bottlenecks for efficiently serving LLM inference with KV offloading, 2025.

[22] META. Llama 3.2 3B Instruct. https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct, 2024. Accessed 2026-07-21.

[23] MICROSOFT. SCBench dataset. https://huggingface.co/datasets/microsoft/SCBench, 2024. Accessed 2026-07-21.

[24] QIN, R., LI, Z., HE, W., CUI, J., TANG, H., REN, F., MA, T., CAI, S., ZHANG, Y., ZHANG, M., WU, Y., ZHENG, W., AND XU, X. Mooncake: A KVCache-centric disaggregated architecture for LLM serving. *ACM Transactions on Storage* (2025).

[25] QIU, S., HU, Y., WANG, X., ZHU, W., YAN, J., CHEN, H., XU, K., CHEN, K., AND ZHANG, Y. Tutti: Making SSD-backed KV cache practical for long-context LLM serving, 2026.

[26] QURESHI, Z., MAILTHODY, V. S., GELADO, I., MIN, S., MASOOD, A., PARK, J., XIONG, J., NEWBURN, C. J., VAINBRAND, D., CHUNG, I.-H., GARLAND, M., DALLY, W., AND HWU, W.-M. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (2023).

[27] QWEN TEAM. Qwen3-4B-Instruct-2507. https://huggingface.co/Qwen/Qwen3-4B-Instruct-2507, 2025. Accessed 2026-07-21.

[28] RAPIDS PROJECT. KvikIO python documentation. https://docs.rapids.ai/api/kvikio/stable/, 2026. Accessed 2026-07-21.

[29] RITESH. liburing python library, version 2026.3.30. https://github.com/YoSTEALTH/Liburing/releases/tag/2026.3.30, 2026. Accessed 2026-07-21.

[30] SURF. Snellius: The national supercomputer. https://www.surf.nl/en/services/compute/snellius-the-national-supercomputer, 2026. Accessed 2026-07-21.

[31] THE OPEN GROUP. link, linkat: link one file to another file relative to two directory file descriptors. https://pubs.opengroup.org/onlinepubs/9799919799/functions/link.html, 2024. The Open Group Base Specifications Issue 8, IEEE Std 1003.1-2024. Accessed 2026-09-09.

[32] THUDM. LongBench v2 dataset and benchmark. https://github.com/THUDM/LongBench, 2024. Accessed 2026-07-21.

[33] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017).

[34] VLLM PROJECT. Automatic prefix caching. https://docs.vllm.ai/en/latest/design/prefix_caching/, 2026. Accessed 2026-06-29.

[35] VLLM PROJECT. KV offloading usage guide. https://docs.vllm.ai/en/latest/features/kv_offloading_usage/, 2026. Accessed 2026-06-29.

[36] WANG, J., HAN, J., WEI, X., SHEN, S., ZHANG, D., FANG, C., CHEN, R., YU, W., AND CHEN, H. KVCache cache in the wild: Characterizing and optimizing KVCache cache at a large cloud provider. In *2025 USENIX Annual Technical Conference* (2025).

[37] WANG, T., CHU, L., FAN, R., AND REN, J. Swarm: Co-activation aware KVCache offloading across multiple SSDs, 2026.

[38] WANG, W., HOU, X., TANG, P., ZHOU, H., WANG, J., WANG, X., LI, C., AND GUO, M. PCR: A prefetch-enhanced cache reuse system for low-latency RAG serving, 2026.

[39] WU, Y., CHEN, S., ZHONG, Y., HUANG, R., TAN, Y., ZHANG, W., ZHANG, L., ZHOU, S., LIU, Y., ZHOU, S., ZHANG, M., JIN, X., AND HUANG, P. DualPath: Breaking the storage bandwidth bottleneck in agentic LLM inference, 2026.

[40] XNVME PROJECT. SPDK backend. https://xnvme.io/background/backends/spdk/index.html, 2026. Accessed 2026-07-21.

[41] ZHENG, L., YIN, L., XIE, Z., SUN, C., HUANG, J., YU, C. H., CAO, S., KOZYRAKIS, C., STOICA, I., GONZALEZ, J. E., BARRETT, C., AND SHENG, Y. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems* (2024), vol. 37.

[42] ZHENG, X., WEI, D., GAO, J., SONG, Y., MI, Z., AND CHEN, H. SolidAttention: Low-latency SSD-based serving on memory-constrained PCs. In *24th USENIX Conference on File and Storage Technologies (FAST 26)* (2026), USENIX Association.

[43] ZOU, J., WU, S., DUAN, H., LI, Q., AND XUE, C. J. ContiguousKV: Accelerating LLM prefill with granularity-aligned KV cache management, 2026.

Notes: IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates. Other products and service names might be trademarks of IBM or other companies.