Research project - LLM KV Cache Performance Characterization When Using Disk Offloading for Prefix Caching

Duc Anh Phan

September 2025

1 Abstract

Large Language Model (LLM) inference workloads are becoming increasingly popular in cloud, HPC, and data centers as they are widely adopted across different sectors such as education [10], finance [5], healthcare [1]. This widespread use leads to inference systems handling up to 2.5 billion of prompts per day [7], requiring a large infrastructure with high-end hardware systems. To meet the increasing computing demands, various techniques are used that optimize the performance of inference systems. Center to these is KV-cache, a structure to store keys and values in the attention layer for all input and previously generated tokens. KV cache allows the model, during autoregressive generation, to reuse these past keys and values instead of recomputing them at every decoding step, making inference faster. However, as the KV cache grows because of increased batching size or multi-turn conversations, it could exceed the capacity of GPU memory, prompting the use of CPU memory and disks as part of the offloading hierarchy [6, 8, 3]. Among KV caching uses, prefix caching is a popular optimization in LLM inference to avoid redundant prompt computations, and are the primary focus of this study. The core idea is simple, we save the KV-cache blocks of processed requests, and reuse these blocks when a new request comes in with the same prefix as the previous requests. Existing studies mainly focus on the performance of offloading the KV cache to CPU memory, while the performance of KV prefix cache offloading to disks remains understudied. In this work, we address this research gap and characterize the performance of vLLM inference engine when using disk-based KV cache backend (LMCache) and compare it against the baselines of keeping the KV cache entirely on the GPU or offloading it to CPU memory. The results show that prefix caching brings performance gain even when cache blocks are served from CPU memory or SSDs at different conditions. In addition, storage and system configuration choices such as file system types or chunk sizes are important to achieve the optimal performance. The artifact for this project is at **https://github.com/anhphantq/kv-cache-offloading-benchmarking**.

2 Background and Related Work

2.1 Transformer and KV cache

Large Language Models (LLMs) such as GPT, LLaMA, and Gemini are transforming the way we generate and interact with text, code, images and audio. These models are built upon a neural network architecture known as the decoder-only transformer (Figure 1). Input tokens are first mapped into continuous vector representations through an embedding layer and enriched with positional encoding to capture sequence order. These representations are then passed through a stack of identical transformer blocks, each consisting of layer normalization, multi-head self-attention, and feed-forward layers with residual connections [9]. The self-attention mechanism allows the model to relate each token to all previously generated tokens, making it well-suited for autoregressive text generation. After passing through multiple layers, the final hidden states are projected through a linear layer and normalized before being converted into output probabilities via a softmax function. This process enables the model to predict the next token in the sequence, step by step, until complete text is generated.

The decoder works in an auto-regressive and causal fashion (i.e., the attention of a token only depends on its preceding tokens), at each generation step we are recalculating the same previous token attention, when the goal is only to compute the attention for the new token. This is where KV makes benefits. By caching the previous keys and values, we just need to calculate the attention for the new token. This reduces the computation, making inference feasible for large-scale models. Without KV caching, generating long sequences would make the cost scale quadratically with the input length, leading to higher latency and more computation. In practice, the KV cache is what

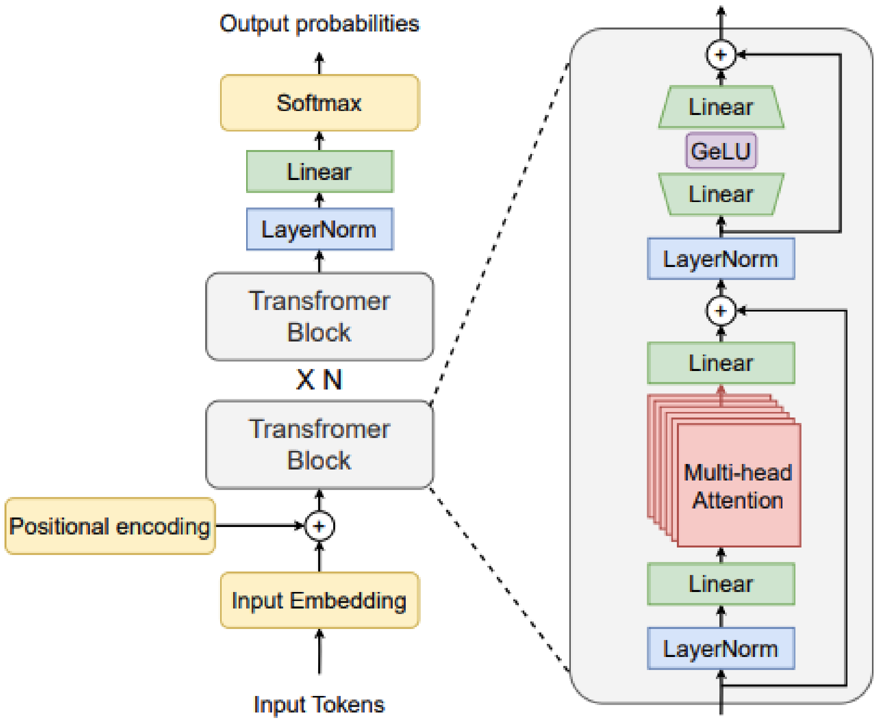


Figure 1: Decoder-only transformer [2]

enables LLMs to handle long contexts and serve real-time applications efficiently. However, it also introduces a new challenge, in which its size grows linearly with sequence length and batch size, often surpassing the GPU memory capacity and necessitating offloading strategies.

2.2 Existing serving architecture and optimizations

2.2.1 vLLM

vLLM is an open-source framework for memory-efficient inference and serving of large language models [4]. It introduces PagedAttention, which manages key-value (KV) caches like virtual memory pages, utilizing GPU memory efficiently. vLLM can dynamically batch requests of varying lengths, maximizing GPU utilization and throughput. Prompt length refers to the number of input tokens.

2.2.2 Prefix Caching

Prefix caching is a technique that reduces computation by reusing key–value vectors for tokens. Instead of recomputing these tokens for each request, their KV blocks are computed once and stored in paged memory [4], enabling subsequent prompts with the same prefix to directly use the cached results. This approach lower the overhead of prefill phase, though it provides no benefit during the decoding phase. As a result, prefix caching also helps lower TTFT (time-to-first-token, time from request submission until the first output token is received), which is an important metric, since the model can skip recomputing the shared prefix and begin generating output sooner.

2.3 Continuous Batching and Memory Pressure in vLLM

In static batching, a batch of requests starts at the same time and waits for all the requests in that batch to finish. The problem with batching is that the system does not know how many tokens would be generated, resulting in requests waiting for unfinished ones. This makes the GPU idle and leads to under-utilization. Continuous batching is introduced to overcome this disadvantage of static batching. Instead of waiting for all the requests in a batch to start a new one, it constantly schedules a new request when a request in the batch finishes, leaving no empty slot.

However, continuous batching introduces dynamic memory pressure on the system. Each incoming request consumes a number of KV blocks depending on its current sequence length and generation progress, causing the memory footprint of active requests to fluctuate over time. As new requests join the batch, they continually demand additional KV memory for their newly generated tokens. If the number of concurrent requests increases, the cumulative GPU memory usage can grow rapidly, leading to sudden spikes in memory pressure and potential contention for limited GPU resources.

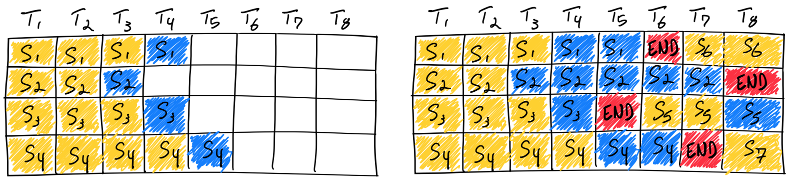


Figure 2: In continuous batching, a next request will be scheduled as soon as a request finishes its iteration [?]

When GPU memory is not enough to allocate additional KV cache blocks, vLLM employs an all-or-nothing eviction policy to maintain forward progress. Under this policy, a request is preempted, mostly the one that was most recently processed or has lower scheduling priority. Once selected, all KV cache blocks associated with the preempted request are immediately freed from GPU memory to make space for current requests. The preempted request itself is not discarded, it remains logically active within the scheduler and can later resume execution. However, because its cached key and value tensors have been completely removed, the system has no remaining record of its intermediate computation. When that request is resumed, vLLM must recompute its entire KV cache from the beginning of the prompt, effectively replaying all prior decoding steps to construct the necessary attention states.

This mechanism guarantees that memory is always available for progressing requests and prevents deadlock conditions where all GPU memory is full. However, it introduces redundant computation overhead, especially in workloads with frequent preemptions or long contexts, since recomputation can consume significant time and GPU cycles that could otherwise be used for new tokens.

For example, consider a scenario with three concurrent requests, A, B, and C running on a 24 GB GPU. Requests A and B have long prompts and have already allocated most of the GPU memory for their KV caches. When a new request C arrives, vLLM attempts to allocate KV blocks for it but later discovers that memory is not enough. To make space, the scheduler preempts the most recent request, C itself, freeing all of its partially built KV cache. Later, when C is resumed (for instance, after A finishes), it must recompute all of its previous tokens from scratch because none of its KV blocks were preserved. In effect, the GPU work performed during C's earlier partial run is wasted.

2.4 Introducing LMCache: Hierarchical KV Storage via Connector Interface

LMCache serves as an KVCacheConnector that integrated into vLLM. This connector abstracts a multi-tier KV storage hierarchy spanning GPU, CPU, and disk. The offloading flow has two main operations:

- During the forward pass of the attention layers, vLLM periodically invokes LMCache to asynchronously store newly generated KV blocks from GPU memory. These KV blocks are first buffered in CPU RAM and further offloaded to SSD. This data movement occurs without blocking the main inference pipeline, while KV blocks are being transferred in the background, token generation and attention computations continue uninterrupted.

- Later, when vLLM processes a new token chunk of the same request or resumes a preempted one, it retrieves previously stored KV blocks. LMCache looks up for a matching entry corresponding to the current token prefix or previously computed layer states, and if found, streams the relevant tensors back into GPU memory. This mechanism enables reuse of already computed attention states, effectively avoiding redundant forward passes for known chunks.

- LMCache saves KV cache tokens into blocks of tokens and each block is saved into a separated file. The size of this block/file is configurable through chunk_size in LMCache. As LMCache could create a large number of files, the choice of file system would directly impact the performance of vLLM.

Through this design, LMCache acts as an intermediate between GPU, CPU, and SSD, allowing vLLM to decouple memory management from inference scheduling. As a result, the system can use lower-tier storage as an extension of GPU memory, improving flexibility and efficiency under varying workload and memory conditions.

2.5 Three Distinct Use Cases of Offloading

The same GPU-to-CPU/SSDs transfer can represent three distinct caching mechanisms: prefix caching, swapping, and multi-turn caching. Understanding their distinctions is key to analyzing performance outcomes.

2.5.1 Prefix Caching (Inter-Request Reuse)

Prefix caching is designed to exploit prompt reuse across different requests. In many real-world applications, such as instruction-tuned models, prompt templates, multiple user queries begin with identical or nearly identical prefix tokens.

Mechanism:

- The input is broken down into blocks of tokens. LMCache computes a hash of each block.
- When a new request arrives, vLLM queries LMCache to check if a matching prefix exists.
- If a match is found, the cached KV tensors are loaded to GPU memory and appended, skipping recomputation of that prefix.

In this scenario, once one request has computed the KV cache for a given prefix, subsequent requests can reuse those same key and value tensors rather than recomputing them.

2.5.2 Swapping (Intra-Request Offloading Due to Preemption)

Swapping occurs within the lifetime of a single request when GPU memory becomes insufficient. In vLLM's original design, preemption leads to complete eviction, all KV blocks of the request are discarded. With LMCache, these blocks can instead be offloaded asynchronously to a lower-tier device before being freed from GPU memory. Mechanism:

- When a request is preempted, vLLM initiates freeing KV cache blocks of this request in GPU memory.
- LMCache temporarily holds these blocks in CPU RAM or SSD, depending on capacity and policy as these blocks are asynchronously saved during attention computation.
- Later the request is resumed, if a match is found, the cached KV tensors are directly loaded to GPU memory and appended, skipping recomputation of that prefix.

This behavior is particularly relevant in the continuous batching setting, where requests put pressure on memory. Swapping remedies the memory pressure by temporarily moving out inactive request states.

2.5.3 Multi-turn Caching

Multi-turn caching extends caching beyond a single request to support conversational persistence. In interactive applications (e.g., chatbots), users engage in multi-turn dialogues, where each new turn depends on the model's previous output.

Without caching, each turn would require reprocessing the entire conversation history, leading to cost growth with the number of turns. With multi-turn caching:

- LMCache preserves the KV states corresponding to the dialogue history.
- When the next user message arrives, only the new tokens need to be processed; prior tokens' KV states are reloaded.

Figure 1 summarizes the differences between three usecases.

Table 1: Comparison of KV cache offloading behaviors in vLLM + LMCache.

Behaviors	**Scope**	**Trigger**	**Goal**	**Typical Benefit**
Prefix caching	Across requests	Reused prompt prefix	Avoid recomputation	Saves compute for repeated prefixes
Swapping	Within request	GPU memory exhaustion / preemption	Manage memory pressure	Relieving preemption overhead
Multi-turn caching	Across turns (same user session)	Conversational continuation	Enable persistence	Reduce latency in dialogue systems

3 Methodology

3.1 Motivation

Current LLM serving systems have a fundamental challenge: as KV cache for prefix caching grows beyond GPU memory capacity, offloading becomes necessary. While existing research has explored CPU memory offloading and proposed hierarchical approaches, the performance characteristics of disk-based KV cache management remain incompletely understood. Given that disks represent the last tier in the memory hierarchy and the most cost-effective storage option for large-scale deployments, understanding their practical limitations is essential for building efficient, economically viable LLM serving infrastructure. Meanwhile, among the three different scenarios of KV caching offload, in this work, we focus on the performance characterization of KV cache only for prefix caching.

3.2 Research questions and experiments

This section will set out the research questions that we try to answer and how we design experiments for each research question. The first research question (RQ) is about the performance variations under different factors, comparing offloading to disk versus keeping cache at CPU memory or only at GPU. This aims to answer whether offloading could actually deliver any benefits for the inference engine under prefix caching optimization. Next, we propose the second research question to answer how different hardware configurations would affect the performance. Finally, we try to investigate the underlying operations of disk offloading by last research question, what are I/O access patterns during offloading.

3.2.1 Research question 1: Performance Characterization

Core question "What is the prefilling performance in TTFT (2.2.2) for vLLM when managing the prefix on storage devices compared to the GPU HBM and host DRAM memory?"

Why does this matter? This research question establishes the fundamental performance characteristics of storage-based KV cache offloading by measuring the actual overhead introduced when prefix caches are stored on disks compared to GPU or CPU memory. By comparing these configurations, we identify where the cost of reading cached data from storage/SSDs exceeds the computational cost of recomputation or what is the other way, thereby defining when offloading remains beneficial. Table 2 shows the experiments included in this research question and details of experiment are presented in 4.

Table 2: Experiments included

Experiment	Purpose	How it contributes
Varying prompt lengths (see 2.2.1) with 100% hit rate	Shows how offloading overhead scales with context size; identifies whether there is a point where SSD offloading becomes viable.	Determines when recomputation is cheaper than offloading.
Different hit rate	Shows how offloading benefits vary with different hit rates.	Determines what kinds of workloads benefit from offloading.
Different chunk sizes (see 2.4)	Examines how chunk size (i.e., number of files) affects performance.	Reveals the optimal chunk size.
Different file systems (see 2.4)	Identifies whether file system design significantly impacts KV cache I/O patterns.	Provides actionable guidance on infrastructure configuration.

3.2.2 Research question 2: Hardware variations

Core question "How do different GPU variations interact with SSD offloading performance?"

Why does this matter? Real-world deployments operate under fixed hardware budgets. Different GPU types vary in memory size, memory bandwidth, and compute throughput, all of which influence how often offloading is triggered and how costly it is.

3.2.3 Research question 3: I/O Pattern Characterization

Core question "What does KV cache I/O actually look like at the storage layer?"

Why does this matter? Understanding read/write patterns and access characteristics enables software optimization, identifying opportunities for prefetching, batching, or compression or bottleneck identification to determine if latency or bandwidth is the limiting factor. It is crucial for building specialized KV cache storage backends or optimizing existing systems.

Table 3: What to observe

Metric	**Purpose**	**How it contributes**
Disk access pattern	Shows the access patterns at the storage layer (e.g., sequential vs. random).	Guides optimization efforts by revealing how to tune the storage layer for the observed patterns.
Request counts by request size	Captures the distribution of I/O operations across different transfer sizes, revealing whether the workload is dominated by small random accesses or large sequential transfers.	Informs block-size configuration and system design choices based on the access pattern.
File access patterns	Measures how many files are accessed, access order, and access latency.	Reveals opportunities for designing more efficient KV-cache block organization.

3.3 Experiment setups

This part explains the general setup throughout the experiments. Later, in each experiment, there is a specific part to explain the details of the setup for that specific experiment.

3.3.1 Model

All the experiments use meta-llama/Llama-3.2-3B-Instruct. The reason behind this is that it has a small number of parameters that could be hosted on DAS-6 GPUs while still offering GPU memory space for KV cache. Larger models require substantially more memory for parameters and proportionally more KV cache per token, which is not suitable for DAS-6 GPUs. Each token requires 114688 bytes of cache, which is 112 KiB. The parameters of this model take up 6 GiB of GPU memory. Even though this real-world inference systems deploy models at 7B, 8B, 14B, or larger, but it is representative for experimental setups constrained by academic GPU clusters such as DAS-6.

3.3.2 Server

The servers used in this project vary across experiments. There are three different types:

- **Node 027 (1):** NVIDIA RTX A5000 (24,564 MiB), 122 GiB CPU memory, 48 cores
- **Node 032 (2):** NVIDIA RTX A6000 (49,140 MiB), 122 GiB CPU memory, 48 cores
- **Node 010 (3):** NVIDIA RTX 4000 Ada Generation (20,475 MiB), 245 GiB CPU memory, 32 cores

3.3.3 Storage

For each node, the storage configurations differ:

- **Node 027 (1):** NFS with 200 GiB space
- **Node 032 (2):** NFS with 200 GiB space
- **Node 010 (3):** Samsung MZQL21T9HCJR-00A07 SSD

NFS uses the network, so reading and writing files can be slower and can vary depending on network traffic. NVMe is local storage, so it is much faster and more stable because it does not rely on the network.

3.3.4 How to Warm Up the Cache

Before measuring the actual performance of the system while using prefix caching, we must make sure that the requests sent are actually cached in the system. Across the experiments, we use a consistent mechanism to warm up the cache at different layers:

- **GPU:** Send a request and then immediately send the same request again.
- **CPU memory:** Send a request, then send several additional requests whose total KV-cache size exceeds the GPU memory capacity. This guarantees that the KV-cache blocks of the first request are evicted to CPU memory.
- **Disk:** Same principle as CPU memory: send a request, then send enough requests such that their total KV-cache size exceeds both GPU and CPU memory capacities. This forces the KV-cache blocks of the first request to reside only on disk.

4 Experiments details, results and analysis

All the experiments done in this parts used vLLM as the inference engine and LMCache as a plugin to offloading KV cache blocks to CPU memory and disks.

4.1 Research question 1: Performance Characterization

4.1.1 Experiment 1

In this experiment, we measure the TTFT (see 2.2.2) of a request by warming up the cache at different storage locations, ensuring that the request will get a cache hit and KV blocks of it will be fetched back from the backend storage.

Experiment setup We evaluated four serving schemes by measuring Time-to-First-Token (TTFT) at different prompt lengths. Because TTFT is dominated by the prefill phase, it is the most appropriate metric for detecting performance differences arising from prefix reuse and KV-cache offloading. The model is Meta-Llama-3.2-3B-Instruct (each token accounts for 122KiB in KV cache) with output len=1. TTFT primarily reflects prefill latency and therefore is the right way to check for prefix reuse and offloading effects. The experiment is run on NVIDIA RTX A5000 with 24GiB. This means under GPU utilization of 90%, the model takes up 6.02 GiB and leaves 14.19 GiB for KV caching. The maximum number of tokens this GPU could handle then is 132856 tokens. Next, we also run the experiment on Node 10, with the same approach.

- **Re-computation (no prefix caching):** prefix caching is disabled, every request recomputes the entire prefix. This represents the baseline without KV re-use across identical prompts. It is compute-bound and scales with context length because the prefill phase must rebuild all attention states each time. Configuration-wise, we disabled prefix-caching of vLLM using –no-enable-prefix-caching
- **GPU-only:** we warm the cache once by sending a request, then send repeated identical prompts so subsequent requests reuse the cached KV directly from GPU memory. With vLLM's paged attention, this avoids prefilling cost for a request and is the best-case for latency when the cache fits on device.
- **CPU offloading:** After warming, we send additional large requests to evict the warmed KV blocks from GPU into CPU RAM (via the connector). When the original prompt is sent again, the system fetches KV from CPU back into GPU prior to decoding. More specifically, first we send a request with the targeted input length followed by two 80K tokens requests (surpassing maximum of 132K tokens in GPU) to fully flush KV blocks of the first request out to CPU memory. This process is repeated 10 times to get the average TTFT. CPU offloading happens when the GPU no longer has room for KV cache. By sending the target request and then enough large requests to fill the GPU, we force vLLM to move the target KV blocks into CPU memory. This reflects real situations where long prompts or many requests push KV data off the GPU.
- **Disk offloading:** After warming, we flush KV beyond CPU capacity so the warmed request's KV goes to disk. A subsequent replay might reload from disk $\rightarrow$ CPU $\rightarrow$ GPU or goes directly from disk to GPU with GPU-Direct Storage (GDS). In this work, we ignore GDS as it requires a file system supportings GDS. More specifically, CPU memory is set to 18GiB and disk size is set to 64GiB. Then we first send the request with targeted length, followed by three requests with 100K tokens (which is equivalent to 32 GiB).

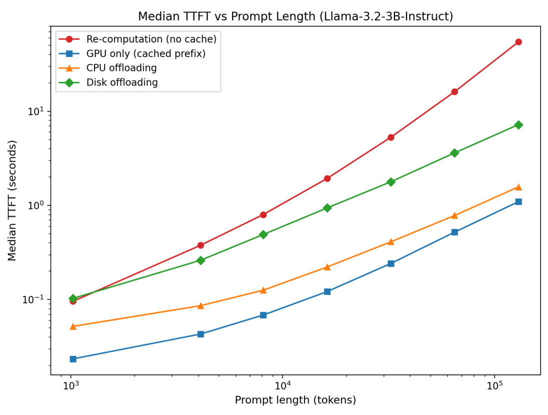
Hypothesis Given that prefix reuse occurs with a 100% cache hit rate, we hypothesize that the performance of the prefix KV cache will follow a tiered hierarchy aligned with the bandwidth and latency characteristics of the storage medium used. Specifically:

- GPU-resident KV caching will achieve the lowest TTFT and highest prefill throughput, as the KV blocks remain in high-bandwidth HBM and require no data movement.
- CPU memory offloading will introduce a moderate overhead, but should still retain a substantial portion of the latency benefit compared to recomputation.
- Disk (SSD) offloading will lead to higher TTFT than CPU offloading, as KV blocks must be staged through CPU memory before reaching the GPU, but it is expected to remain faster than full recomputation.
- Recomputation will incur the highest latency because the entire prefix must be reprocessed and either NFS or SSD setup in the experiment gives a low latency of read operations.

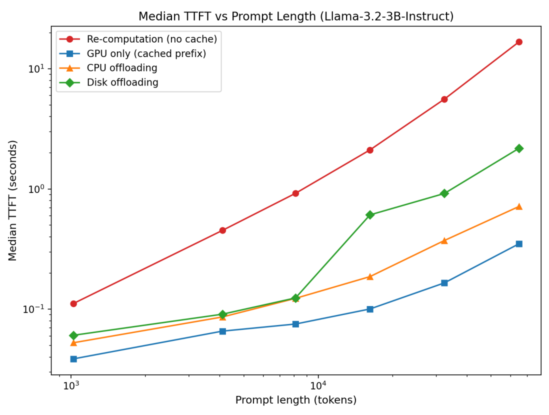
Sanity check This part explains how sanity checks are carried out to make sure that the experiment works as expected.

- Recomputation: This is trivial as the option is supported in vLLM by passing the correct configuration. We can check the vLLM logs showing that the cache hit rate is 0%.
- GPU-only: This is also straightforward as we do not enable CPU memory, and the vLLM log shows that the cache hit rate is 100%.
- CPU-memory: This is more complex since the vLLM log shows a hit rate of 0%. However, LMCache exposes its own metrics at the vLLM metric endpoint. We can look at two main metrics, lmcache:lookup_hit_rate and lmcache:retrieve_hit_rate, which should be 1 during the experiment. Another useful metric is lmcache:num_hit_tokens_total, which should roughly match the number of tokens generated.
- Disk: The LMCache metrics also reflect a 100% hit rate, and files are created at the designated folder. Another indicator is lmcache:local_storage_usage, which matches exactly the number of tokens generated multiplied by the KV cache size (bytes per token).

Figure 3 shows the results of this experiment on two different nodes, which plots TTFT (y-axis) against input prompt length (x-axis) for the four serving schemes.



(a) Experiment 1 result on Node 27



(b) Experiment 1 result on Node 10

Figure 3: Experiment 1 results on two different nodes

Quantitative result overview

- In out setup, TTFT increases linearly with input length under recomputation on both nodes.
- GPU caching keeps TTFT at 1.09 seconds for 129K tokens, which corresponds to a 49.4× speedup on Node 27 and a 30.4× speedup on Node 10.
- Disk offloading is much slower than GPU caching, with the latency gap growing significantly as prefix length increases. For short prefixes (e.g., 8K tokens), disk TTFT is about 0.42 s vs. 0.21 s on GPU, 2.0× slower. For very long prefixes (e.g., 129K tokens), disk TTFT rises to 7.20 s vs. 1.10 s on GPU, 6.5× slower. Yet from 8K tokens onward, disk offloading remains 5–7× faster than recomputing the prefix from scratch.

These results establish a clear hierarchy of performance aligned with the caching tiers at 100% hit rate: GPU, CPU, disk, recomputation.

Analysis

Prefix caching is beneficial Given the condition that the prefix matching rate is 1, prefix caching provides performance improvement. Without caching, TTFT grows proportionally with the number of input tokens, since every token must be reprocessed to rebuild the attention keys and values. Prefix caching breaks this dependency: once the prefix KV cache is stored, subsequent queries bypass prefill and start generation immediately.

GPU caching achieves the lowest latency The GPU-only configuration shows the lowest TTFT among other schemes (recomputation, CPU memory offloading, and disk offloading). In practice, it enables low latency (less than 1 second) even at long context input (100K+ tokens), which is essential for long-document chat or retrieval-augmented LLMs.

CPU offloading preserves performance with minor overhead Offloading the KV cache to CPU DRAM introduces overhead but retains the same scaling trend. It increases TTFT by only 1.5× compared to GPU caching while extending memory capacity. This makes it a promising secondary storage tier for large-context workloads without suffering from recomputation latency.

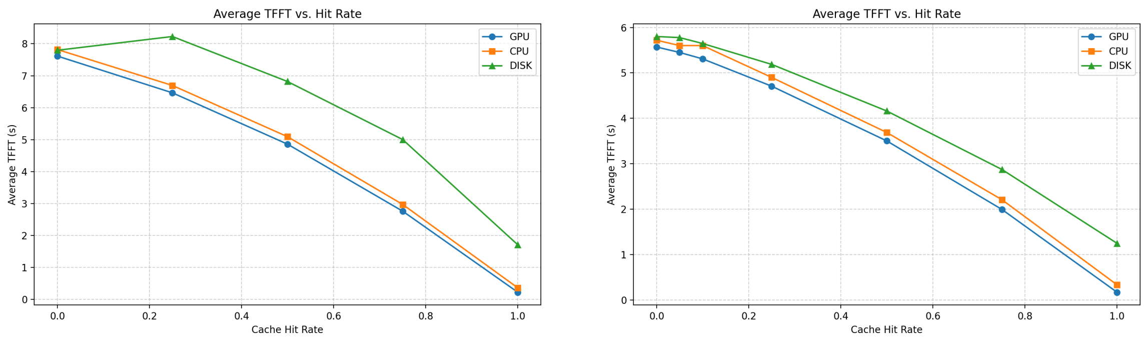
Disk offloading extends capacity further, at a cost Although the disk tier adds I/O latency, the TTFT remains lower than recomputation. This shows that loading KV states from persistent storage is still cheaper than regenerating them via transformer prefill.

Recomputation becomes expensive for long contexts Recomputation time grows from milliseconds to tens of seconds as context length increases. At 129K tokens, recomputation is 50× slower than GPU caching (54.38 s vs 1.10 s), while CPU offloading is 1.44× slower than GPU (1.57 s) and disk offloading 6.6× slower (7.20 s). Only at very short contexts (below 1K tokens) is recomputation preferable to disk offloading.

4.1.2 Experiment 2

Experiment Setup Experiment 2 evaluates performance under different cache hit rates. The input length is fixed at 32K tokens as it is large enough for prefix caching to matter while still fitting comfortably within the GPU's KV-cache capacity. The cache is warmed up using a prefix corresponding to the desired hit rate. Requests are generated by taking that prefix and appending additional tokens to reach the target length.

Hypothesis At all hit rates, disk offloading always show lower TFFT than recomputation.



(a) Experiment 2 results on Node 27 (b) Experiment 2 results on Node 10

Figure 4: Experiment 2 results across two nodes

Quantitative Result Overview The results on both nodes show a consistent pattern: increasing hit rate reduces TTFT significantly.

- **GPU caching achieves sub-second TTFT at high hit rates.** On Node 027, TTFT drops from 7.62s at hit rate zero to 0.22s at 1.0, a 34× improvement. On Node 10, TTFT drops from 5.57s to 0.17s, a 33× improvement.

- **CPU offloading remains within 5–10% margin to GPU performance when hit rate exceeds 90%** On Node 027, CPU TTFT at 1.0 hit rate is 0.36s, only 1.6× slower than GPU caching. On Node 10, CPU TTFT is 0.33s, about 2× slower than GPU caching.
- **Disk offloading provides gains when hit rate is above 90%** At 1.0 hit rate, disk TTFT reaches 1.71,s on Node 027 and 1.25,s on Node 10. This is ∼ 7–8× slower than GPU caching but still ∼ 4–5× faster than full recomputation.
- **At low hit rates, disk offloading may underperform recomputation.** On Node 27, disk TTFT at hit rate of zero and 0.25 (7.80–8.23s) slightly exceeds the recomputation baseline (7.62s). On Node 10, disk TTFT at 0.0–0.1 hit rate (5.65–5.80s) is marginally slower than the recompute baseline (5.57s).

Analysis

Higher hit rates translate directly into reduced TTFT When the cache hit rate approaches 1.0, nearly all prefill computation is avoided. TTFT drops from seconds to sub-second levels, demonstrating that prefix caching successfully removes the linear dependence between TTFT and the number of input tokens. Each cached prefix eliminates redundant computation, turning prefill into fast lookups.

Disk offloading does not always outperform recomputation at low hit rates For hit rates below 0.3 on Node 27 (in which NFS is used as the storage backend), the overhead of moving data from disk through network can make recomputation faster. Once the hit rate surpasses this threshold, cached prefixes amortize data movement costs, causing TTFT to fall rapidly. This suggests that for workloads with low prefix locality, systems must carefully weigh the trade-off between recomputation and lookup-based retrieval. The threshold where caching becomes beneficial depends on factors such as the model architecture, GPU type, and broader hardware configuration.

4.1.3 Experiment 3

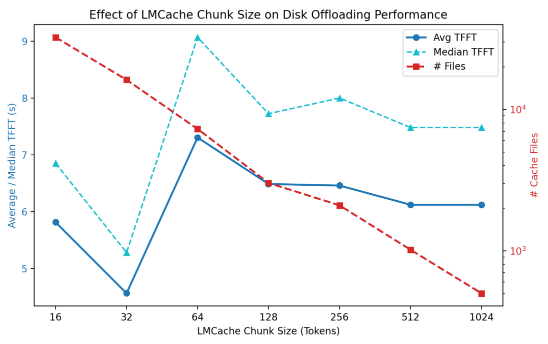
Experimental setups The experiment shows how LMCache chunk size affects performance when caching large key–value tensors to remote storage during prefix reuse. Each run uses the same model (Llama-3.2-3B-Instruct) with four independent requests, each having an input length of about 129K tokens. Together, these occupy roughly 55 GiB of KV-cache data, forcing offloading from GPU to CPU, and finally disk as new requests arrive. By adjusting chunk_size, LMCache divides this cache into different-sized blocks before writing them to disk. The following chunk sizes were tested: 16, 32, 64, 128, 256, 512, and 1024 to representing a wide range of chunk sizes, producing roughly 32 K to 500 files respectively on Node 27. Meanwhile, chunk sizes from 16 to 8192 are tested on Node 10. Each configuration was benchmarked using 10 repeated runs, measuring average TTFT (time to first token) and total benchmark duration to capture how file granularity influences latency under remote I/O.

Why this experiment? By testing a range of chunk sizes, this experiment shows how storage granularity influences the latency of fetching KV data from disk during prefix caching. This helps determine not only whether disk-tier caching is viable but also under what storage granularities it maintains acceptable prefill performance, directly contributing to answering RQ1.

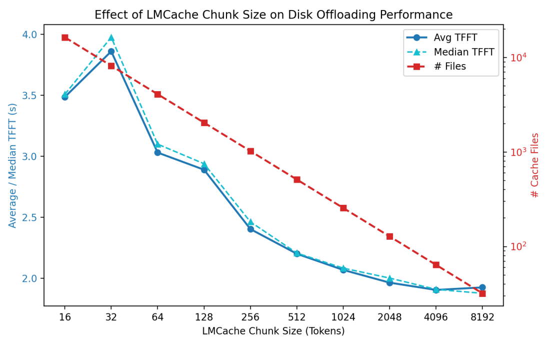
Hypothesis Since smaller chunk sizes increase the number of files and I/O operations required to reconstruct the KV cache from disk, we hypothesize that TTFT will increase as chunk size decreases. When the chunk size is small, prefix reloading requires many separate read operations, which can introduce higher file-system overhead and I/O scheduling contention, especially under repeated retrieval. Conversely, larger chunk sizes should require fewer reads, reducing metadata lookups and system call overhead, and therefore should yield lower TTFT and shorter total recovery time.

Quantitative Result Overview

- Node 27 shows a non-linear pattern: by an initial improvement, followed by a slowdown, and then a stable region.
- Node 10 shows a downward trend: as chunk size increases, TTFT consistently reduces.
- The difference likely comes from the storage system: Node 27 uses NFS, while Node 10 uses a local SSD, which handles large sequential reads much better.



(a) Experiment 3 results on Node 27



(b) Experiment 3 results on Node 10

Figure 5: Experiment 3 results 98 two nodes

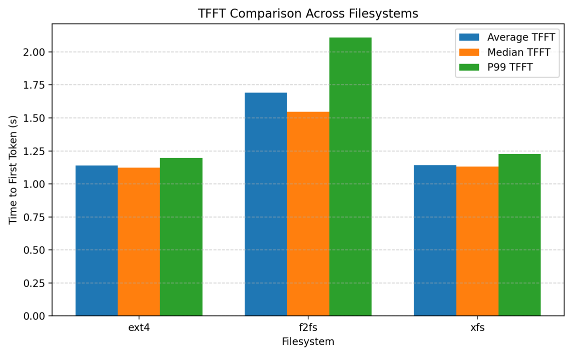


Figure 6: Experiment 4 results

Analysis

Largest gain happens early The biggest drop in TTFT comes from increasing chunk size from very small values (e.g., 16 to 32). This is where the system benefits the most.

Large chunks provide small returns From 128 MiB upward, TTFT changes very little. Increasing chunk size beyond this point does not meaningfully improve first-token time.

Stability at higher chunk sizes Once past the mid-range, performance settles into a predictable, stable pattern, meaning the system is no longer sensitive to chunk size adjustments.

NFS introduces instability The baseline performances in Appendix A and B showed a huge gap between the performance of network-based storage and SSD-based storage. We hypothesis that the instability in the NFS-based node showed in this experiment is partly due to the characteristics of network-base storage - dependent on network status and a higher latency.

4.1.4 Experiment 4

Experiment Setups In this experiment, we evaluate different filesystems (`ext4`, `xfs`, `f2fs`) using the same setup as Experiment 1 with 32K input tokens. All experiments are conducted on Node 10, which uses a local SSD.

Hypothesis We hypothesize that filesystem design characteristics, particularly metadata handling, management overhead, and I/O scheduling, will affect KV-cache offloading performance. `ext4`, as a general-purpose filesystem with full journaling, introduce moderate overhead. `f2fs`, designed for flash storage with a log-structured layout, show better average performance but suffer higher tail latency due to garbage collection. `xfs`, optimized for large files and parallel I/O, is expected to perform similarly to `ext4` for our workload.

Quantitative Result Overview

- `ext4` and `xfs` show nearly identical performance: average TTFT $\sim$ 1.13–1.14 s, median $\sim$ 1.12–1.13 s, P99 $\sim$ 1.19–1.22 s.

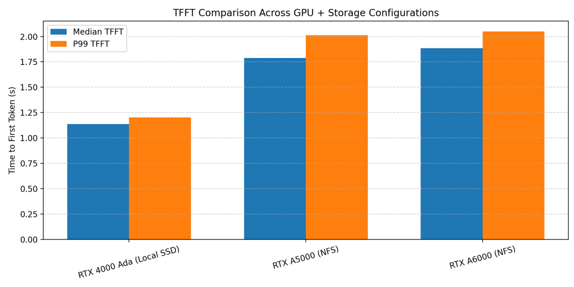


Figure 7: Experiment 5

- `f2fs` performs significantly worse: average TTFT ~ 1.69 s ($\sim 50\%$ slower), median ~ 1.54 s, P99 ~ 2.08 s ($\sim 70\%$ higher than ext4/xfs).
- The gap between average and P99 is small for `ext4`/`xfs` ($\sim$ 7–8%) but substantially larger for `f2fs` ($\sim 23\%$).
- Median values closely match averages across all filesystems, suggesting consistent behavior for typical requests.

Analysis

`ext4` and `xfs` provide comparable, optimal performance. Both filesystems produce nearly identical TTFT metrics across all percentiles, with average latencies around 1.13 s.

`f2fs` significantly underperforms despite being flash-optimized. Instead of improving performance, `f2fs` shows $\sim 50\%$ higher average TTFT and $\sim 70\%$ higher tail latency compared to `ext4`/`xfs`. This unexpected result likely arises from `f2fs`'s log-structured design, which introduces garbage-collection pauses and write-amplification effects when handling read-heavy KV-cache workloads with periodic updates.

Implications for RQ1. This experiment shows that filesystem selection can introduce significant and sometimes surprising overhead when offloading KV cache to SSDs. The intuition that flash-specific filesystems always perform best does not hold for LLM-serving workloads. System designers should prefer mature, general-purpose filesystems such as `ext4` or `xfs` over flash-optimized alternatives for this use case.

4.2 Research question 2: Hardware variations

Experiment Setup This experiment reuses the 32K-token input from Experiment 1 (as this represents the base line case where the length of input is large enough to trigger a large amount of disk operations) and evaluates how different hardware configurations affect prefix-cache loading under disk-tier offloading. Three GPU systems were tested: an RTX 4000 Ada node backed by a local Samsung SSD, and RTX A5000 / RTX A6000 nodes backed by the `/var/scratch/` NFS remote filesystem. All experiments used identical vLLM configurations and the same input workload.

Hypothesis We expect the RTX A5000 and A6000 systems to show higher TTFT because retrieving KV blocks from a remote NFS filesystem introduces additional latency and overhead. The RTX 4000 Ada system, using a local SSD, should achieve lower and more stable TTFT, with GPU compute capabilities playing only a secondary role.

Figure 7 shows the results of this experiment.

Quantitative Result Overview

- Median TTFT increases from 1.13 s on the RTX 4000 Ada node to 1.79–1.88 s on the NFS-backed A5000/A6000 nodes.
- P99 TTFT increases substantially as well, from 1.20 s (local SSD) to 2.01–2.05 s on NFS-backed systems.
- Prefill throughput is reduced by approximately 30% on the NFS-backed systems, indicating additional I/O bottlenecks during KV-block retrieval.

Analysis

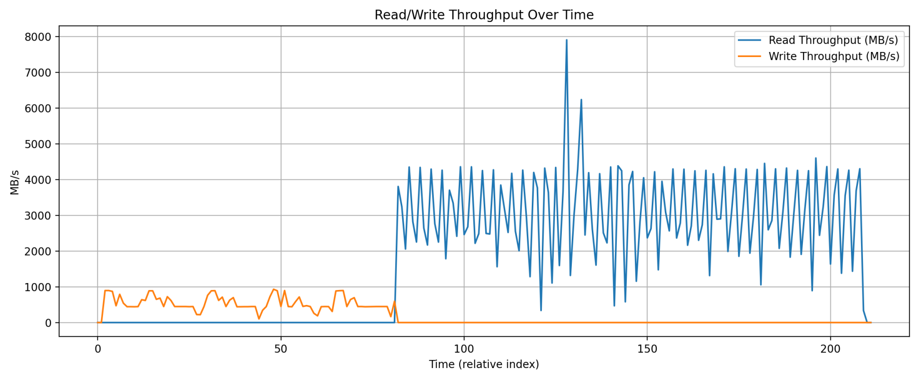


Figure 8: Read-write throughput

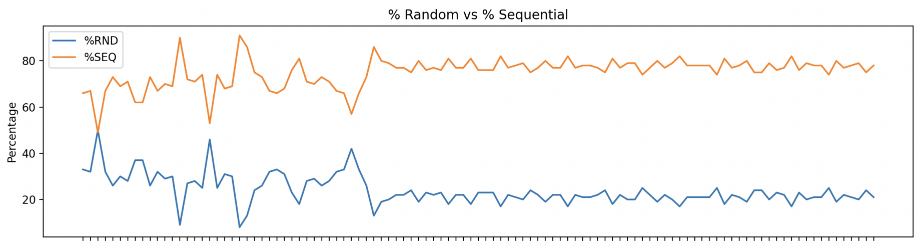


Figure 9: Sequential/Random access pattern using **ext4** file system

The dominant performance factor is the storage backend, not the GPU. The RTX 4000 Ada node benefits from a local NVMe SSD.

NFS-backed A5000/A6000 nodes incur significant overhead. We hypothesize that network latency and metadata operations on the remote filesystem directly increase TTFT, despite these nodes having more powerful GPUs.

GPU compute no longer matters once NFS latency dominates. The nearly identical performance of A5000 and A6000 indicates that GPU compute and memory bandwidth have minimal effect under remote-sto

4.3 Research question 3: I/O Pattern Characterization

Experiment setups In this experiment, we continued to use 32K token input from Experiment 1 and observe the disk operations using bpftrace. This experiment is run on Node 10 with an SSD. Again, we first send 5 requests to the server to warm up the cache, then repeatedly send these five requests to the server and observe the disk behaviour. Except from the sequential and random I/O patterns, all other metrics are measured under ext4 file systems.

Hypothesis As this is a simple experiment, it does not involve real world workload so the behavior should be easy to expect:

- As in the cache warming phase, only write requests are issued to disk. Later when processing requests only means fetching KV cache back and forth, then the workload only exhibits read requests.
- Request size distribution: We expect KV cache blocks to translate into large I/O requests (>1MB) since each cache block contains contiguous key-value tensors.
- File system access patterns: as we repeat sending requests 10 times, as a result, each file should be opened 10 times. Total number of bytes written to each files should be by chunk size multiplied by KV cache for each token (28MiB in this experiment).

4.3.1 Sequential vs random I/O patterns

Figure 9, 10, and 11 show the sequential and random access patterns of the experiment across three different file systems. Both ext4 and xfs access pattern shows a mixed workload of sequential and random access. After that, sequential access dominates the access patterns. In contrast, f2fs behaves almost entirely as random access. Recalling that in 4.1.3, f2fs gives the highest TTFT, which suggesting a correlation between random access pattern and TTFT.

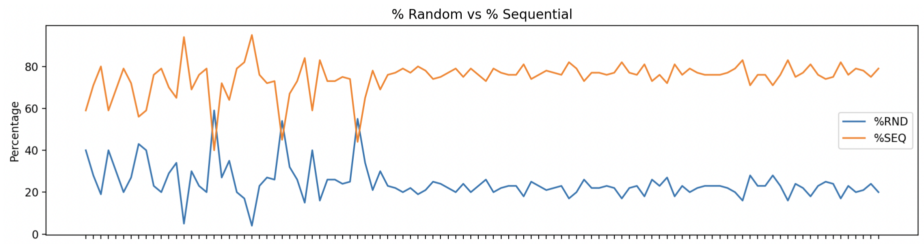


Figure 10: Sequential/Random access pattern using **xfs** file system

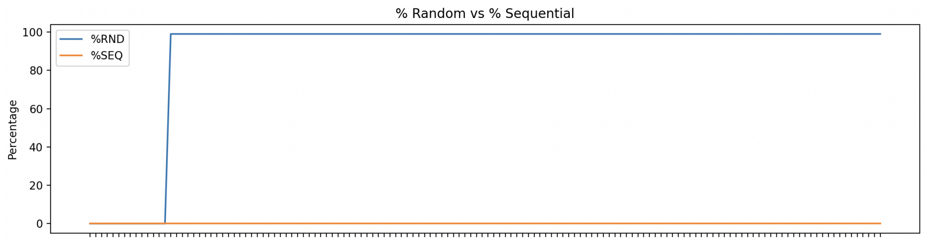


Figure 11: Sequential/Random access pattern using **f2fs** file system

4.3.2 Request size distribution

Figure 12 showed that workload is dominated by large requests (greater than 1MB): 150,000 read requests, 17,000 write requests. Meanwhile there is a moderate activity in 256-512KiB range, about 10,000 read requests, 3,000 write requests and a minimal small I/O activity (less than 256KiB), only a few thousand requests total.

4.3.3 File access patterns

Table 4: File Access Statistics

Metric	**Value**
Number of times opening the file	10
Number of bytes written to the file	29,360,128
Number of bytes read from the file	264,241,152

Each file is opened exactly 10 times, matching the number of repeated requests and confirming deterministic prefix reuse. Bytes written per file (28 MiB) indicate that each KV block is written only once during the warm-up phase. Bytes read per file (264 MiB) show that the same 28 MiB block is read 10 times, once for each repeated request. The 1:10 write-to-read ratio confirms that LMCache performs no redundant disk writes and reloads each KV block precisely once per query.

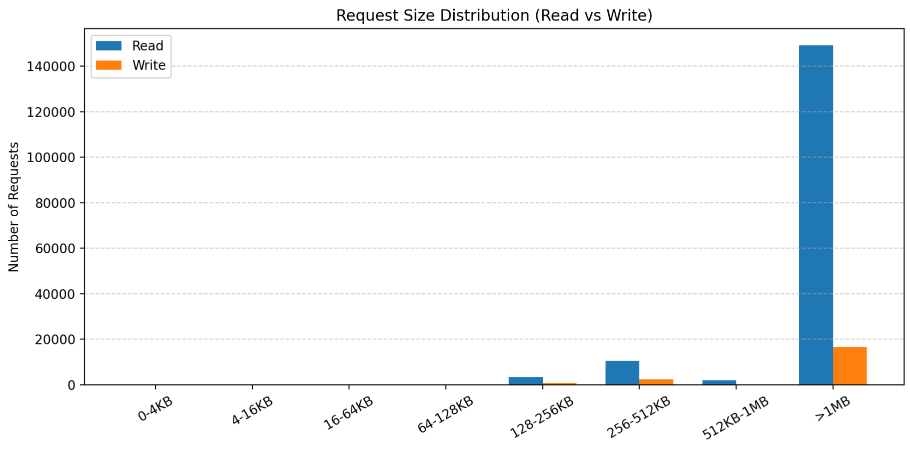


Figure 12: Request size distribution

4.3.4 Analysis

Sequential and random patterns mix due to block-level access across multiple files Within each file, reads are sequential. Across files, the inter-file order produces randomness.

Prefix reuse produces large sequential reads Once warmed up, the workload becomes read-only, with the majority of data fetched as large multi-megabyte contiguous reads. These reads reflect LMCache's storage of KV tensors in large, sequentially laid out chunk files.

Request size distribution is dominated by large I/Os Most read operations exceed 1 MB, confirming that LMCache retrieves KV data in large blocks aligned with chunk boundaries. Only a small number of requests fall into sub-256 KB categories.

File system behavior influences access patterns and performance. ext4 and xfs stabilize into sequential-dominant access and provide comparable TTFT performance, while f2fs exhibits highly random access patterns and delivers the worst TTFT. This suggests that LMCache's large sequential read workload aligns better with traditional block file systems (ext4, xfs) than with log-structured designs such as f2fs.

4.4 Summary of the experiments

Throughout all experiments, we examined how prefix KV-cache offloading to different memory tiers and storage backends affects LLM prefill performance, focusing on TTFT as the primary metric.

Experiment 1 evaluated the performance hierarchy under a 100% prefix hit rate. We compared four configurations: full recomputation without prefix caching, GPU-resident prefix caching, CPU-memory offloading, and disk-based offloading. The results in our setup showed a clear tiered ordering: GPU caching < CPU offloading < disk offloading < recomputation. GPU caching kept TTFT below ∼1.1 s even at ∼129K tokens, achieving up to ∼50× speedup over recomputation. CPU offloading introduced only an overhead (about 1.5× slower than GPU), while disk offloading was 5–7× faster than recomputation for sufficiently long prefixes, despite being slower than GPU caching. In practice, this shows when each tier is viable: GPU offers the lowest latency; CPU preserves most benefits when GPU memory is full; disk still bring benefits; and recomputation should be avoided. It provides a clear baseline for deciding which tier to rely on during serving.

Experiment 2 studied the impact of varying prefix hit rates at a fixed 32K-token context. As the hit rate increased, TTFT decreased sharply for all offloading tiers. GPU and CPU tiers maintained sub-second or near-sub-second TTFT at high hit rates. Disk offloading is clearly beneficial once the hit rate crossed a workload- and system-dependent threshold: at low hit rates, disk-based retrieving speed could be slightly slower than recomputation, but at high hit rates it delivered 4–5× lower TTFT than recomputation. This highlights that the effectiveness of disk-based prefix caching is highly sensitive to prefix locality. In practice, it is useful for identifying when to enable or disable disk-tier reuse and demonstrates that caching policies must be adaptive to workload characteristics.

Experiment 3 shows the effect of LMCache chunk size (i.e., KV-block granularity) on disk-based prefix loading. On the NFS-backed node, we observed a non-consistent pattern with an "improve → decrease → stabilize" trend as chunk sizes increased, reflecting the interaction between chunk size and networked filesystem overheads. On the SSD-backed node, larger chunk sizes consistently reduced TTFT, with the largest gains obtained when moving away from very small chunks (e.g., 16 MiB). Beyond a mid-range threshold (128 MiB and above), performance improvements became marginal, suggesting a threshold where the system is technically insensitive to further chunk-size increases. In practice, this shows that very small chunks should be avoided and that moderate-to-large chunk sizes provide stable performance. It is useful for tuning LMCache to reduce filesystem overhead.

Experiment 4 focused on the role of filesystem choice for disk-based KV caching on a local SSD. We compared `ext4`, `xfs`, and `f2fs` under identical workloads. `ext4` and `xfs` exhibited nearly identical and consistently low TTFT, while `f2fs` showed higher average and tail latencies (up to ∼50–70% worse). This result indicates that mature general-purpose filesystems (`ext4`, `xfs`) are better suited to KV-cache workloads than flash-optimized `f2fs` in this setting. In practice, this shows that ext4 and xfs are better suited for KV-cache workloads and should be preferred in LLM serving systems. Filesystem choice matters significantly, and f2fs is not a good default for prefix caching unless carefully tuned.

Experiments 1–4 together answer RQ1 by quantifying how much latency in terms of TFFT each storage tier (GPU, CPU, and disk) adds during prefix reuse, and by identifying the factors that control whether offloading is beneficial. Experiment 1 establishes the performance hierarchy; Experiment 2 shows that the effectiveness of offloading depends on prefix-hit rate; Experiment 3 shows that chunk size directly shapes disk-tier latency; and Experiment 4 reveals that filesystem choice can further increases or reduces storage overhead.

Experiment 5 addressed hardware variation, comparing three GPU nodes (RTX 4000 Ada with local SSD, RTX A5000 and RTX A6000 with NFS-backed scratch space). Despite the A5000/A6000 being more powerful GPUs, the node with the local SSD consistently achieved lower and more stable TTFT. NFS-backed nodes incurred 30–60% higher median and P99 latencies, demonstrating that once KV cache hits the disk tier, the storage backend (local vs. remote) dominates performance, and additional GPU compute does not compensate for I/O delays. In practice, disk I/O could potentially be a bottleneck. It highlights that storage placement decisions could be even more important than just GPU capability for offloaded KV caching. **Experiment 5 answers RQ2** by showing how different hardware setups influence disk-based prefix loading.

Finally, Experiment 6 characterized the I/O patterns at the storage layer using `bpftrace`. After warm-up, the workload became read-dominated with large, mostly sequential reads and a small fraction of smaller I/Os. Each file was written once and read multiple times, resulting in a high read-to-write ratio and confirming that prefix caching behaves like a read-heavy, large-block workload with intra-file sequential and inter-file random access. These observations are important for tuning chunk sizes and choosing appropriate filesystems and storage hardware. **Experiment 6 answers RQ3** by showing the actual I/O behavior when KV data is stored on disk. The workload showed mostly large, sequential reads with repeated accesses to the same files, and only a small amount of smaller or random I/O. This helps clarify what kinds of read patterns storage systems must handle when serving prefix-cached KV blocks.

5 Conclusion and Future Work

In this work, we evaluated how disk-based KV-cache offloading for prefix caching affects LLM inference performance, using vLLM integrated with LMCache across multiple hardware and storage configurations. Our experiments focused on prefill latency (TTFT) under varying prefix lengths, hit rates, chunk sizes, filesystems, and GPU/storage combinations, and complemented these measurements with a low-level I/O characterization.

Overall, our results show that prefix caching remains highly beneficial even when the KV cache must be served from CPU memory or SSDs, as long as prefixes are reused sufficiently often and contexts are not extremely short. At a hit rate of 100%, GPU-only cache delivers the lowest TTFT and largest speedups, but CPU DRAM offloading holds most of the benefits with only little overhead. Disk-based offloading, while clearly slower than GPU and CPU tiers, still gives latency gains compared to full recomputation for medium and long contexts. Only at very short prefixes and low hit rates does recomputation become competitive or slightly faster than disk.

We also found that storage and system design choices are crucial. Chunk size has a strong impact when it is too small, but benefits quickly saturate beyond a mid-range size where fewer large I/Os amortize filesystem overheads. Filesystem selection can make a surprisingly large difference: general-purpose filesystems such as `ext4` and `xfs` provided lower and more stable TTFT than `f2fs`, despite the latter being explicitly optimized for flash. Finally, hardware experiments showed that local SSDs consistently outperform NFS-backed storage for KV-cache retrieval.

From an I/O perspective, prefix caching shows a read-heavy, large-block workload with sequential access within individual files and random access across files. Each KV block is typically written once and read many times, making write performance less critical than read throughput, latency, and filesystem metadata behavior. This pattern is good for modern SSDs.

References

[1] Vincenza Carchiolo and Michele Malgeri. Trends, challenges, and applications of large language models in healthcare: A bibliometric and scoping review. *Future Internet*, 17(2), 2025.

[2] Georgia Chalvatzaki, Ali Younes, Daljeet Nandha, An Le, Leonardo F. R. Ribeiro, and Iryna Gurevych. Learning to reason over scene graphs: A case study of finetuning gpt-2 into a robot language model for grounded task planning. *arXiv preprint arXiv:2305.07716*, 2023.

[3] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'24, USA, 2024. USENIX Association.

[4] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[5] Yinheng Li, Shaofei Wang, Han Ding, and Hang Chen. Large language models in finance: A survey. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, ICAIF '23, page 374–382, New York, NY, USA, 2023. Association for Computing Machinery.

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

[7] Emma Roth. Openai says chatgpt users send over 2.5 billion prompts every day, July 2025. Accessed: 2025-09-12.

[8] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: high-throughput generative inference of large language models with a single gpu. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[10] Hanyi Xu, Wensheng Gan, Zhenlian Qi, Jiayang Wu, and Philip Yu. Large language models for education: A survey. 05 2024.

A Performance baseline for NFS

We benchmarked the performance of the NFS-mounted storage on DAS-6 using `fio` (v3.41). The goal was to measure random read throughput. The `psync` engine was used to emulate synchronous POSIX read behavior.

A.1 Methodology

- Random reads (`randread`), 128 KiB blocks
- 32 jobs, I/O depth 1 (synchronous)
- Direct I/O, 120 s runtime (10 s ramp)
- Filesystem target: `/var/scratch/szz435/testfile.img`
- Command:

```
fio --ioengine=psync --bs=128k --iodepth=1 \
  --rw=randread --numjobs=32 --direct=1 \
  --time_based=1 --runtime=120s --ramp_time=10s \
  --filename=/var/scratch/szz435/testfile.img
```

A.2 Results

- Throughput: 1942 MiB/s (2.0 GB/s)
- IOPS: 15.5k
- Avg latency: 2.06 ms
- 99th percentile latency: 2.64 ms
- Total data read: 228 GiB

These measurements provide a baseline for the achievable random-read throughput and latency over NFS on DAS-6, serving as a reference point in subsequent experiments.

B Performance baseline for NVMe SSD

To establish a baseline for raw storage performance, we benchmarked the local NVMe SSD using the `fio` tool (version 3.36). The goal of this was to quantify the peak random read throughput and latency characteristics of the device. The benchmark was run on `node10` using the `io_uring` I/O engine.

B.1 Methodology

The following configuration was used:

- **Operation**: 100% random reads (`randread`)
- **Block size**: 128 KiB
- **I/O depth**: 64
- **Jobs**: 4 parallel workers
- **Duration**: 120 seconds of measurement after a 10-second ramp period
- **Direct I/O**: enabled (`--direct=1`)
- **Engine**: `io_uring` with `--registerfiles=1` and `--fixedbufs=1`
- **Target**: raw block device `/dev/nvme0n1p1`

The exact invocation was:

```
sudo fio --ioengine=io_uring --registerfiles=1 --fixedbufs=1 \
  --bs=128kb --iodepth=64 --group_reporting=1 --time_based=1 \
  --ramp_time=10s --runtime=120s --size=100% --direct=1 \
  --rw=randread --allow_file_create=0 --filename=/dev/nvme0n1p1 \
  --numjobs=4 --name=disk-baseline
```

B.2 Results

The device sustained high random read performance throughout the run. Aggregated across all four jobs, we observe:

- **Throughput**: 6787 MiB/s (6.6 GiB/s)
- **IOPS**: 54.3k operations/s
- **Average latency**: 4.7 ms (end-to-end)
- **Latency distribution**:
  - 50th percentile: 4.69 ms
  - 90th percentile: 5.41 ms
  - 99th percentile: 6.06 ms
  - Maximum observed: 10.04 ms
- **CPU utilization**: 2.7% user, 6.8% system
- **Device utilization**: 80.44%

Across the 120 s period, the benchmark issued approximately 6.5 million read operations, totaling 795 GiB of transferred data. The overall throughput achieved the maximum throughput provided by the manufacturer.