A Study of Prefix Sharing in LLM Serving: Current Policies May not be Optimal for Mixed Workloads

Sergio Lavao [1] Sanya Garg [2] Marko Tanevski [2] Ana Urruticoechea [2] Sai Teja [1]

Abstract

Prefix sharing is a critical optimization in LLM serving that enables the reuse of precomputed KV values across requests sharing identical input prefixes. While modern serving engines like vLLM integrate this optimization, a comprehensive analysis of the system-level factors influencing its efficacy, such as cache capacity, block size, and eviction policies, remains underexplored. In this work, we modify vLLM to function as a trace-driven simulator to bypass GPU execution. We then systematically evaluate prefix sharing across multiple workloads. Finally, we propose and evaluate an optimized eviction policy designed for mixed workloads that reduces TTFT by up to 8%.

1 Introduction

1.1 Prefix Sharing Mechanism

Prefix sharing in vLLM (Kwon et al., 2023) enables requests with identical or partially identical input prefixes to reuse previously computed KV-cache blocks, avoiding redundant prefill computation and significantly reducing latency and memory traffic. When a new request arrives, vLLM identifies the longest matching prefix and reuses all KV blocks that align with this prefix at block boundaries; because the KV cache is organized into fixed-size blocks, reuse is performed through lightweight block reference mapping rather than copying. After the shared portion is mapped, only the unmatched suffix of the prompt requires new KV computation and decoding proceeds normally. The effectiveness of this mechanism depends heavily on cache capacity, block size, and the eviction policy, since evicting a shared block invalidates reuse for all dependent requests. Overall, prefix sharing is a core optimization that leverages structural redundancy across prompts to improve both throughput and latency in real-world LLM serving.

1.2 vLLM Architecture Overview

vLLM is a library that aims to maximize LLM throughput and minimize latency. Its architecture centers on three main pillars: continuous batching, a block-based KV cache, and prefix sharing. Continuous batching increases the chances that repeated prompts occur close together, improving temporal locality. Block-based KV caching determines the granularity of reuse, since too-large blocks reduce sharing, while too-small blocks may fragment memory. The cache eviction policy decides which prefixes survive long enough to be reused, which is crucial in heterogeneous workloads.

1.3 Our Contributions

Much work has been done on studying optimal cache parameters and cache eviction policies to maximize hit rate and minimize latency, including learning an eviction policy that predicts future reuse probabilities of conversations (Yang et al., 2025) and utilizing other heuristics like "hotness" (Li et al., 2025). Another important application of prefix serving is serving diverse workloads, and optimal approaches have been explored for this context as well (Pan et al., 2024). Nevertheless, this problem remains open. In this work, we conducted a rigorous study of cache performance regarding various cache parameters (such as cache size, block size, and eviction policies), and designed and benchmarked our own cache eviction policy on various historical datasets.

2 Methods

2.1 Bypassing GPU Inference

To enable study without heavy GPU resource requirements, we modified the vLLM engine to bypass actual GPU inference. The overall architecture is shown in Figure 1.

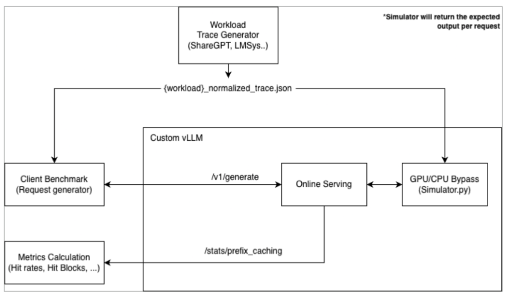

Figure 1. Simulation Architecture

2.1.1 Trace Generation

The `generate_traces.ipynb` script implements a preprocessing pipeline that cleans the workload dataset and extracts alternating human prompts and LLM responses, grouping them by conversation. To standardize multi-turn storage, we concatenate all turns within a conversation into a single sequence, called `full_conversation_text`, which is tokenized in one pass to generate `full_conversation_tokens`. To track the boundaries between interaction turns, we calculate cumulative offsets by processing each segment (i.e., each prompt or response) individually. This yields `char_spans`, which maps these boundaries in `full_conversation_text`, and `token_spans`, which records the endpoint of each segment in `full_conversation_tokens`. These arrays, which alternate between human and model segments, enable the simulator to distinguish prompt tokens from generated tokens during trace replay, and also allow the simulator to replay traces without having to do any tokenization by itself.

2.1.2 Simulator

Our simulator bypasses GPU execution by replacing the real GPU executor with a mock executor that replays pre-recorded token sequences. It intercepts executor selection and delegates to `TraceSimulator` instead of running model inference. The `TraceSimulator` uses a Trie to store conversation traces from JSON trace files containing full conversation tokens and token spans that mark human versus model-generated parts of a conversation. When `execute_model()` is called, `simulate_worker_step()` matches the current prompt tokens against the Trie and returns the next token from the stored trace. This allows the vLLM scheduler, block manager, and other components to run normally without the GPU.

2.1.3 Online Inference Client

Our client (`client_multiturn.py`) benchmarks the running vLLM server by replaying single and multi-turn conversations from the trace files. It reads each trace and sends each turn as an HTTP request to the server. For multi-turn conversations, the client accumulates the conversation history by appending each response to the prompt for the next turn and validates generated text against the expected output from the trace. It supports concurrent execution via a thread pool, allowing multiple conversations to run in parallel. After execution, it collects and reports prefix caching statistics (hit rates, time to first token), block reuse metrics, and overall test results (success/failure counts, early terminations).

All together, our implementations of the trace generator, simulator, and client enable us to evaluate prefix caching on realistic multi-turn workloads without running actual model inference.

2.2 Collecting Metrics

2.2.1 Metrics Collection Implementation

We collect three metrics that capture both cache efficiency and user-visible performance: (i) the per-request Hit Rate, which shows how often KV-cache entries are reused; (ii) the average Hit Rate, which summarizes cache effectiveness across each workload and configuration; and (iii) the Time-To-First-Token (TTFT), which indicates how cache behavior impacts actual latency. To support this, we modified `vllm/core/scheduler.py` to implement Hit-Rate tracking, we extended our simulator in `vllm/simulator.py` to record TTFT and compute aggregated statistics. These metrics let us evaluate the internal behavior of the KV-cache and its impact on end-to-end inference performance. We also added an API endpoint `/stats/prefix_caching` when doing online serving to expose these metrics during runs.

2.2.2 Automated vLLM Benchmarking Framework

To evaluate KV-cache behaviors under different workloads, we implemented an automated benchmarking pipeline on top of vLLM. The pipeline is driven by a shell script `vllm_benchmark_multiple_traces.sh` that takes an arbitrary list of traces files as input. For each trace, the script does a full sweep of experimental conditions, including varying the KV-cache size by `(num_gpu_blocks)`, `(block_size)` and switching between eviction policies. It then launches the corresponding vLLM runs, collects all relevant KV-cache metrics, and stores the results in a structured format for later analysis.

2.2.3 Metrics Collection Validation

This experiment, shown in Figure 2, validates our custom hit-rate implementation in vLLM by generating a controlled sequence of requests with progressively increasing shared prefixes. Using `client_offline_minimal.py`, we issue synthetic prompts where each new request appends tokens to the previous one (e.g., extending ”Hello” to ”Hello, how are...”). This gradual extension allows us to calculate exactly how many prefix tokens should hit in the existing KV cache. The KV Cache size is 100 blocks, each block is 1 token, and we send requests until we reach a token size of 1000.

Plotting the measured hit rate against the request token size reveals whether our metric matches reuse behavior: the hit rate rises sharply as the shared prefix grows, peaks when tokens fill the KV-cache window, and subsequently decays as the request size exceeds cache capacity and the KV cache fills only a fraction of the request. This predictable behavior confirms two things: (1) the hit-rate computation in `vllm/core/scheduler.py` correctly tracks prefix reuse, and (2) the KV cache management logic responds as expected under controlled growth.

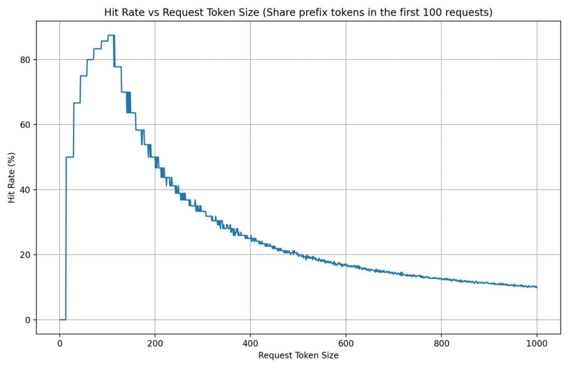

Figure 2. Hit Rate Validation

2.3 Cache Eviction Policy

2.3.1 LRU, LFU, and FIFO

vLLM employs the Least Recently Used (LRU) algorithm as its default cache eviction policy, prioritizing the removal of the least recently accessed blocks when free space is required. In accordance with project specifications, we additionally implemented the First-In, First-Out (FIFO) and Least Frequently Used (LFU) policies for the prefix token cache. These three algorithms are well-established staples of cache management research.

2.3.2 Custom Policy: Aging LFU Evictor

Both the LRU and LFU policies leverage specific access pattern characteristics, i.e. usage recency and usage frequency, respectively, to predict future demand. However, research in web proxy caching has demonstrated that combining these metrics yields a superior predictor (Cherkasova, 1998). Taking this into account, we developed the Aging LFU Evictor, a hybrid eviction policy inspired by the Greedy-Dual-Size-Frequency algorithm. This policy integrates LFU's frequency-based prioritization with a temporal aging mechanism to prevent the retention of obsolete, high-frequency blocks. Formally, the Aging LFU Evictor calculates a retention score S for each block as:

$$S = N_{accesses} - A$$

where $N_{accesses}$ denotes the cumulative number of accesses since the block's insertion, and A (age) represents the number of global cache operations elapsed since the block was last accessed. The policy evicts the block with the lowest score. This approach addresses the pitfalls of standard policies: it prevents "cache pollution" by historically popular but currently irrelevant blocks (a weakness of LFU) while simultaneously robustly handling "one-hit wonder" blocks that can thrash an LRU cache. Of course, to mitigate the computational overhead of updating the age A for every block during every cycle, our implementation employs a global inflation counter. This counter is added to a block's access count upon each access. Because the inflation value increases monotonically over time, the resulting rank ordering of blocks remains mathematically equivalent to that of the theoretical model above.

We also explored implementing additional cache eviction policies, including (Yang et al., 2025), which uses a learned method to perform LLM prefix cache eviction. This method uses conversational content analysis to predict which conversations are likely to continue to determine what should be evicted from the cache.

2.4 Homogenous and Heterogenous Workloads

A homogeneous workload is a workload composed of requests of the same type, which in our experiments was defined as workload with requests from the same dataset. These are the following datasets we utilized for simulating homogeneous workloads while evaluating the parameters:

- ShareGPT: general conversations with ChatGPT (ShareGPT Team, 2023)
- AgentBank: instances of agentic usages of various LLMs (Song et al., 2024)
- LMSys: user-chatbot dialogues from a platform where users compare various LLMs (Zheng et al., 2023)
- CC-Bench trajectories: coding tasks (Z.ai Team, 2025)
- EkaCare: medical / healthcare note-taking (Eka Care, 2025)

Additionally, in order to evaluate prefix-sharing performance on truly heterogeneous workloads, we created a script (`generate_heterogenous_traces.py`) to interleave conversations from multiple datasets. In particular, these are the sizes of the datasets we worked with:

Table 1. Dataset Properties.

Dataset	Conversations	Avg. input length	Avg. output length
LMSys	4×10^3	63 tokens	179 tokens
ShareGPT	2.5×10^3	113 tokens	305 tokens
AgentBank	1.5×10^3	87 tokens	155 tokens
CCBench	0.37×10^3	821 tokens	413 tokens
EkaCare	0.16×10^3	2330 tokens	1267 tokens
Mixed	1×10^3	79 tokens	190 tokens

Note that, for practical purposes, we had to truncate some of the datasets and work with limited versions of them as doing the analysis on entire datasets was computationally expensive. Thus the above statistics might not reflect the full datasets.

3 RESULTS

3.1 Evaluation Metrics

To quantify prefix-sharing efficiency, we measured: (1) **hit rate**, the average fraction of blocks in a request that are found in the KV cache and reused; and (2) **TTFT**, the time spent from receiving a request until the first token is generated. Note that TTFT is inversely correlated with hit rate, because a high hit rate reduces the need to compute KV values for most tokens, allowing the decoding phase to begin faster.

3.2 Milestone 2 Cache Tuning Results

Figures 3 and 4 illustrate the relationship between the KV cache capacity (in tokens) and the average cache hit rate across the ShareGPT and AgentBank workloads. As expected, both datasets exhibit a monotonic increase in hit rate as cache capacity expands, confirming that larger allocation budgets consistently reduce re-computation. However, the functional form of this improvement differs significantly between the two workloads, reflecting distinct underlying access patterns.

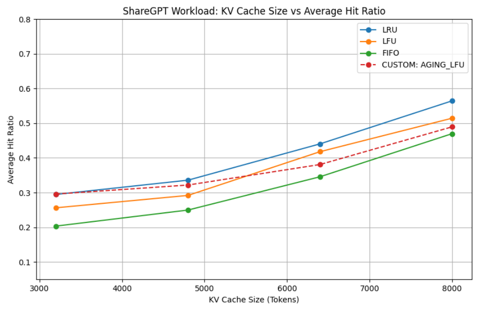

Figure 3. ShareGPT Workload Cache Size v. Average Hit Rate

ShareGPT (Figure 3) displays a largely linear growth trend. Even at the smallest constrained cache size (about 3200 tokens), the system achieves a nontrivial baseline hit rate of 0.2–0.3. This suggests that the workload contains a significant volume of short, frequently reused prefixes, e.g. system prompts or short conversational headers that fit within small allocations. The linear climb indicates that cacheable content is uniformly distributed across varying lengths.

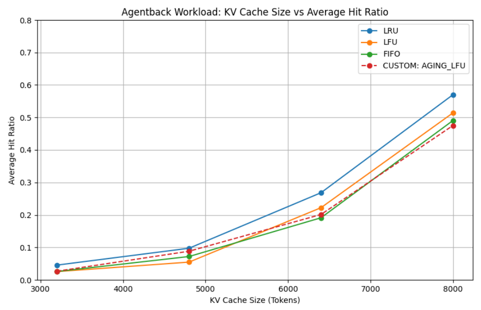

Figure 4. AgentBank Workload Cache Size v. Average Hit Rate

AgentBank (Figure 4), however, exhibits a convex growth pattern. Performance remains negligible (near 0.05) at lower cache sizes until a threshold is reached around 4800–6000 tokens, after which the hit rate accelerates. This behavior is characteristic of workloads with a large working set size. The agents in this dataset likely rely on long interaction histories or documents that need to be fully cached to yield hits. Until the cache size exceeds this working set threshold, the eviction policy is forced to thrash and discard parts of a sequence before they can be reused. Once the capacity is sufficient to hold these complete contexts, the hit rate improves dramatically.

3.3 Milestone 3 Eviction Policy Results

As illustrated in Figures 3 and 4, the empirical results align with theoretical expectations: FIFO exhibited the lowest performance on standard datasets, while LFU improved upon FIFO, and LRU performed marginally better than LFU. This hierarchy is consistent with the consensus in cache behavior literature, which posits that recency (LRU) is typically a more reliable predictor of future reuse probability than insertion order (FIFO) or frequency (LFU). (Smith, 1982) We note that the Custom Aging LFU eviction policy does not perform particularly well on the ShareGPT and AgentBank datasets, which may be a consequence of the specific access characteristics of these workloads. These datasets may have a simpler structure that is dominated by strong short-term temporal locality, a pattern that strictly favors recency-based eviction (LRU). Our Aging LFU policy is optimized for heterogeneous environments, where the system must discriminate between transient bursts and sustained long-term popularity. In the absence of significant frequency variation in the ShareGPT and AgentBank traces, the frequency-weighting mechanism in our custom policy likely introduces non-predictive noise into the eviction priority, offering no advantage over the simpler, recency-focused LRU baseline.

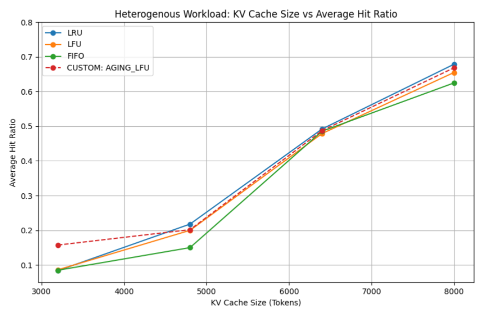

Figure 5. Mixed Workload Cache Size v. Average Hit Rate

On the other hand, the Aging LFU eviction strategy performs particularly well on heterogeneous workloads. As seen in Figure 5, it trails only slightly behind LRU when it comes to hit rate (but performing better than LFU and FIFO).

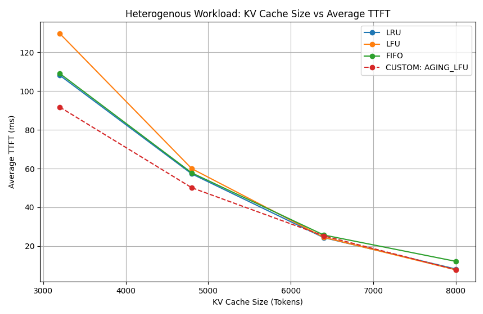

Figure 6. Mixed Workload Cache Size v. TTFT

Furthermore, as illustrated in Figure 6, Aging LFU outperforms the other, more standard eviction policies in the TTFT metric. The rationale for this performance is likely the presence of high-utility blocks, in the mixed workload—i.e., each workload type relies on important prefix blocks that require significant processing time, which Aging LFU is able to successfully retain. While LFU may also retain these blocks, it appears to be penalized by ephemeral heavy-hitting blocks; as noted above, Aging LFU strikes an effective balance between LFU and LRU.

3.4 Overall Mixed Workload Results

To summarize the effectiveness of our proposed policy, we will compare the Hit Rate and Time-To-First-Token (TTFT) of all policies at maximum cache capacity (`block_size=16`) and (`num_gpu_blocks= 500`):

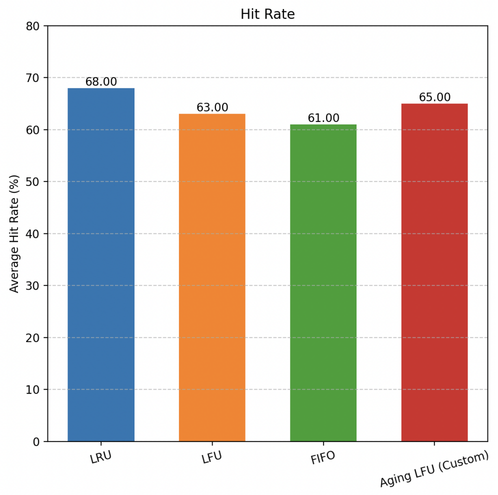

Figure 7. Mixed Workload Average Hit Rate

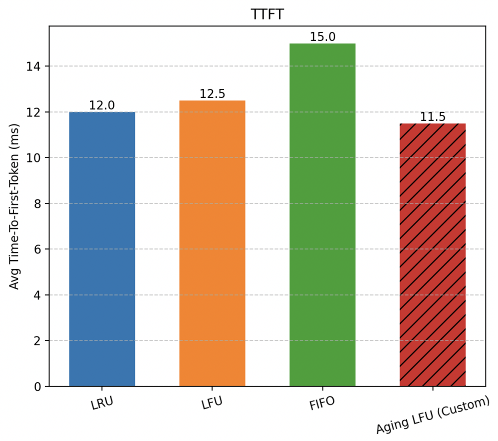

Figure 8. Mixed Workload Average TTFT

LRU remains the most effective Avg Hit Rate (68%), likely due to its ability to retain large, recently accessed context blocks. Our Aging LFU policy follows close (65%), proving it can still capture the majority of reusable prefixes.

Aging LFU outperforms LRU in latency, delivering the lowest Time-To-First-Token. This suggests that Aging LFU is making "smarter" evictions. By discarding high-utility blocks that clog the cache (high token count but low utility), it frees up capacity for a larger volume of lighter, latency-sensitive requests.

4 CONCLUSION

As a whole, our study reveals that prefix sharing offers substantial gains, but its effectiveness depends on workload structure, cache configuration, and eviction strategy. Conversational workloads such as ShareGPT exhibit high temporal locality, which enables significant block reuse and benefits from larger KV caches. More diverse workloads such as CCBench and heterogeneous mixed traces showed lower overall reuse, and increasing KV cache size consistently reduced TTFT, confirming the central role of memory pressure in enabling block reuse. LRU performed very well under tight memory and strong recency locality, which is common in conversational workloads, where LFU and our aging-LFU became increasingly advantageous as cache size grew or workloads became more diverse. On the other hand, FIFO consistently underperformed, regularly discarding reusable prefixes and yielding lower hit ratios. Ultimately, our findings highlight that effective prefix caching requires policies tailored to the locality patterns of the underlying workload.

5 AI Usage

Throughout this project, we experimented with using many different LLM and AI-coding assistants (ChatGPT, Gemini, AntiGravity, and Cursor) to understand the large vLLM codebase and assist with relatively simple processes such as generating Python and Bash scripts. We found it to be most useful for small explanation-related tasks, as we did encounter situations where using AI generated a lot of extraneous code that complicated the codebase instead of completing the task at hand. Additionally, Cursor's ability to combine its trained knowledge with its scan of the codebase was extremely helpful when fixing minor bugs. For example, when we were trying to implement online learning with the replay, we were stuck on an async error for hours until Cursor instructed us to add a small if-statement that ended up instantly fixing our issue.

References

Cherkasova, L. Improving www proxies performance with greedy-dual-size-frequency caching policy. Technical Report HPL-98-69, Hewlett-Packard Laboratories, Palo Alto, CA, November 1998.

Eka Care. Eka structured clinical note generation dataset. https://huggingface.co/datasets/ekacare/clinical_note_generation_dataset, 2025.

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.

Li, Y., Gu, R., Huan, C., Wang, Z., Yao, R., Tian, C., and Chen, G. Hotprefix: Hotness-aware kv cache scheduling for efficient prefix sharing in llm inference systems. *Proceedings of the ACM on Management of Data*, 3(4): 250:1–250:27, 2025. doi: 10.1145/3749168.

Pan, R., Wang, Z., Jia, Z., Karakus, C., Zancato, L., Dao, T., Netravali, R., and Wang, Y. Marconi: Prefix caching for the era of hybrid LLMs. *arXiv preprint arXiv:2411.19379*, 2024.

ShareGPT Team. Sharegpt: Share your wildest chatgpt conversations. https://sharegpt.com/, 2023.

Smith, A. J. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

Song, Y., Xiong, W., Zhao, X., Zhu, D., Wu, W., Wang, K., Li, C., Peng, W., and Li, S. Agentbank: Towards generalized LLM agents via fine-tuning on 50000+ interaction trajectories. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 1–20, 2024.

Yang, D., Li, A., Li, K., and Lloyd, W. Learned prefix caching for efficient llm inference. NeurIPS, 2025.

Z.ai Team. Cc-bench-trajectories. https://huggingface.co/datasets/zai-org/CC-Bench-trajectories, 2025.

Zheng, L., Chiang, W.-L., Sheng, Y., Li, T., Zhuang, S., Wu, Z., Zhuang, Y., Li, Z., Lin, Z., Xing, E. P., et al. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *arXiv preprint arXiv:2309.11998*, 2023.

[1]Department of Electrical and Computer Engineering, Rice University, Houston, Texas, USA [2]Department of Computer Science, Rice University, Houston, Texas, USA. Correspondence to: Marko Tanevski <mt102@rice.edu>.

Proceedings of the 8th *MLSys Conference*, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).