UniCache: Unifying Prefix Cache Eviction for Heterogeneous LLM Serving Workloads

BEI OUYANG*, Rice University, United States
YIFAN QIAO, UC Berkeley, United States
JIARONG XING, Rice University, United States

Prefix caching is a key performance optimization in Large Language Model (LLM) serving systems, enabling reuse of attention Key-Value (KV) states across requests with shared prompt prefixes. However, the size of GPU memory limits cache capacity, making the eviction policy a critical factor in overall system performance. Existing systems primarily rely on simple heuristics, such as LRU, and apply the same policy across task categories, implicitly assuming homogeneous workloads. In practice, however, modern LLMs serve heterogeneous workloads that mix multi-turn conversational traffic with diverse single-turn API requests, leading to fundamentally different prefix reuse patterns. In this work, we first design a trace-driven prefix cache simulator built on vLLM to systematically characterize prefix reuse across representative workloads. Our analysis reveals two dominant reuse patterns—session reuse and structural reuse—that vary significantly across task types. Motivated by these observations, we propose UniCache, a unified eviction policy that jointly captures both reuse patterns and dynamically balances cache allocation across tasks. When implemented in vLLM, UniCache achieves substantial improvements under heterogeneous workloads, improving prefix cache hit rates by up to 17.32% and reducing inference latency by up to 3.63× compared to existing policies.

CCS Concepts: • **Computer systems organization**; • **Computing methodologies** → **Modeling and simulation**;

Additional Key Words and Phrases: LLM Serving, Prefix Caching, Eviction Policy

ACM Reference Format:
Bei Ouyang, Yifan Qiao, and Jiarong Xing. 2026. UniCache: Unifying Prefix Cache Eviction for Heterogeneous LLM Serving Workloads. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 2, Article 54 (June 2026), 27 pages. https://doi.org/10.1145/3805652

1 Introduction

Prefix caching [6, 7, 21, 27, 36, 38] is a widely adopted optimization to improve Large Language Model (LLM) serving efficiency [9, 37, 41]. It exploits redundancy across requests by reusing previously computed attention Key-Value (KV) states (i.e., KV cache) [30]. Specifically, when multiple requests share an identical input (prompt) prefix, the KV cache corresponding to the prefix tokens is identical. By caching and retaining these KV states, the serving system can reuse them for subsequent requests and thereby avoid redundant computation for the shared prefix. As a result, prefix caching can substantially reduce request latency and improve system throughput, and has therefore been adopted in nearly all modern LLM serving systems [17, 23, 32, 41].

However, fully realizing the benefits of prefix caching requires the system to carefully retain prefixes under tight GPU memory constraints. Modern flagship GPUs typically provide 80-140 GB

*This work was done during Bei Ouyang's internship advised by Jiarong Xing.

Authors' Contact Information: Bei Ouyang, Rice University, United States, bei_ouyang@outlook.com; Yifan Qiao, UC Berkeley, United States, yifanqiao@berkeley.edu; Jiarong Xing, Rice University, United States, jxing@rice.edu.

ACM 2476-1249/2026/6-ART54
https://doi.org/10.1145/3805652

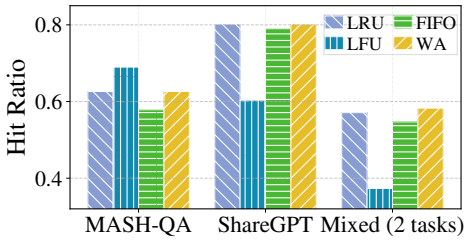


Fig. 1. **KV-cache hit ratio under different eviction policies.** LLAMA-3.2-1B-INSTRUCT on A100 across MASH-QA, ShareGPT, and Mixed (MASH-QA+ShareGPT).

Table 1. Hit ratio breakdown of the **Mixed** workload (ShareGPT+MASH-QA) in Fig. 1.

Eviction Policy	Total	MASH-QA	ShareGPT
LRU	0.5709	0.4247	**0.7457**
LFU	0.3732	0.3627	0.3858
FIFO	0.5469	0.4092	0.7113
WA	**0.5827**	**0.5083**	0.6717

of High-Bandwidth Memory (HBM) [15, 16], a substantial fraction of which is consumed by storing LLM model weights. As a result, only a limited residual memory budget is available for KV cache storage. Although the exact footprint varies with model architecture and precision, a single prefix token typically consumes several megabytes of GPU memory. Consider an LLaMA-70B model deployed across four NVIDIA A100 GPUs with tensor parallelism (TP) degree four. Each cached token requires approximately 5 MB of GPU memory, leaving room for only tens of thousands of cached tokens in total (e.g., ~36K tokens with 180 GB available for caching).

Similar to CPU caches, when GPU memory is exhausted, LLM serving systems will evict previously cached tokens to accommodate new ones. Mainstream production engines, such as vLLM [9] and SGLang [41], adopt a Least Recently Used (LRU) policy by default. This design prioritizes temporal locality and implicitly assumes that recency is a reliable predictor of future reuse. Recent work also designs eviction policies for specific workloads [10, 22, 34]. For example, Workload-Aware (WA) policy [34] classifies requests by task category (e.g., multi-turn or single-turn) and applies LRU within each category separately.

While these eviction strategies can be effective for specific LLM workloads served independently, real-world inference workloads are inherently heterogeneous and include multiple task types. Each foundation model today supports a broad spectrum of applications, such as conversational agents [18], document understanding [42], and code generation [14]. In production, these capabilities are delivered through two primary access channels: user-facing web portals and programming interfaces (APIs) [1, 19, 34]. Portal-based services enable interactive, session-oriented usage, where end users perform multi-step tasks, such as dialogue, file analysis, and research assistance, over multiple conversational turns [13, 28, 29]. In contrast, API-based services expose LLMs through stateless API calls, where requests are typically single-turn and support tasks such as tool invocation [8], code generation [14], and document question answering (QA) [42].

These diverse workloads exhibit qualitatively different prefix reuse patterns, rendering eviction strategies that rely on a single fixed heuristic inherently suboptimal. To demonstrate this effect, we evaluated the prefix cache hit ratios of four eviction policies across multiple representative workloads. As shown in Fig. 1, no single policy consistently dominates: Least Frequently Used (LFU) performs best on MASH-QA, whereas LRU and WA achieve higher hit rates on ShareGPT and the mixed workload. Moreover, even for WA, it fails to achieve the best hit ratio for all task categories comprised of the heterogeneous workload, as shown in Table 1. This variability highlights a key limitation of existing approaches and underscores the need for a unified eviction policy that can dynamically adapt to heterogeneous task characteristics in hybrid LLM serving environments.

Designing an effective eviction policy requires extensive profiling across diverse workloads to uncover reuse characteristics and iterative trial-and-error evaluation of candidate policies.

Conducting such profiling directly on real hardware is both time-consuming and costly, making rapid design-space exploration impractical. To address this issue, we build a CPU-based KV cache simulator on top of vLLM that enables controlled, trace-driven analysis of prefix caching behavior (Section 3). The simulator can replay request traces through the serving pipeline and faithfully reproduce cache events, such as hits, insertions, and evictions, without executing GPU inference, thereby enabling fast and systematic evaluation of eviction policies.

Using this simulator, we conducted an extensive characterization of prefix reuse in LLM serving across seven representative task categories (Section 4). Our profiling yields several observations; we give two key examples here. (1) Prefix reuse is dominated by two distinct patterns: session reuse and structural reuse. *Session reuse* arises within multi-turn interactions, where successive requests incrementally extend prior context and reuse prefixes from earlier turns in the same session. In contrast, *structural reuse* stems from fixed, template-like prompt prefixes that recur across independent requests, such as tool-use instructions or safety policies. (2) Prefix reuse exhibits strong task locality: cache hits overwhelmingly occur among requests belonging to the same task category, with negligible reuse across task boundaries. For example, requests for code generation rarely reuse prefixes generated by math or question-answering workloads.

Inspired by these observations, we propose UniCache, a *unified* prefix cache eviction policy that jointly captures session reuse and structural reuse, enabling high cache hit ratios under realistic LLM serving workloads where requests from multiple task types are mixed and served by a single model (Section 5). The core challenge here lies in the *non-comparability* of the optimal eviction policies across tasks: eviction heuristics that are optimal for one task type can be suboptimal for others. Addressing this challenge requires both task-specific eviction logic and a global coordination mechanism that reconciles competing objectives to maximize the overall cache hit ratio.

We address this challenge using two key design principles. (1) To support task-specific eviction behavior, we organize cached KV states into task-specific queues, each governed by a priority metric tailored to the task's dominant reuse pattern. Upon eviction, each queue independently selects a candidate entry according to its local policy. (2) To select the final evicted entry among these candidates, we introduce a global metric called hit efficiency, which quantifies the trade-off between a queue's contribution to cache hits and the amount of cache capacity it consumes. This metric rewards queues that achieve high hit ratios with modest memory usage while penalizing those that exhibit poor reuse despite occupying substantial cache space. Together, these mechanisms allow the eviction policy to dynamically bias cache allocation toward workloads that provide the greatest overall efficiency in prefix reuse.

We implemented our policy in vLLM and evaluated it under diverse workload compositions. Across hybrid traces, our policy improves prefix caching hit ratio by 3.86%–17.32% and reduces QTTFT by 1.10×–3.63× relative to existing baselines.

In summary, our key contributions are summarized as follows:

- We build a CPU-based simulator to efficiently analyze prefix-caching behavior under controlled workloads and cache budgets.
- We develop an offline optimal eviction policy that represents an upper bound on the theoretical prefix-cache hit ratio.
- We characterize prefix reuse across diverse LLM serving workloads, identifying distinct reuse mechanisms and properties that inform eviction-policy design.
- We propose and implement a unified cache eviction policy that jointly captures session and structural reuse patterns, improving cache hit rate by 3.86%–17.32% and reducing latency by 1.10×–3.63× under heterogeneous workloads.

2 Background and Motivation

2.1 Background: Generative LLM Inference Basics

LLM inference consists of two main phases: *prefill* and *decode*. In the prefill phase, the model processes input prompts in a batched manner, with all prompt tokens computed in parallel, and generates the first output token. This is followed by the decode phase, in which the model generates subsequent tokens auto-regressively, using the previously computed states to produce one token at a time. Accordingly, different metrics are used to evaluate their performance: Time-to-First-Token (TTFT) measures the latency from when a request starts execution until the first output token is generated during prefill, while Time-Per-Output-Token (TPOT) captures the average time to generate each subsequent token during the decode phase, reflecting decoding throughput.

Most of today's LLMs are built on the self-attention mechanism [30], which captures dependencies among tokens in a sequence by allowing each token to attend to all preceding tokens. Concretely, self-attention is computed as $\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$, where the query ($Q$) of the current token is multiplied with the keys (K) of all prior tokens and aggregated with their corresponding values (V). As a result, each decoding step must load and process the full set of historical K and V tensors, causing inference cost to grow linearly with sequence length. To avoid this inefficiency, modern systems cache the K and V values of previously processed tokens in a KV cache [9, 41] and reuse them across decoding steps.

2.2 Background: Prefix Caching in LLM Serving Systems

Beyond reusing KV states within a single request, LLM serving systems can also reuse KV states *across requests* when those requests share an identical prompt prefix. If two requests begin with the same prefix, then the KV states corresponding to the prefix tokens are identical and can be safely reused. For example, consider the following two requests: (1) "You are a helpful assistant. Summarize the following document ..." (2) "You are a helpful assistant. Answer the following question ..." The underlined prefix is identical in both requests; therefore, the KV states corresponding to this shared prefix can be cached once and reused across the two requests.

This type of cache is referred to as a *prefix cache*. Modern LLM serving engines, such as vLLM [32] and SGLang [41], implement prefix caching using a block-based representation. Specifically, they partition KV states into fixed-size *KV blocks*, each storing the KV states for a contiguous span of tokens. These KV blocks are organized into a Trie structure [41] to efficiently represent and index shared prefixes across requests. When a new prompt arrives, the system traverses the Trie to identify the longest matching prefix between the incoming prompt and previously cached prefixes. If a match is found, the system reuses the corresponding KV blocks; otherwise, it computes the missing prefix tokens and inserts them into the Trie for future reuse.

Prefix caching is widely deployed in production LLM serving systems and plays a critical role in improving inference efficiency. Previous studies show that realistic workloads often exhibit substantial prefix reuse, especially due to shared system prompts, instruction templates, and conversational histories [6, 10, 27, 32, 34, 38]. In such settings, prefix cache hit ratios commonly range from 20% to more than 80%, depending on the composition of the workload and the cache capacity. Exploiting this reuse can significantly reduce prefill computation, leading to 2-5× reductions in TTFT under memory-constrained deployments [9, 34, 41]. As a result, effective prefix caching has become a first-class optimization in modern LLM serving stacks, directly impacting latency, throughput, and overall GPU efficiency.

2.3 Motivation: Leashed Prefix Cache Performance Under Heterogeneous Workloads

Prefix cache design would be straightforward if GPU memory were unconstrained: it could simply retain all computed KV states and achieve the maximum hit ratio. In practice, however, GPU memory is tightly limited (e.g., 80 GB), and only a small fraction can be used to cache prefix after accounting for model weights. For example, processing a context with a length of 1K tokens for a Llama-7B model requires approximately 15 GB of memory, a fraction of an A100 GPU's capacity, while a context with a length of 1000K tokens demands over 500 GB, equivalent to the combined memory of about seven A100 GPUs [11]. As a result, eviction policy becomes critically important, as it directly determines cache contents and, consequently, the achievable hit ratio. However, the mismatch between increasingly diverse and complex workloads and the simple eviction policies used today, most commonly LRU, fundamentally limits the performance of prefix caching.

Diverse workload types served by one LLM. In today's AI ecosystem, a single foundation model is used to support a wide range of applications and access patterns. For instance, OpenRouter's large-scale usage study organizes real-world traffic into a set of 11 dominant task categories [20]. The same model can be simultaneously used through user-facing web portals that support multi-turn, interactive conversations, as well as through programmatic APIs that expose the model for diverse tasks such as code generation [14], tool calling [24], and web search [13]. Portal-based interactions are often session-oriented and exhibit substantial prefix reuse across turns, whereas API requests are commonly single-turn but vary widely in prompt structure and reuse characteristics across tasks. This multiplicity of usage patterns results in highly diverse prefix reuse behaviors within a single serving system, posing fundamental challenges for prefix cache management.

Gap in existing eviction policies. However, today's deployed LLM serving systems still rely on simple recency-based heuristics for cache eviction, such as LRU, LFU, and First-In-First-Out (FIFO). Even worse, these heuristics are oblivious to workload heterogeneity, treating requests from diverse tasks and usage patterns uniformly. While recent work has started to incorporate workload signals, existing designs still treat different workloads in isolation and rely on the same eviction strategy within each workload. For example, WA handles multi-turn and single-turn requests independently. Based on this, it partitions requests by application category but applies LRU within each category. This design implicitly assumes that temporal locality is the dominant reuse signal, an assumption that does not necessarily hold across diverse task types with distinct prefix reuse behaviors.

A case study. To illustrate how workload heterogeneity affects eviction effectiveness, we consider two representative tasks under a fixed cache budget. (1) *ShareGPT* [25]: a multi-turn conversation task in which successive turns within a session share the conversation history. (2) *MASH-QA* [42]: a single-turn document question answering task, where requests are independent but often share stable, template-like prompt prefixes. We also blend these two tasks to create a mixed workload.

Fig. 1 compares the cache hit ratio achieved by different policies in these workloads. The results show that no single policy consistently performs the best across tasks. For ShareGPT, prefix reuse naturally arises across successive turns within the same session, as the chat history is concatenated into the prompt of each subsequent request. Consequently, recency-based policies, such as LRU, achieve high hit ratios, and WA behaves similarly in this setting. In contrast, LFU performs poorly, as long-running sessions accumulate high access counts, allowing stale KV states from completed sessions to persist in the cache. For MASH-QA, the reuse pattern is dominated by repeated, template-like prefixes across independent requests; here, LFU achieves the highest hit ratio by retaining frequently reused structural prefixes, whereas WA behaves the same as LRU.

Table 1 further reports the cache hit ratio breakdown for the mixed workload, illustrating that relying on a *single* reuse signal can be brittle under heterogeneous traffic. For example, LFU—which performs the best for MASH-QA-only workload—performs worst in the mixed workload. This is

because when multi-turn and single-turn traffic coexist, long conversational sessions can inflate access counts, causing the cache to retain blocks with high historical frequencies even after they become obsolete. Moreover, although WA achieves the highest total and ShareGPT task hit ratios, it becomes the worst for the MASH-QA task.

Summary. Prefix cache eviction under heterogeneous workloads is inherently challenging. Different tasks served by the same foundation model exhibit distinct prefix reuse mechanisms, making a single signal insufficient. This calls for a unified eviction policy that considers the task heterogeneity and achieves the global best prefix cache performance.

3 High-Fidelity Prefix Cache Simulation

Designing and evaluating effective prefix cache eviction policies requires exploring a large design space across diverse workloads and memory budgets. Conducting such experiments directly on real GPUs is prohibitively expensive and time-consuming. To enable rapid and reproducible evaluation, we build a high-fidelity prefix cache simulator that faithfully models KV cache allocation, prefix matching, and eviction behavior in modern LLM serving systems.

3.1 The Simulator Architecture

We build our simulator based on vLLM [33], the most popular open-source LLM inference engine. At a high level, we retain vLLM's CPU-side control plane, including request admission, scheduling, and KV cache bookkeeping, and replace GPU-side model execution with an output simulator that emits predefined tokens from traces.

Fig. 2 illustrates the original vLLM architecture and how we modify it into a high-fidelity prefix cache simulator. The core component of vLLM is an *LLMEngine*, which orchestrates a *Scheduler* running on CPUs and one or multiple *Workers* typically running on GPUs. The scheduler performs request admission, scheduling, and KV cache bookkeeping (e.g., KV block allocation and mapping). In particular, it records KV block IDs, the token IDs stored in the KV block, and a hash computed from the block's token content. The block hash serves as a compact fingerprint for identifying blocks and enables prefix matching during prefix-cache lookup. The worker executes the model on the GPU and comprises a *CacheEngine*, which manages KV cache in GPU memory, and a *ModelRunner*, which initializes the model and performs inference. The LLMEngine handles unfinished requests in a loop. At each inference step (`LLMEngine.step()`), the scheduler selects requests to advance and emits a *SchedulerOutput* describing the work for this iteration (e.g., sequences to process and the corresponding KV cache mappings). Based on this, the worker accesses cached KV states, executes a forward pass, and returns a *ModelOutput* to the scheduler to update per-request state.

To achieve high-fidelity simulation, we preserve the Scheduler but replace the GPU worker with an *OutputSimulator*. Instead of executing the model, the OutputSimulator produces *simulated output* using predefined request-level data from collected traces. This design preserves vLLM's scheduling and KV cache management logic while eliminating GPU execution, enabling fast and accurate prefix cache behavior simulation.

3.2 High-Fidelity Prefix Cache Simulation

Workload preparation. Our simulator is trace-driven and supports replaying collected LLM inference traces. Each request in our simulation is represented by its token sequence and per-request metadata, including task type, session ID, turn ID, and arrival timestamp. Following prior work [6, 9, 35], we generate session arrival times using a Poisson distribution with configurable arrival rates. Single-turn tasks are generated as independent requests, each forming a session with $turn\_id = 1$. Multi-turn tasks are generated at the session level: once a session starts, turns are

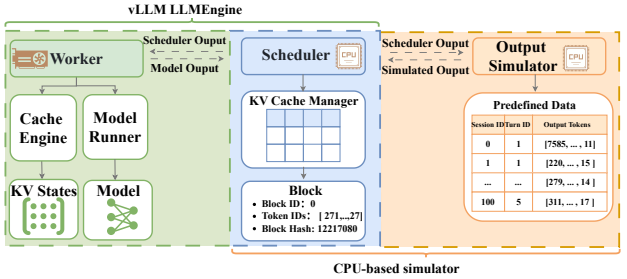


Fig. 2. **The architecture of vLLM and our CPU-based simulator. Left:** vLLM's LLMEngine schedules requests and KV cache on the CPU while executing the model on the GPU. **Right:** our simulator keeps the same scheduler but replaces GPU execution with an output simulator that replays outputs from traces.

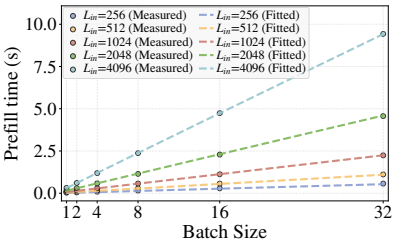


Fig. 3. **Power-law model fitting for prefill time across different configurations on A100.** Llama-3.2-1B: $prefill\_time = 5.56{\times}10^{-5} \cdot BS^{0.992} \cdot L_{in}^{1.034}$ with $R^2 = 0.999$, indicating excellent fit.

released sequentially by sampling per-turn inter-arrival times. All sessions share a single global KV cache capacity budget and therefore contend for the same cache space during replay.

Timing modeling and management. Accurate timing modeling is essential because it directly affects request admission (arrival timestamps), scheduling decisions, and timing-dependent eviction priority calculation. Our simulator adopts a *global virtual clock*: all events (e.g., request arrival, inference completion) are advanced precisely at their scheduled timing. One challenge here is how to model the inference time, which is related to the batch size and sequence length.

We solve this problem by modeling the prefill and decode time separately. For the *prefill* step, we fit a timing model using a power-law function of batch size and input length L_{in}: $prefill\_time = a \cdot \texttt{batch\_size}^{\,b} \cdot L_{\text{in}}^{c}$. We collect real prefill times on GPUs across a range of batch sizes and input lengths to estimate the parameters (a, b, c). Fig. 3 and Fig. 14 in the appendix show that the fitted models closely match real measurements across configurations on NVIDIA A100 and H100 GPUs, achieving near-perfect goodness-of-fit. For decode time, we model the average time per output token (i.e., TPOT) as a constant. Our profiling shows that TPOT exhibits relatively small magnitude but high variance across settings and does not admit a stable parametric fit. We therefore approximate TPOT using a constant equal to the measured mean. Our fidelity evaluation (§3.3) confirms that this approximation preserves cache-reuse behavior.

The simulation process. Algorithm 3 (in Appendix) shows the simulator's logic. It maintains a global virtual clock T and a min-heap-based event heap Q using turn-arrival timestamps as keys. We initialize Q by inserting the first turn of every session (lines 4–7). The simulator then iterates as follows. (1) It admits all requests in Q whose timestamps are $\leq T$ into the engine (lines 11–14). (2) If the engine has unfinished requests, the simulator invokes `LLMEngine.step()` to advance the system by one scheduling iteration. It then increments T by ΔT, computed using the fitted latency model (lines 15–19). (3) When a turn completes, the simulator inserts the next turn of the same session (if any) into Q with its scheduled arrival timestamp (lines 21–26). (4) If the engine is idle, the simulator fast-forwards T to the timestamp of the next event in Q, avoiding idle waiting and ensuring efficient simulation progress (lines 29–30).**The simulation process.** Algorithm 3 (in Appendix) shows the simulator's logic. It maintains a global virtual clock T and a min-heap-based event heap Q using turn-arrival timestamps as keys. We initialize Q by inserting the first turn of every session (lines 4–7). The simulator then iterates as follows. (1) It admits all requests in Q whose timestamps are $\leq T$ into the engine (lines 11–14). (2) If the engine has unfinished requests, the simulator invokes `LLMEngine.step()` to advance the system by one scheduling iteration. It

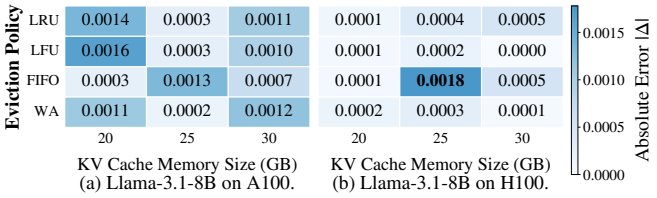


Fig. 4. **Simulator fidelity.** Absolute deviation in cache hit ratio (hit ratio in $[0, 1]$) between GPU execution and our simulator across cache budgets and policies.

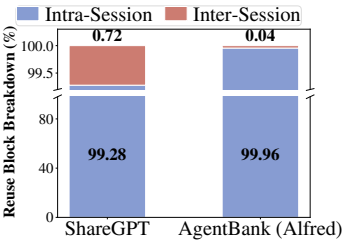


Fig. 5. **Intra-/inter-session reuse.** KV block reuse statistics across two multi-turn tasks.

then increments T by ΔT, computed using the fitted latency model (lines 15–19). (3) When a turn completes, the simulator inserts the next turn of the same session (if any) into Q with its scheduled arrival timestamp (lines 21–26). (4) If the engine is idle, the simulator fast-forwards T to the timestamp of the next event in Q, avoiding idle waiting and ensuring efficient simulation progress (lines 29–30).

3.3 Fidelity and Efficiency Evaluation

We evaluated the fidelity and efficiency of our simulator by comparing KV cache reuse under the same workload on real GPU execution and our simulation. To ensure both executions observe identical KV cache contents, we ran the GPU implementation with full model inference but *forced* token generation to follow the trace: at each decoding step, we overrode the sampled next-token ID with the corresponding output token from the trace. This produces the same token sequence (and thus identical KV states contents) in both systems while still executing vLLM's actual scheduling and cache-management logic on the GPU.

Under identical configurations for the model, cache budget, block size, and eviction policy, we collected cache hit ratios as an evaluation metric. Higher hit ratios indicate more prefix cache reuse, thus resulting in better performance. As shown in Fig. 4, across all configurations, the absolute hit-ratio difference is below 0.0018. We also compared end-to-end execution time for running the same trace. On average, GPU execution takes 4692 s, whereas the simulator completes in 150 s, yielding a ~30× speedup.

4 Prefix Reuse Analysis and Eviction Implications

In this section, we analyze prefix reuse across *seven* representative multi-turn and single-turn tasks, as summarized in Table 2. Our goal is to characterize the prefix reuse patterns and distill design insights for a unified prefix eviction policy.

4.1 Multi-Turn Tasks and Session Reuse

We first analyze multi-turn workloads, where an LLM interacts with a user or an autonomous agent over a sequence of dependent requests. In such workloads, requests are naturally grouped into *sessions*, where a session corresponds to an ordered sequence of turns generated within a single conversation, task, or agent trajectory. Our analysis here is based on four multi-turn traces, including chat conversations from ChatGPT [25] and Qwen [5], and agentic traces for embodied tasks [26] and vibe coding [39].

Observation 1: Reuse is primarily session-local. We first examine *where* prefix reuse arises by measuring KV block reuse at block granularity. Specifically, we quantify intra-session versus

Table 2. Datasets used in prefix reuse analysis.

Workload Type	Task Category	Dataset	Description
Multi-turn	Chat conversations	ShareGPT [25]	A large collection of user-shared ChatGPT conversations.
	Chat conversations	Qwen-Bailian (Trace A)[1]	An anonymized trace collected from a Qwen serving cluster deployed in production on Aliyun Bailian, capturing interactive chat traffic through cloud-hosted services.
	Agentic trajectories	AgentBank (ALFRED) [26]	Agentic trajectories for embodied task execution (ALFRED subset).
	Agentic trajectories	CC-Bench[1]	Agentic coding trajectories using Claude Code as the execution testbed.
Single-turn	Qwen API requests	Qwen-Bailian (Trace B)[1]	An anonymized trace from a production Qwen serving cluster on Aliyun Bailian, consisting of API-based single-turn requests.
	Untemplated tasks	AgentBank (Apps) [26]	Challenging competition mathematics problems.
	Programming	CodeParrot [14]	A code-generation benchmark with 10,000 problems for evaluating models that generate code from natural-language specifications.
	Tool use	ToolBench [8]	Real-world RESTful APIs spanning 49 categories (e.g., social media, e-commerce, and weather) for tool-augmented instruction following.
	Document QA	MASH-QA [42]	Non-factoid questions paired with consumer-health knowledge articles for long-document question answering.

inter-session KV block reuse on two representative multi-turn workloads: ShareGPT [25] and AgentBank (ALFRED subset) [26]. As shown in Fig. 5, reuse is overwhelmingly intra-session: inter-session reuse accounts for only 0.72% of reused blocks in ShareGPT and 0.044% in AgentBank. This behavior is expected because prefix caching requires *exact* token-level prefix matches, which are rare across independent sessions.

Implication 1: With negligible inter-session reuse, multi-turn KV block reuse is primarily driven by a session's next turn.

Observation 2: Reuse timing follows task-dependent log-normal distributions. Based on Implication 1, *when* reuse is determined by the *inter-turn interval*, which is the elapsed time between the arrivals of two consecutive turns within the same session. We measure the reuse interval of three workloads and find that it is highly task-dependent. Specifically, *chat-based* tasks involve a human in the loop and thus exhibit longer inter-turn intervals because users need to read and respond to model outputs. For example, for the Qwen-Bailian trace [5], a two-hour trace from a production Qwen serving cluster in Aliyun Bailian, the 80th percentile (P80) inter-turn interval is approximately 372 s (Fig. 6(a)). In contrast, *agentic* multi-turn tasks consist of autonomous agents executing many consecutive steps with minimal delay between turns. For example, for the CC-Bench trajectories trace [39], which contains complete agentic trajectories, inter-turn intervals are substantially shorter, with P80 around 12 s (Fig. 6(b)).

Despite the difference in timescales, the inter-turn interval distributions in both chat-based and agentic tasks can be well approximated by log-normal [4] models. As shown in Fig. 6(c) and (d), the fitted log-normal CDFs closely track the empirical CDFs over most of the probability mass, providing a compact statistical characterization of multi-turn timing behavior.

Implication 2: Given that reuse intervals vary across task types, eviction must be task-aware. The log-normal timing models provide task-specific reuse horizons that guide how long prefixes should be retained.

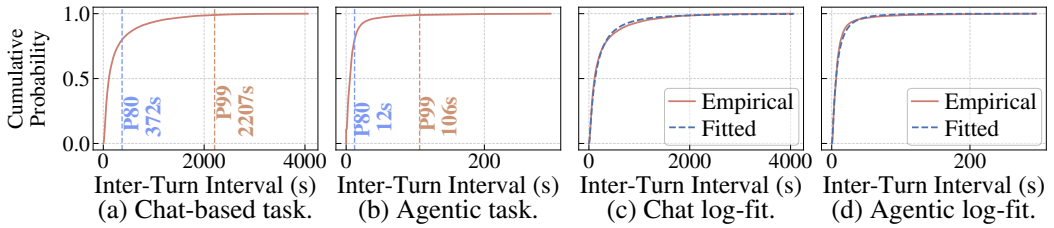


Fig. 6. Inter-turn time intervals for multi-turn workloads: (a)-(b) show interval CDFs from real traces, while (c)-(d) present the corresponding log-normal fits.

4.2 Single-Turn Tasks and Structural Reuse

We next analyze single-turn workloads, in which each request is processed independently without conversational state carried across turns. In these workloads, the model is typically invoked through stateless API calls to perform specific tasks, such as code generation, tool invocation, and document question answering, with each request providing a complete prompt and receiving a single response. We analyze various single-turn tasks from five datasets, including Qwen API traffic [5], math problems from AgentBank [26], programming tasks from CodeParrot [14], tool use requests from ToolBench [8], and document QA requests from MASH-QA [42].

Observation 3: Task-local structural reuse. Although single-turn workloads do not involve conversational sessions, they still exhibit prefix reuse. This reuse arises primarily from shared *prompt structure* across requests. As summarized in Table 3, many modern LLM applications employ task-specific prompt templates with recurring components, such as system instructions, tool specifications, or fixed contextual text. This structural regularity enables *structural reuse*: identical prompt segments reappear across independent requests, creating reusable prefixes.

Similar to multi-turn workloads, prefix reuse in single-turn workloads is also strongly task-local. This occurs because different tasks employ distinct prompt templates. As shown in Table 3, document QA tasks use prompts of the form *System prompt + Document + User query*, whereas tool-use tasks use *System prompt + Tool description + User query*. These task-specific components typically differ across tasks, making cross-task prefix reuse rare even when requests are issued concurrently. In Appendix A.2, we present two concrete prompt templates as examples.

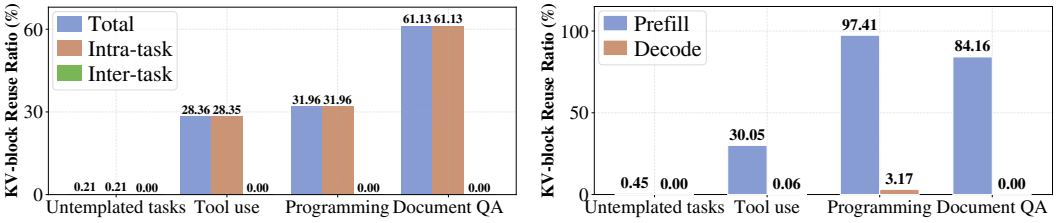
To quantify this effect, we measure task-local reuse, cross-task reuse, and total reuse across the four single-turn traces. The results are shown in Fig. 7a, which reveals two key observations. (1) The reuse ratio varies substantially across tasks and is largely determined by prompt structure. For example, in untemplated (open-ended) tasks, the user query follows the system prompt directly, leaving little shared prefix across requests, whereas tasks with fixed components, such as tool descriptions or program specifications, exhibit much higher reuse. (2) Reuse is overwhelmingly task-local: cross-task reuse is negligible, accounting for less than 0.01% of reused blocks.

Implication 3: Different single-turn tasks should be managed independently, with eviction decisions driven primarily by their shared prompt structure.

Observation 4: Decode token reuse is negligible. We further find that *within a task*, reuse is dominated by prefill-phase tokens, while reuse of decode-phase tokens is essentially negligible. Reusing decode tokens requires all previously generated tokens to match exactly. In practice, any variation in the user query breaks this constraint. Moreover, most LLMs employ temperature-based decoding, which samples tokens stochastically from the model's output distribution to promote response diversity and avoid deterministic repetition. Therefore, even when two requests share an

Table 3. Request structure of single-turn tasks.

Task	Dataset	Request Structure
Untemplated tasks	AgentBank	System prompt + User query
Programming	CodeParrot	System prompt + Program description
Tool use	ToolBench	System prompt + Tool description + User query
Document QA	MASH-QA	System prompt + Document + User query



(a) Reuse locality breakdown.

(b) Prefill vs. decode comparison.

Fig. 7. **Analysis of KV-block reuse characteristics across different tasks.** (a) Reuse locality breakdown by task: *Total* denotes all KV-block recurrences, *Intra-task* denotes reuse within the same task type, and *Inter-task* denotes reuse shared across ≥2 task types. (b) Comparison between prefill and decode stage reuse ratios. Experimental setup: block size = 16, number of GPU KV blocks = ∞, and 1,800 requests per task.

identical prefill prompt, their outputs are very likely different. This pattern is confirmed in Fig. 7b: while prefill blocks exhibit substantial reuse across all tasks, decode-block reuse remains near zero (at most 3.17%).

Implication 4: Because decode-phase KV states are rarely reused, they should be assigned the highest eviction priority and reclaimed aggressively in favor of more valuable prefill prefixes.

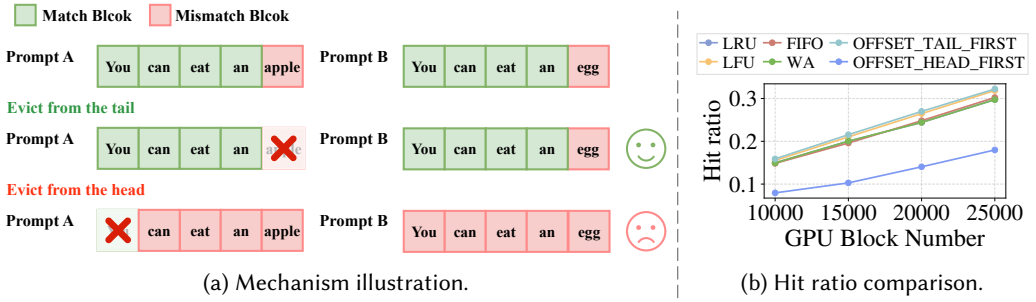
Observation 5: Positional reuse probability. Finally, we observe that *within a single prompt*, the reuse of prefill KV blocks is strongly correlated with their position in the prompt: blocks appearing later in the prompt are less likely to be reused. This follows directly from strict prefix matching: a later block can be reused only if *all* preceding tokens are also identical. As prompts grow, the probability that two requests share an exact prefix decreases with depth into the prompt. In practice, early tokens—such as system prompts or fixed task instructions—are often shared across many requests, whereas later tokens—such as tool descriptions, documents, or user queries—are more variable. Consequently, blocks closer to the beginning of the prompt have significantly higher reuse potential than those near the end. Fig. 8a illustrates how mismatches at different positions affect prefix reuse, and Fig. 8b quantifies the resulting impact on prefix cache hit ratio.

Implication 5: Prefix cache eviction should be *position-aware*, prioritizing the retention of earlier prompt blocks and evicting later blocks first.

4.3 The Optimal Eviction Policy

To better evaluate eviction policies, we establish an upper bound on prefix cache reuse. Specifically, we derive an *optimal* eviction policy that assumes perfect knowledge of future accesses, analogous to Belady's optimal page replacement (OPT) in operating systems [3]. Although such a policy is not implementable in practice, it provides a gold standard against which we can quantify how much performance is lost due to limited information and heuristic design in real systems.

Offline OPT. Given a cache budget, OPT evicts the KV block whose *next access* occurs farthest in the future (or that is never accessed again). We compute this offline upper bound using a two-pass



(a) Mechanism illustration. (b) Hit ratio comparison.

Fig. 8. **Impact of eviction position on KV cache reuse.** (a) Qualitative analysis showing why evicting from the tail preserves structural prefix sharing, while evicting from the head breaks it. (b) Quantitative results on MASH-QA under different block number constraints, comparing two eviction variants: *OFFSET_TAIL_FIRST* (evict tail blocks first) and *OFFSET_HEAD_FIRST* (evict head blocks first).

procedure. In the first pass, we replay the trace to record the KV block access stream, i.e., the ordered sequence of engine iterations in which each block is referenced. The relative order of accesses is sufficient to determine the next reuse distance. In the second pass, we run OPT using this access information to compute the resulting cache hit ratio, which serves as an offline upper bound for the same trace and KV cache budget.

5 Unified Eviction Policy for Heterogeneous LLM Serving Workloads

Inspired by the design implications derived in the previous section, we propose the first unified prefix cache eviction policy for heterogeneous LLM inference workloads. Our policy explicitly captures different reuse behaviors across workload types while coordinating their competition for shared KV cache capacity within a unified eviction framework.

5.1 Algorithm Overview

Implications 2 and 3 from Section 4 motivate the need for task-aware cache management: eviction decisions should distinguish among different task types rather than treating all KV cache blocks uniformly. In practice, task types can be inferred from request metadata, such as prompt semantics or the API endpoint used (e.g., distinguishing web-portal traffic from API calls). The key challenge, however, is how to coordinate eviction across tasks within a shared cache. Our key insight is that, to maximize overall cache performance, capacity should be preferentially allocated to task classes that provide higher reuse benefit per unit of memory.

Inspired by this, we adopt a *two-stage design*. (1) We maintain separate queues for blocks with distinct reuse characteristics and impose a task-specific ordering within each queue (Section 5.2). (2) We apply a global weighting mechanism that adjusts eviction priority across queues based on their observed hit efficiency (Section 5.3). Fig. 9 provides an overview of the eviction workflow. At a high level, the eviction proceeds as follows (Algorithm 1). First, we exhaust the *evict-first* queue, which contains blocks with negligible reuse potential (lines 5–8). For the remaining queues, we select the top candidate from each queue based on its internal ordering rule. For a candidate from queue q, we compute a local priority score s_q. Then, we derive a globally comparable score

$$G_q = \alpha_q \times s_q,$$

where α_q is a queue-level weight capturing the recent hit efficiency of queue q (lines 9–14). Queue weights are updated periodically based on runtime statistics, so that queues yielding more cache hits per unit capacity are penalized less during eviction. Finally, the block with the lowest G_q

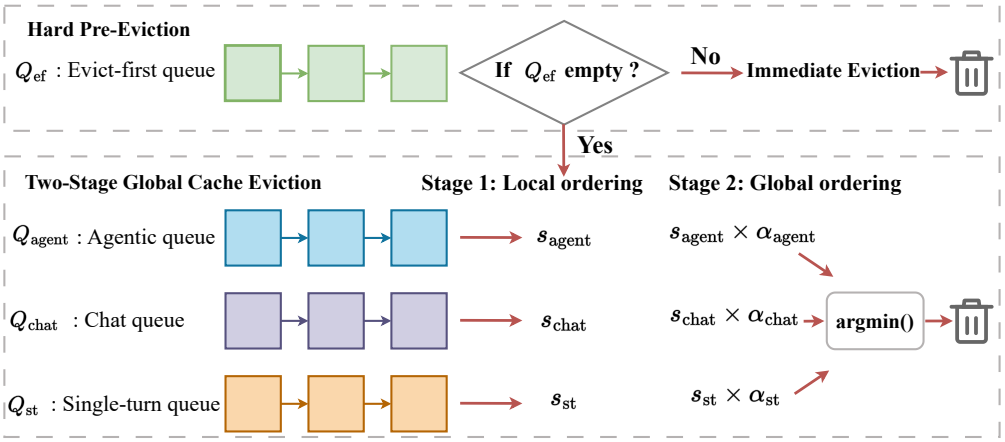


Fig. 9. **Overview of the UniCache eviction workflow.**

is selected as the eviction victim, while non-selected candidates are returned to their respective queues.

5.2 Task-Aware Multi-Queue Organization

KV cache blocks in UniCache are managed by a multi-queue architecture, where blocks with distinct reuse patterns are assigned to separate queues. The key idea here is to separate blocks whose reuse behavior is driven by different signals, allowing each queue to maintain an eviction order aligned with its reuse characteristic. Specifically, we partition KV blocks into four queues.

Evict-first queue. This queue holds blocks with low reuse potential, including (1) prefill blocks from single-turn, untemplated tasks (*Implication 3*) and (2) all decode-phase blocks from single-turn tasks (*Implication 4*). As shown in Section 4.2, these blocks are rarely reused due to strict prefix matching and decoding randomness. Within this queue, blocks are ordered by positional offset, and eviction proceeds from the largest offset to the smallest (*Implication 5*), preserving early-prefix tokens whenever possible. Importantly, this queue serves as a hard pre-eviction stage: eviction always drains the evict-first queue before considering any other queue.

Session-reuse queues (agentic and chat). We maintain two separate queues for multi-turn workloads—an *agentic* queue and a *chat* queue—because different task types exhibit distinct inter-turn interval distributions (*Implications 1 and 2*). As shown in Section 4.1, the inter-turn interval Δ in both workloads can be well modeled by a log-normal distribution. In practice, this distribution can be fitted using historical traces (e.g., data from prior days or from the same weekday in previous weeks), yielding the cumulative distribution function $IntervalCDF(\cdot)$.

Under the OPT eviction principle, a KV block should be retained if it is expected to be reused in the near future. In multi-turn sessions, a cached block can only be reused when a subsequent turn from the *same session* arrives. Let t_0 denote the block's last access time, Δ the inter-turn interval, and t_{now} the current time. We define the elapsed time since the last access as $t = t_{\text{now}} - t_0$. Then, $IntervalCDF(t) = \Pr(\Delta \le t)$ is the probability that the next turn *has already arrived* within time t, whereas $1 - IntervalCDF(t) = \Pr(\Delta > t)$ is the probability that the next turn has *not yet arrived*. A larger value of $1 - IntervalCDF(t)$ therefore indicates a lower likelihood of near-term reuse. Following the OPT intuition of retaining blocks that are more likely to be reused soon, we define

Algorithm 1 Two-Stage Global Cache Eviction

Require:
$\mathcal{Q}_{ef}$: Evict-first queue,
$\mathcal{Q}_{mt} = \{\mathcal{Q}_{agent}, \mathcal{Q}_{chat}\}$: Multi-turn queue,
$\mathcal{Q}_{st}$: Single-turn queue,
$\boldsymbol{\alpha}$: Current weight vector $\{\alpha_q \mid q \in \mathcal{Q}_{mt} \cup \mathcal{Q}_{st}\}$
Ensure: b_{victim}: The selected block for eviction
if $\mathcal{Q}_{ef}$ is not empty **then**
  return $b \leftarrow$ pop_candidate($\mathcal{Q}_{ef}$)
end if
$C \leftarrow \emptyset$ ▷ Candidate set for weighted competition
for all $q \in \mathcal{Q}_{mt} \cup \mathcal{Q}_{st}$ **do**
  $b_q \leftarrow$ peek_top(q)
  $s_q \leftarrow$ CalculateLocalPriority(b_q, q)
  $G_q \leftarrow \alpha_q \times s_q$
  $C \leftarrow C \cup \{(b_q, G_q)\}$
end for
$b_{victim} \leftarrow \text{argmin}_{b \in C}(G_q)$
return b_{victim}

function CalculateLocalPriority(b, q)
  if $q \in \mathcal{Q}_{mt}$ **then**
    $t \leftarrow t_{now} - t_{last\_access}$
    return $1 - IntervalCDF_q(t)$
  else ▷ $q \in \mathcal{Q}_{st}$
    return $1 - (offset/max\_offset)$
  end if
end function

Algorithm 2 Dynamic Queue Weight Update

Require:
$\mathcal{Q}$: Set of queues,
$\mathbf{H} = \{H_q\}_{q \in \mathcal{Q}}$: Cumulative hit-token counts,
$\mathbf{C} = \{C_q\}_{q \in \mathcal{Q}}$: Allocated cache capacity fractions,
$\boldsymbol{\alpha}_{prev}$: Previous weight vector,
β: EMA smoothing factor ($0 < \beta < 1$),
T: Temperature parameter,
ϵ: Numerical constant.
Ensure: Updated weight vector $\boldsymbol{\alpha}_{new}$
// **Step 1: Hit Efficiency Calculation**
for all $q \in \mathcal{Q}$ **do**
  $E_q \leftarrow \frac{H_q}{C_q + \epsilon}$ ▷ Calculate E for each queue
end for
$\bar{E} \leftarrow \text{mean}(\{E_q\}) + \epsilon$
// **Step 2: Relative Scaling & Clipping**
for all $q \in \mathcal{Q}$ **do**
  $\hat{E}_q \leftarrow (E_q / \bar{E})^{1/T}$ ▷ Relative efficiency scaling
end for
$\mu_{\hat{E}}, \sigma_{\hat{E}} \leftarrow \text{mean}(\{\hat{E}_q\}), \text{std}(\{\hat{E}_q\})$
$[L, U] \leftarrow [\max(0.001, \mu_{\hat{E}} - 2\sigma_{\hat{E}}), \min(10.0, \mu_{\hat{E}} + 2\sigma_{\hat{E}})]$
▷ Dynamic clipping bounds
// **Step 3: Weight Refinement via EMA**
for all $q \in \mathcal{Q}$ **do**
  $\alpha_{raw} \leftarrow \beta \cdot \alpha_{prev}[q] + (1 - \beta) \cdot \hat{E}_q$
  $\alpha_{new}[q] \leftarrow \text{clamp}(\alpha_{raw}, L, U)$
end for
return $\boldsymbol{\alpha}_{new}$

the eviction priority for multi-turn blocks as:

$$s_q = 1 - IntervalCDF(t), \quad t = t_{\text{now}} - t_0.$$

Within each multi-turn queue, blocks are maintained in recency order (by t_0), and s is used to select eviction candidates across queues.

Structural-reuse queue. This queue contains KV blocks from single-turn tasks with structured prompts, such as tool use and program generation. As shown in Section 4.2 (*Implication 5*), the reuse value of a block in such workloads is dominated by its position in the prompt: evicting an early block invalidates all subsequent tokens due to strict prefix dependencies. Accordingly, blocks in this queue are ordered by their positional offset, and eviction proceeds from the largest offset to the smallest. We define the eviction priority as a normalized score in the range $[0, 1]$:

$$s_q = 1 - \frac{offset}{max\_offset},$$

where *offset* denotes the block's positional offset within the prompt, and *max_offset* is the maximum offset among all blocks. Under this definition, later blocks receive smaller priority scores and are evicted earlier, while earlier blocks are retained to preserve the structural reuse.

5.3 Global Hit-Efficiency Weighting

While each queue maintains a reuse-aware local ordering, eviction decisions must still be coordinated across queues whose priority scores are not directly comparable (e.g., time-based versus

position-based metrics). To this end, we introduce a global *hit-efficiency* weighting mechanism that quantifies how effectively each queue converts cache capacity into cache hits.

For each queue q, we compute its hit efficiency over a recent time window as

$$E_q = \frac{H_q}{C_q},$$

where H_q denotes the number of cache-hit tokens attributed to queue q, and C_q denotes the fraction of total KV cache capacity currently occupied by that queue (i.e., its normalized queue size). A higher value of E_q indicates that queue q yields more cache hits per unit of capacity, while a low value of E_q suggests inefficient cache utilization.

We then derive a queue-specific scaling factor α_q from E_q and use it to obtain a globally comparable eviction score. Specifically, for a candidate block selected from the queue q with local priority s_q, we compute $G_q = \alpha_q \times s_q$. Intuitively, queues with higher hit efficiency receive larger α_q, reducing the likelihood that their blocks are evicted when competing with other queues. Conversely, queues with lower E_q receive smaller α_q and are preferentially evicted. Across queues, eviction selects the candidate with the lowest G_q. The update rule for α_q is detailed in Algorithm 2.

6 Evaluation

We implemented a prototype of UniCache on top of vLLM v0.8.5 [33]. Our implementation modifies only the prefix cache eviction logic, making it lightweight and easily portable to other inference engines such as SGLang [41]. In this section, we comprehensively evaluate the effectiveness of UniCache against state-of-the-art eviction policies under heterogeneous LLM inference workloads. The code is available at https://github.com/xsyslab/UniCache.

6.1 Setup

Baselines. We compare UniCache against the following baselines, all using the default KV block size of 16 tokens:

- LRU: Evicts the least recently accessed KV block, favoring blocks that have been referenced more recently.
- LFU: Evicts the KV block with the lowest access frequency.
- FIFO: Evicts KV blocks in the order they were inserted into the cache.
- LeCaR [31]: A learning-based cache eviction policy that uses reinforcement learning and regret minimization to dynamically balance recency- and frequency-based eviction strategies.
- ARC [12]: An adaptive eviction policy that continuously balances recency and frequency in an online, self-tuning manner.
- WA [34]: A workload-aware cache eviction policy that applies LRU within each workload.
- OPT-Offline: An offline optimal policy with perfect knowledge of future reuse, serving as an upper bound of cache hit ratio under a given cache budget.

Testbed and models. We conducted all experiments in cloud instances equipped with NVIDIA A6000 (48 GB), A100 (80 GB), and H200 (140 GB) GPUs. Each instance provides 240 GB of system memory and 60 vCPUs, and runs Ubuntu 22.04 with CUDA 12.8. We evaluated three representative LLMs spanning small to medium scales: Llama-3.2-1B, Llama-3.1-3B, and Llama-3.1-8B. We did not evaluate larger models because prefix cache behavior is determined by prefix reuse patterns and cache capacity, not by model size; larger models merely scale the per-token KV footprint without changing the relative effectiveness of eviction policies. We conducted latency experiments (Section 6.4) on real GPUs, while the remaining results were obtained from our simulator. Before

running simulations, we calibrated the simulator against each GPU to match measured latency under the corresponding hardware configuration.

Trace construction. Our evaluation requires request traces that include both the request content (to determine prefix reuse) and realistic timestamps (to capture queueing dynamics). To the best of our knowledge, no public dataset provides both at the granularity needed for LLM serving. We therefore synthesized timestamped traces by pairing content-only task datasets with timestamped traces (Table 2).

We modeled two time scales: *session arrivals* (when a new conversation starts) and *within-session inter-turn gaps* (time between consecutive turns in the same session). Session start times follow a Poisson process, i.e., inter-session start times are exponentially distributed, which is a standard assumption for independent arrivals adopted by many prior studies [6, 9, 37]. For multi-turn chat sessions, we calibrated the within-session inter-turn gap distribution using Qwen-Bailian (Trace A). We fit a log-normal distribution to the observed inter-turn gaps, where $\log \Delta t \sim \mathcal{N}(\mu, \sigma^2)$, obtaining $\mu = 4.15$ and $\sigma = 0.971$ (time unit in seconds). For multi-turn agentic workloads, we fit an analogous model using the CC-Bench dataset, yielding $\mu = 1.81$ and $\sigma = 1.092$. Section 4.1 (Observation 2) has discussed the effectiveness of this modeling.

Given these models, we sampled session start times from the Poisson process, then sampled within-session turn times from the corresponding log-normal distribution, and finally assigned each turn a task instance from the trace request content. For single-turn tasks, each session contains exactly one request; we therefore omitted within-session gap sampling and directly placed the request at the sampled session start time.

Heterogeneous workload construction. To evaluate system behavior under different prefix-reuse scenarios, we constructed three heterogeneous workloads by varying the proportion of multi-turn and single-turn workloads: (1) *Multi-turn–dominant* (Trace #1), where approximately 80% of requests are multi-turn; (2) *Balanced* (Trace #2), with an even split between multi-turn and single-turn requests; and (3) *Single-turn–dominant* (Trace #3), where approximately 80% of requests are single-turn. Each trace spans approximately one hour. The multi-turn workload comprises two task types (multi-turn chat and multi-turn agentic interactions), while the single-turn workload comprises four task types (untemplated task, programming, tool use, and document QA).

Evaluation metrics. We focus on cache effectiveness and latency metrics relevant to LLM serving. (1) *Hit Ratio.* Hit ratio is the fraction of *prefill* tokens that reuse cached KV states rather than being recomputed. For a trace with N total prefill tokens, a hit ratio of $H\%$ means that $H\%$ of prefill tokens are served from the cache. (2) *Queued TTFT (QTTFT).* QTTFT captures user-perceived latency under load. QTTFT is defined as the time from request arrival until GPU execution begins, plus the time to generate the first output token. Prefix caching does not impact the decode phase, where KV states are generated incrementally for newly produced tokens, so we do not report TPOT.

6.2 Prefix Hit Ratio Under Heterogeneous Workloads

We first evaluate the effectiveness of UniCache in terms of prefix hit ratio under heterogeneous workloads. We conducted experiments using three models across three traces on an NVIDIA A100 (80GB) GPU. We defined the KV cache budget as a fraction of the available GPU memory A after loading model weights, and swept the budget from $0.5A$ to $0.9A$. Fig. 10 reports the hit ratio under different KV cache budgets, models, and workload traces. Across all evaluated configurations,

UniCache (the red line) consistently achieves the highest hit ratio, outperforming all *online* baselines by an average of 3.86%–17.32%. [1] We provide a more detailed analysis in the following.

Different workloads. Workload composition has a strong impact on the effectiveness of cache eviction policies. Specifically, Trace #1 (Fig. 10(a,d,g)) is dominated by multi-turn traffic and exhibits strong temporal locality, as successive turns reuse the growing conversation history. In this scenario, recency-based policies perform well: at a cache budget of 0.7A for Llama-3.2-1B, LRU and FIFO achieve hit ratios of approximately 91.64% and 91.19%, respectively, approaching UniCache at 92.32%. In contrast, LFU performs substantially worse (79.42%), as long-running sessions inflate access counts and cause stale KV states from completed sessions to remain in the cache. As the workload shifts toward more single-turn traffic (Trace #2 and #3 in Fig. 10(b,c)), temporal locality weakens and reuse is increasingly driven by structural reuse. In this regime, LFU becomes more competitive for the 1B model, reducing its performance gap to other baselines by an average of 12.22% in Trace #2 and 15.82% in Trace #3.

This effect is further amplified by the larger effective cache budget (0.9A) of the 1B model, which alleviates memory pressure and partially offsets LFU's tendency to retain high-frequency yet stale prefixes. In contrast, purely recency-based policies degrade as temporal locality diminishes, dropping by 16.66%–20.36% between Trace #1 and Trace #3 on average. Across these workload settings, UniCache consistently achieves the best hit ratio among *online* policies by being task-aware: it captures different reuse patterns across task types and coordinates eviction decisions across heterogeneous workloads.

Compared with the adaptive baselines LeCaR and ARC, UniCache still delivers the best performance, outperforming them by 8.19% and 3.86% on average, respectively. This result is notable because LeCaR and ARC already adapt between recency- and frequency-based signals, which are generally effective for cache management. However, under heterogeneous LLM workloads, eviction decisions are governed not only by generic temporal locality, but also by task-specific prefix reuse patterns. Since LeCaR and ARC are not designed to capture such workload-dependent reuse behaviors, they cannot accurately preserve the KV blocks that are most valuable for future reuse. In contrast, UniCache explicitly incorporates task-aware reuse characteristics into eviction decisions, which allows it to achieve consistently higher cache efficiency.

Different models. As the model size scales from 1B to 8B parameters, the overall prefix cache hit ratio decreases across all policies. This is because larger models leave less GPU memory available for the KV cache after loading model weights and also impose a larger per-token KV states footprint, increasing memory pressure and reducing the number of prefixes that can be retained. Despite this increased pressure, UniCache maintains a clear advantage over all baseline policies. Across all three traces at 0.7A, UniCache outperforms LRU by 2.27%–4.63% and LFU by 4.99%–25.95% on the 8B model. These results indicate that UniCache utilizes limited GPU memory more effectively than recency- or frequency-based eviction policies, achieving higher cache efficiency under stringent memory constraints.

Different cache sizes. Across all traces and models, hit ratios increase monotonically with the KV cache budget, since a larger cache can retain more reusable prefixes. Among all policies, UniCache consistently outperforms standard baselines such as LRU, LFU, and FIFO, and remains close to the OPT-Offline upper bound across all cache sizes (Fig. 10). In most cases, the hit ratio grows approximately linearly with cache size. However, some policies, most notably LFU in Fig. 10(b) and (c), exhibit diminishing returns when the budget increases from 0.7A to 0.9A, compared with

[1] It is worth noting that even small improvements in prefix cache hit ratio translate into substantial GPU compute savings at scale, given the massive volume of requests served by modern LLM systems. For example, WA [34] (ATC'25) reports only a small improvement in cache hit ratio over LRU (1.5-3.9%) but is still considered practically meaningful.

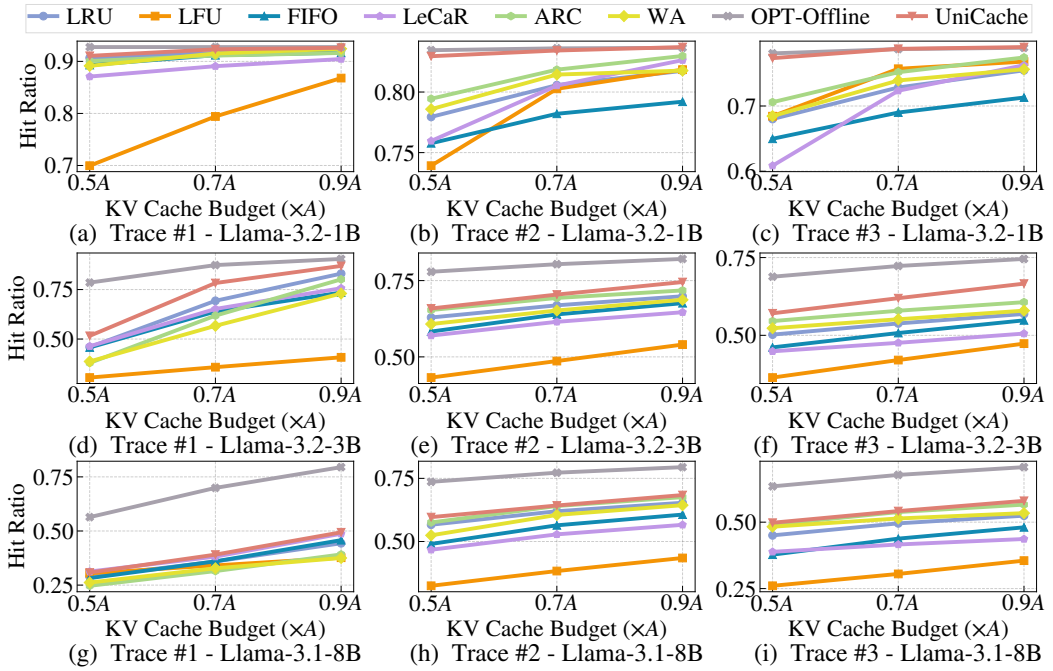


Fig. 10. **Hit ratio across different models, workloads, and KV cache budget.** The x-axis shows the KV cache budget as a fraction of the available GPU memory A after loading model weights.

the increase from $0.5A$ to $0.7A$. This is because their hit ratios have already approached the upper bound, leaving limited room for further improvement.

6.3 Prefix Hit Ratio Across GPU Types

LLMs can be deployed across a wide range of GPUs with substantially different memory capacities. To understand how prefix cache eviction policies perform under these varying hardware constraints, we evaluated hit ratio across three tiers of GPU types, A6000, A100, and H200, as summarized in Table 4. For each GPU, we allocated the remaining device memory after loading model weights to the KV cache.

Fig. 11 reports the hit ratio for Llama-3.2-3B for the three GPU platforms and three traces. Across all traces and eviction policies, the hit ratio increases monotonically from A6000 to A100 and further to H200. This trend is directly attributable to the larger memory capacity of higher-tier GPUs, which provides a larger KV cache budget and allows more KV blocks to be retained. Importantly, changing GPU types does not alter the relative behavior of eviction policies: OPT-Offline consistently defines the theoretical upper bound, while UniCache remains the best-performing *online* eviction policy across all platforms.

Moreover, the performance gain of UniCache over other online baselines varies with trace characteristics. On Trace #1, the improvement of UniCache ranges from 0.38% to 44.66% across GPU types. The gain saturates on high-memory GPUs such as H200 because Trace #1 is dominated by multi-turn traffic and exhibits strong temporal locality. In this setting, recency-aware heuristics already approach the optimal bound as the KV cache budget increases, leaving limited headroom for further improvement beyond UniCache. In contrast, the advantages of UniCache are more

Table 4. GPU memory and KV cache budget for Llama-3.2-3B.

GPU	Mem (GB)	KV cache (GB)
A6000	48	35
A100	80	63
H200	140	117

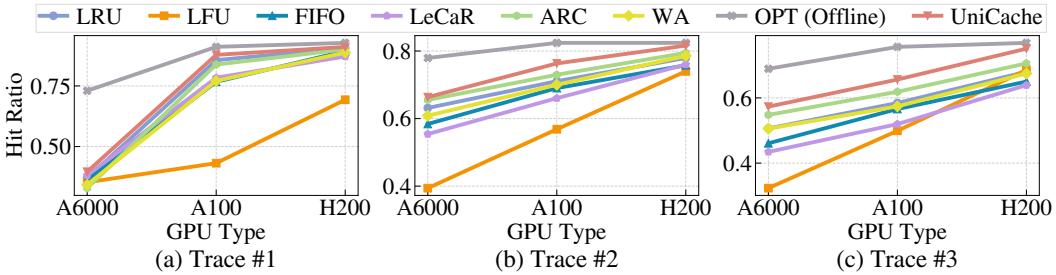


Fig. 11. **Hit ratio across GPU types for Llama-3.2-3B.** Each subplot corresponds to one trace: (a) Trace #1, (b) Trace #2, and (c) Trace #3. For each GPU (A6000, A100, H200), we allocate the remaining device memory after loading model weights to KV cache (Table 4).

Table 5. **Average QTTFT (s)**. UniCache consistently outperforms other online baselines and approaches the *offline* oracle.

	LRU	LFU	FIFO	LeCaR	ARC	WA	OPT (Offline)	UniCache
Trace#1	4.94	8.47	6.87	7.42	4.54	5.14	3.84	3.87
Trace#2	0.28	0.53	0.36	0.47	0.26	0.29	0.21	0.24
Trace#3	0.13	0.37	0.25	0.35	0.12	0.27	0.08	0.10

pronounced on Trace #2 and Trace #3. Across the three GPU platforms, UniCache improves hit ratio by 0.83%–26.94% on Trace #2 and by 6.54%–24.99% on Trace #3. These results show that UniCache is robust across diverse GPU architectures and workload compositions, consistently delivering high cache efficiency.

6.4 QTTFT Improvement

The effectiveness of prefix caching directly affects inference latency: higher cache hit ratios reduce prefill computation and increase throughput, which in turn lowers request queueing delay. Therefore, in this subsection, we evaluate the benefits of UniCache using QTTFT, a metric that captures both prefill latency and queueing delay. All experiments are conducted on a single NVIDIA A100 GPU using Llama-3.1-3B on three traces, with the KV cache budget fixed at $0.9A$.

Table 5 reports the average QTTFT achieved by different cache eviction policies. Across all three traces, UniCache consistently achieves QTTFT close to the offline OPT bound. Relative to existing *online* baselines, UniCache reduces average QTTFT by 1.10×–3.63×, indicating substantial end-to-end latency reduction through more effective prefix reuse. At the same time, UniCache attains 77.90%–99.28% of the offline optimal performance.

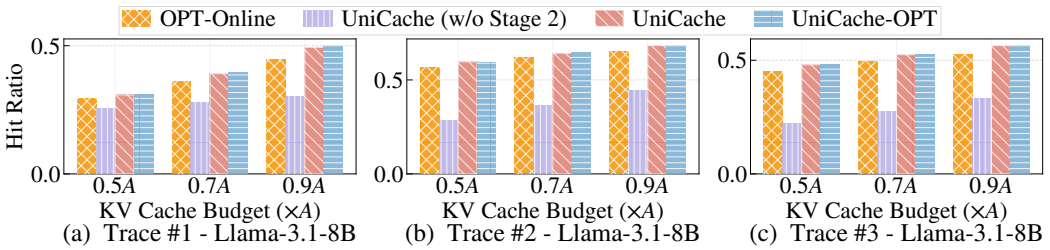


Fig. 12. Ablation results on Llama-3.1-8B, evaluating the impact of Stage 2 cross-queue coordination and limited lookahead.

6.5 Ablation Study

Finally, we evaluate the contribution of individual design components through an ablation study on Llama-3.1-8B.

Stage 2 global ordering. UniCache adopts a two-stage eviction design: Stage 1 selects a task-specific eviction candidate within each queue, and Stage 2 coordinates the final eviction decision across queues. To evaluate the importance of Stage 2, we compare two variants: (1) UniCache without Stage 2 (*w/o Stage 2*), which evicts candidates selected independently within each queue; and (2) the full UniCache design.

As shown in Fig. 12, UniCache (w/o Stage 2) performs substantially worse: the full UniCache improves hit ratio by 5.36%–31.30% over UniCache (w/o Stage 2). This gap arises because Stage 1 establishes only *local* priorities based on queue-specific reuse signals, which are not directly comparable across queues corresponding to different workload classes.

Without Stage 2, the system lacks a principled mechanism for ranking eviction candidates across tasks and therefore cannot adapt effectively to changes in workload composition. By performing cross-queue selection using globally comparable scores, Stage 2 enables UniCache to make coordinated eviction decisions under heterogeneous workloads.

OPT enhancement. We derive OPT-Online, a practical variant of OPT-Offline that exploits the limited lookahead available in real serving systems. The key observation is that the serving engine maintains a waiting queue of requests that have *already arrived* but have not yet been processed. These pending requests provide partial next-use information: by scanning the waiting queue, we can identify a subset of KV blocks that will be referenced in the near future. OPT-Online orders cached blocks according to their estimated next-use time inferred *only* from requests currently in the waiting queue. Upon eviction, it removes the block whose estimated next use is farthest in the future. For blocks whose next use cannot be inferred, OPT-Online conservatively assigns a next-use time of $+\infty$; when multiple blocks tie (e.g., all have unknown next use), it breaks ties by last-access time, which reduces to LRU.

We compare three policies here: (1) the base UniCache; (2) UniCache-OPT, which augments UniCache with OPT-Online next-use estimates when available and falls back to UniCache's when next-use information is missing or tied; and (3) OPT-Online. UniCache-OPT improves hit ratio over UniCache by up to 0.74%. This gain depends on how much lookahead is exposed by the waiting queue, which is jointly determined by the engine's processing rate and workload intensity. For Trace #1, a heavier load yields a deeper waiting queue and thus richer next-use signals, leading to the largest improvement. Compared with OPT-Online, UniCache achieves a 1.53%–4.90% higher hit ratio, indicating that task-aware cross-queue coordination provides benefits beyond what can be obtained from limited lookahead alone.

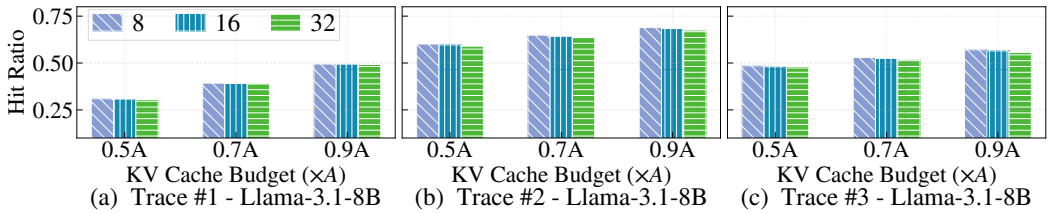


Fig. 13. Impact of KV cache block size on prefix-cache hit ratio for UniCache. We vary the block size (8/16/32 tokens) on Llama-3.1-8B across three traces.

Impact of KV cache block size. We evaluate the sensitivity of UniCache to the KV cache block size by varying the block size among 8, 16, and 32 tokens on Llama-3.1-8B across three traces. Since prefix reuse operates at block granularity, only fully matched blocks can be reused. As a result, larger blocks increase internal fragmentation: partially reusable prefixes often fail to form complete blocks and therefore cannot be served from the cache. Because the total KV cache budget is fixed, changing the block size only modestly affects the effective cache capacity; consequently, the observed differences are small. On average, a block size of 8 improves hit ratio by 0.39% over a block size of 16 and by 0.94% over a block size of 32.

6.6 Discussion

Impact of model type and size. UniCache is largely agnostic to model type and size. Its effectiveness is determined primarily by the cross-request prefix reuse and cache capacity, rather than by the specific model architecture or parameter scale. Specifically, a prefix-cache hit occurs when multiple requests share reusable prompt prefixes, which is independent of the model architecture or computation graph. The eviction policy only determines whether previously computed KV states remain in the cache and can be reused later. It does not change the semantics of model execution: if an evicted KV block is needed again, the corresponding KV states are simply recomputed on demand. Model size mainly affects the per-token KV states footprint, which in turn determines how many KV blocks can be retained under a fixed memory budget.

Distributed inference. UniCache naturally supports distributed inference without requiring implementation changes. When running distributed inference using systems like vLLM, although the underlying KV states are physically distributed across multiple GPUs, a single *KVCacheManager* manages KV block allocation and eviction, as illustrated in Fig. 2. The *KVCacheManager* operates on logical block-level metadata, such as block IDs and token IDs, rather than on the physical tensor placement, which decouples the eviction policy from specific layer mappings or model placement schemes. In our design, the UniCache queues are centrally maintained by the *KVCacheManager* on CPUs. Consequently, distributed execution does not affect the eviction design.

Task type classification. In practice, UniCache does not rely on deep semantic understanding of prompts. Instead, task categories can often be inferred from lightweight signals already available in serving systems, such as request metadata. For example, public traces from Qwen-based serving workloads [5] include fields such as “type”, which can be directly used for task classification.

7 Related Work

Prefix caching. Many recent systems [17, 23, 32, 41] have been proposed to exploit prefix reuse opportunities in LLM inference. These systems primarily cache and reuse prefill KV states when requests share common prompt prefixes, reducing redundant computation within a single serving

engine. CachedAttention [6] and Pensieve [38] further extend this idea by introducing hierarchical KV caching across multiple memory and storage tiers, enabling KV states to be retained beyond GPU memory at lower cost. PREBLE [27] and follow-up work [2] study cluster-level prompt sharing, where request scheduling is optimized to co-locate similar prefixes on the same devices while balancing load across the cluster. Marconi [21] targets a prefix caching system designed to accommodate the unique characteristics of hybrid models that combine attention with state-space or recurrent components. Orthogonally, JENGA [40] focuses on efficient memory allocation for managing heterogeneous embeddings in modern LLM architectures.

Optimizing caching policies. Wang et al. [34] propose a workload-aware eviction policy that adapts cache management to request characteristics. However, their design treats different workloads independently, which limits its ability to achieve high prefix-cache efficiency under heterogeneous workloads. CONTINUUM [10] targets agentic LLM applications by identifying external calls, predicting their durations, and dynamically assigning time-to-live (TTL) values to pinned KV blocks to avoid premature eviction. KVFlow [22] focuses on multi-agent workflows and performs workflow-aware KV cache management by prioritizing eviction according to agent execution order. However, as discussed earlier, both CONTINUUM and KVFlow are tailored to specific workflow structures and are therefore not designed for general heterogeneous serving workloads.

8 Conclusion

In this paper, we study prefix caching under heterogeneous LLM serving workloads and show that effective eviction must account for distinct reuse mechanisms, including session-local reuse in multi-turn interactions and structural reuse in templated prompts. Guided by these observations, we design and implement a unified, task-aware eviction policy that coordinates cache management across workload classes under tight GPU memory budgets. Across hybrid traces, our policy improves prefix-cache hit ratio by 3.86%–17.32% and reduces QTTFT by 1.10×–3.63× compared to existing online baselines.

Acknowledgments

We thank all reviewers for their insightful feedback and our shepherd for valuable guidance that helped us address the reviews and improve the final version of this paper. We also thank Junpan Wu and Xuhang He for their contributions during the early stages of this work.

References

[1] Anthropic. 2025. Claude API. https://www.anthropic.com/api. Accessed: 2025-02-17.

[2] Iñaki Arango, Ayush Noori, Yepeng Huang, Rana Shahout, and Minlan Yu. [n. d.]. Prefix and Output Length-Aware Scheduling for Efficient Online LLM Inference. In *Proceedings of Sparsity in LLMs (SLLM): Deep Dive into Mixture of Experts, Quantization, Hardware, and Inference*.

[3] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

[4] György Buzsáki and Kenji Mizuseki. 2014. The Log-Dynamic Brain: How Skewed Distributions Affect Network Operations. *Nature Reviews Neuroscience* 15, 4 (2014), 264–278.

[5] Alibaba Edu. 2025. Qwen Bailian Usage Traces (Anon.). GitHub Repository. https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon Accessed: 2025-11-18.

[6] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2024. Cost-Efficient Large Language Model Serving for Multi-Turn Conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC '24)*. 111–126.

[7] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[8] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. *arXiv preprint arXiv:2403.07714* (2024).

[9] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles*. 611–626.

[10] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. *arXiv preprint arXiv:2511.02230* (2025).

[11] Bin Lin, Chen Zhang, Tao Peng, Hanyu Zhao, Wencong Xiao, Minmin Sun, Anmin Liu, Zhipeng Zhang, Lanbo Li, Xiafei Qiu, et al. 2024. Infinite-LLM: Efficient LLM Service for Long Context with DistAttention and Distributed KVCache. *arXiv preprint arXiv:2401.02669* (2024).

[12] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low-Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*.

[13] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. WebGPT: Browser-Assisted Question-Answering with Human Feedback. *arXiv preprint arXiv:2112.09332* (2021).

[14] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint arXiv:2305.02309* (2023).

[15] NVIDIA. 2025. NVIDIA A100 Tensor Core GPU Specifications. https://www.nvidia.com/en-us/data-center/a100/. Accessed: 2025-01.

[16] NVIDIA. 2025. NVIDIA H100 Tensor Core GPU Specifications. https://www.nvidia.com/en-us/data-center/h100/. Accessed: 2025-01.

[17] NVIDIA. 2025. TensorRT-LLM Documentation: KV Cache. https://nvidia.github.io/TensorRT-LLM/features/kvcache.html. Accessed: 2025-01.

[18] OpenAI. 2022. Introducing ChatGPT. https://openai.com. Accessed: 2025-12-20.

[19] OpenAI. 2023. OpenAI Developer Platform. https://platform.openai.com/docs/overview. Accessed: 2025-02-17.

[20] OpenRouter. 2026. State of AI. https://openrouter.ai/state-of-ai Accessed: 2026-03-10.

[21] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. 2025. Marconi: Prefix Caching for the Era of Hybrid LLMs. *Proceedings of Machine Learning and Systems* 7 (2025).

[22] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefix Caching for Accelerating LLM-Based Multi-Agent Workflows. *arXiv preprint arXiv:2507.07400* (2025).

[23] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation—A KVCache-Centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST '25)*. 155–170.

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[25] ShareGPT. 2023. ShareGPT: Share Your Wildest ChatGPT Conversations. https://sharegpt.com Accessed: 2025-12-20.

[26] Yifan Song, Weimin Xiong, Xiutian Zhao, Dawei Zhu, Wenhao Wu, Ke Wang, Cheng Li, Wei Peng, and Sujian Li. 2024. AgentBank: Towards Generalized LLM Agents via Fine-Tuning on 50000+ Interaction Trajectories. *arXiv preprint arXiv:2410.07706* (2024).

[27] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. 2024. Preble: Efficient Distributed Prompt Scheduling for LLM Serving. *arXiv preprint arXiv:2407.00023* (2024).

[28] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. Galactica: A Large Language Model for Science. *arXiv preprint arXiv:2211.09085* (2022).

[29] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. LaMDA: Language Models for Dialog Applications. *arXiv preprint arXiv:2201.08239* (2022).

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* 30 (2017).

[31] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-Based LeCaR. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*.

[32] vLLM Project. 2025. Automatic Prefix Caching. https://docs.vllm.ai/en/latest/design/prefix_caching/. Accessed: 2025-11-17.

[33] vLLM Project. 2025. *vLLM: A High-Throughput and Memory-Efficient Inference and Serving Engine for LLMs (v0.8.5)*. https://github.com/vllm-project/vllm/tree/v0.8.5 Accessed: 2026-01-12.

[34] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 28, 18 pages.

[35] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *arXiv preprint arXiv:2305.05920* (2023).

[36] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*. 94–109.

[37] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. 521–538.

[38] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the Twentieth European Conference on Computer Systems*. 144–158.

[39] Z.ai. 2025. CC-Bench Trajectories: Agentic Coding Task Trajectories. Hugging Face Datasets. https://huggingface.co/datasets/zai-org/CC-Bench-trajectories Accessed: 2025-11-18.

[40] Chen Zhang, Kuntai Du, Shu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, et al. 2025. JENGA: Effective Memory Management for Serving LLM with Heterogeneity. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 446–461.

[41] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, et al. 2024. SGLang: Efficient Execution of Structured Language Model Programs. *Advances in Neural Information Processing Systems* 37 (2024), 62557–62583.

[42] Ming Zhu, Aman Ahuja, Da-Cheng Juan, Wei Wei, and Chandan K. Reddy. 2020. Question Answering with Long Multiple-Span Answers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 3840–3849.

A Appendix

A.1 Power-law Fit of Prefill Time on H100

To characterize how prefill latency scales with input length and batch size, we fit a power-law model to the measured prefill time on NVIDIA H100 for Llama-3.1-8B. The fitted model captures the near-linear dependence on both input sequence length and batch size, and is used to justify the analytical scaling assumptions in our simulator. Figure 14 reports the fitting results.

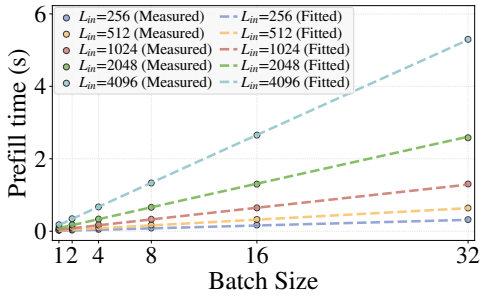


Fig. 14. **Power-law model fitting for prefill time across different configurations on H100.** Llama-3.1-8B: $prefill\_time \approx 3.59e\text{-}5 \cdot BS^{0.991} \cdot L_{in}^{1.018}$, with $R^2 = 0.999$, indicating excellent fit.

A.2 Single-turn task prompt example

We provide representative prompt instances from two single-turn benchmarks. ToolBench exemplifies *tool-use* workloads, where each request includes a reusable *Tool description* block that enumerates available tools and their schemas, in addition to a fixed *System prompt*. MASH-QA exemplifies *document QA* workloads, where a large *Document* context is appended to a shared instruction header before the *User query*.

Example prompt for ToolBench

```
System prompt: <|im_start|>system\nSystem: You are AutoGPT, you can use many tools(
    functions) to do the following task.\nFirst I will give you the task description, and
    your task start.\nAt each step, you need to give your thought to analyze the status
    now and what to do next, with a function call to actually execute your step.\nAfter
    the call, you will get the call result, and you are now in a new state.\nThen you will
     analyze your status now, then decide what to do next...\nRemember: \n1.the state
    change is irreversible, you can't go back to one of the former state, if you want to
    restart the task, say \"I give up and restart\".\n2.All the thought is short, at most
    in 5 sentence.\n3.You can do more than one tries, so if your plan is to continuously
    try some conditions, you can do one of the conditions per try.\nLet's Begin!\nTask
    description: You should use functions to help handle the real time user queries.
    Remember:\n1.ALWAYS call \"Finish\" function at the end of the task. And the final
    answer should contain enough information to show to the user,If you can't handle the
    task, or you find that function calls always fail(the function is not valid now), use
    function Finish->give_up_and_restart.\n2.Do not use origin tool names, use only
    subfunctions' names.<|im_end|>\n<|im_start|>user\n
```

Tool description: You have access of the following tools:\n1.whatsapp_api: To Send Messages From WhatsApp\n\nSpecifically, you have access to the following APIs: [{'name': 'phonelist_for_whatsapp_api', 'description': 'This is the subfunction for tool \"whatsapp_api\", you can use this tool.', 'parameters': {'type': 'object', 'properties': {'product_id': {'type': 'string', 'description': '', 'example_value': 'product_id'}}, 'required': ['product_id'], 'optional': []}} ...

User Query: User: \nMy family and I are planning a vacation and we want to send WhatsApp messages to our friends and relatives to invite them. Is there an API available that can provide us with a list of phone numbers associated with a specific product ID? It would also be helpful to check the logs to ensure that the messages are being sent successfully.

Example prompt for MASH-QA

System prompt: <|im_start|>system\nYou are a helpful assistant.<|im_end|>\n<|im_start|>user\nPlease answer the question based on the texts below.

Document: \nvar s_context; s_context= s_context || {}; s_context['wb.modimp'] = 'vidfloat'; if(webmd.useragent && webmd.useragent.ua.type === 'desktop'){ webmd.ads2.disable Initial Load(); webmd.ads2.disable Ads Init = true; $(function() { webmd.p.pim. increment(); $('.responsive-video-container').insert After('.module-social-share-container'); require(['video2/1/responsive-player/video-loader'], function(video Loader) { video Loader.init({ autoplay: webmd.useragent.ua.type === 'desktop' && ! ! s_sensitive, chron ID: $('article embeded_module[type=video][align=top]:eq(0)').attr('chronic_id'), continuous Play: true, cp Options: { flyout: true }, display Ads: true, mode: 'in-article', sticky: true }) }); }); } else { $(function(){ $('.responsive-video-container').remove(); }); } Gastritis is an inflammation, irritation, or erosion of the lining of the stomach. It can occur suddenly (acute) or gradually (chronic). ...

User Query: \nQuestion: What are the symptoms of gastritis?\nAnswer:<|im_end|>\n<|im_start|>assistant\n

A.3 Simulation Process

Algorithm 3 describes the simulation loop in the LLM serving engine. The simulator maintains a global virtual clock and explicitly models request arrivals, engine execution, and inter-turn delays, enabling accurate measurement of cache behavior.

Algorithm 3 Simulation Process

Require: A set of sessions $\mathcal{S}$; an engine `Engine`; request timestamps from the workload model

$T \leftarrow 0$ ▷ global virtual clock
$Q \leftarrow \emptyset$ ▷ min-heap keyed by event.timestamp
Initialization:
for all $s \in \mathcal{S}$ **do**
    $r \leftarrow s$.first_turn
    Push$(Q, (r.\text{timestamp}, r))$
end for
Main loop:
while $Q \neq \emptyset$ **or** `Engine.has_unfinished_requests()` **do**
    ▷ Step 1: admit all requests whose arrival time has passed
    while $Q \neq \emptyset$ **and** Q.top.timestamp $\leq T$ **do**
        $(\_, r_{\text{next}}) \leftarrow$ Pop(Q)
        `Engine.add_request(`r_{next}`)`
    end while
    if `Engine.has_unfinished_requests()` **then**
        ▷ Step 2: advance the engine by one iteration
        $outputs \leftarrow$ `Engine.step()`
        $\Delta T \leftarrow$ ComputeStepDelta$(outputs)$
        $T \leftarrow T + \Delta T$
        ▷ Step 3: schedule follow-up turns (timestamps include inter-turn delays)
        for all $o \in outputs$ **do**
            if o.is_finished **and** o.has_next_turn **then**
                $r' \leftarrow o$.next_turn
                Push$(Q, (r'.\text{timestamp}, r'))$
            end if
        end for
    else
        ▷ If idle, jump to the next event time
        if $Q \neq \emptyset$ **then**
            $T \leftarrow Q$.top.timestamp
        end if
    end if
end while

Received January 2026; revised March 2026; accepted April 2026