A Systematic Study of KV Cache Management for Efficient LLM Inference

Abstract

Key-value (KV) caching accelerates inference in transformer-based large-language models (LLMs) by storing attention states for reuse during decoding. However, as sequence lengths and concurrent requests grow, managing limited cache memory becomes a critical bottleneck. In particular, the choice of eviction and request scheduling policies strongly affects cache efficiency. We present a systematic study of KV cache management under memory constraints. Using a configurable simulator, we evaluate classic (LRU, LFU) and hybrid eviction strategies, as well as scheduling heuristics across synthetic and real-world LLM workloads. Our results show that the eviction policy has the greatest impact on the cache hit rate, with LRU performing well and a simple hybrid strategy providing further gains under large cache sizes. In contrast, the request scheduling order has only marginal effects in single-node settings. These findings offer practical guidance for the deployment of LLMs more efficiently and highlight opportunities for future adaptive or learned cache management approaches.

1 Introduction

Transformer-based Large Language Models (LLMs) have become central to modern NLP, enabling impressive performance across diverse tasks. To accelerate inference, many systems leverage KV caching, which stores previously computed key and value tensors from attention layers. This avoids redundant computation during autoregressive generation, significantly reducing latency (see Figure 1).

However, KV caching introduces new memory challenges. Its footprint grows with sequence length, attention heads, and layer count. In long-sequence or high-throughput settings, cache memory can quickly become a bottleneck. Once fully operational, the system must remove existing entries, making the design of the eviction strategy critical. A high cache hit rate means fewer recomputations and faster inference.

This work explores how to manage the KV cache under memory constraints. We investigate a range of eviction and request scheduling strategies, and evaluate them using a flexible simulator across both synthetic and real-world workloads. Our results highlight the strengths of classic policies like LRU, and show that a lightweight hybrid strategy can further improve performance. We also analyze when scheduling order matters, and how cache-aware prioritization may help. Our main contributions are threefold: (1) We design and implement a flexible simulation framework for studying KV cache management. (2) We provide the first systematic comparison that decouples the effects of eviction and scheduling policies. (3) Our empirical results offer direct, practical guidance, demonstrating that optimizing the eviction policy yields significantly higher returns than complex request scheduling in single-node environments.

Preprint. Under review.

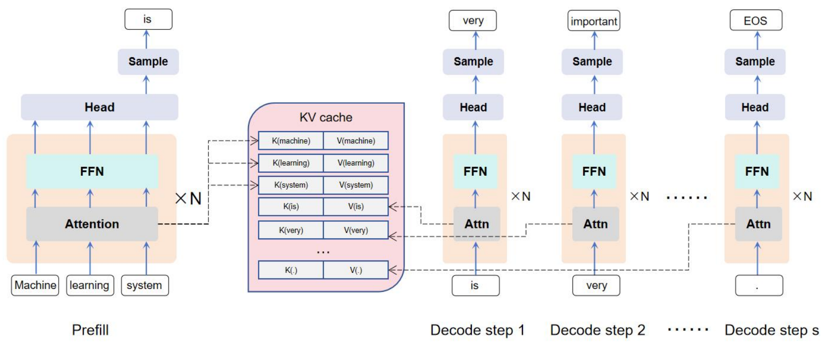

Figure 1: Illustration of the KV Cache mechanism during the decoding steps of a Transformer LLM. The cache stores K and V pairs for each token, which are then used by subsequent attention computations.

2 Related Work

The efficiency of large language model (LLM) inference is fundamentally tied to the management of the Key-Value (KV) cache. Introduced as part of the Transformer architecture [5], the KV cache stores the key and value vectors from the self-attention mechanism for all preceding tokens in a sequence. This prevents costly re-computation during the autoregressive generation of each new token. However, the memory footprint of this cache grows linearly with the sequence length and batch size, rapidly becoming the primary bottleneck in LLM serving. As detailed by [3], this memory pressure limits context length and throughput, creating a critical need for intelligent cache management strategies that can maximize the utility of limited GPU memory. Our work directly addresses this challenge by focusing on the eviction policies for a shared KV cache.

The core of our investigation lies in adapting and designing cache eviction policies for the unique workload of LLM inference. The problem of cache replacement is well-studied in classical computer systems, with algorithms like Least Recently Used (LRU) and Least Frequently Used (LFU) providing foundational principles for managing finite resources. In the context of LLMs, these policies can be applied to prefix caching, where the KV states of common prompts (e.g., system instructions or shared documents) are stored for reuse across multiple requests. While a simple LRU policy offers a reasonable baseline, it may not be optimal, as it fails to consider factors like the length of a prefix (its re-computation cost) or its global popularity. This motivates our exploration of both classical algorithms and novel hybrid strategies tailored to the specific cost-benefit trade-offs in KV cache management.

State-of-the-art LLM serving systems have provided sophisticated memory management frameworks that serve as a foundation for advanced caching policies. A landmark contribution is Paged Attention, introduced by the vLLM system [1], which virtualizes the KV cache into non-contiguous blocks, enabling efficient prefix sharing. Other systems like Orca [6] have focused on scheduling to improve resource utilization. More recently, architectures like Mooncake [4] push this further, explicitly trading larger storage resources—potentially across memory tiers—for reduced computation by preserving more KV cache history. While these cutting-edge systems focus on engineering novel architectures to directly boost performance, our work takes a complementary path. We do not aim to build a new high-performance system, but rather to conduct a systematic exploration and comparative analysis of various eviction strategies within a controlled environment. Our goal is to illuminate the fundamental trade-offs of these policies, providing insights into their behavior under different workloads, rather than maximizing raw performance.

3 Method

To systematically evaluate KV cache strategies in LLM inference, we design a modular simulation framework that mimics real-world serving conditions, tracks cache state and requests over time, and supports pluggable eviction and scheduling policies. An overview of the architecture is shown in Figure 2.

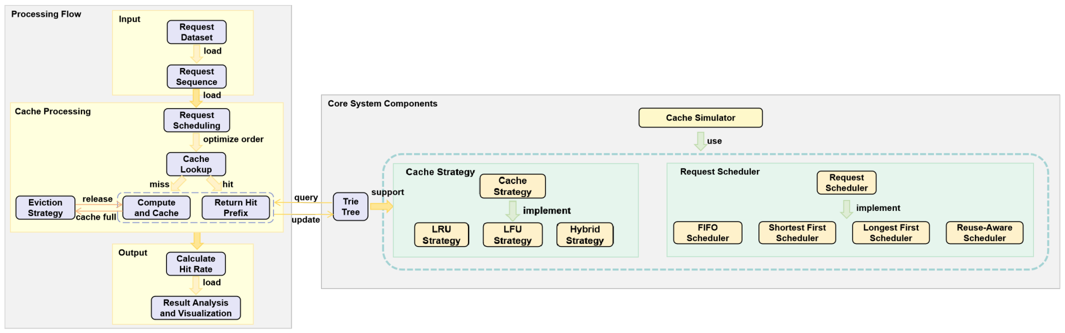

Figure 2: Architecture of the cache simulation framework. The request scheduler determines execution order; the cache simulator performs token-wise decoding and interacts with the KV store; the eviction policy selects sequences to evict when the cache exceeds capacity. Prefix-sharing is handled via a hash-based Trie index.

3.1 Simulation Framework

At a high level, the simulator reads token sequence requests from dataset traces, schedules their execution, and simulates token-by-token decoding. During each step, it performs cache lookup for the required key-value (KV) pairs. On cache hits, KV states are reused; on misses, they are computed and inserted into the cache. If insertion exceeds memory limits, the active eviction policy is invoked to free space.

Internally, the simulator manages three modular components: the core engine tracks simulation time and maintains the global request queue and cache state; the eviction policy module implements the logic for selecting which cached sequences to remove; and the request scheduler determines request execution order. Each module can be independently configured to enable controlled comparisons across strategies.

3.2 Prefix Sharing via Trie-Based Indexing

To exploit prefix reuse across requests (e.g., shared prompts or contexts), we implement a Trie-based cache index. Each node in the Trie represents a token (or its hashed prefix), and paths correspond to token sequences. This enables efficient lookup and sharing of common prefixes, which are critical in high-throughput serving.

To avoid storing raw token sequences, we apply a prefix hashing scheme: each token contributes to a cumulative hash that uniquely represents its path. For a sequence $[t_1, t_2, t_3]$, we compute hash IDs recursively: $h_1 = H(t_1)$, $h_2 = H(h_1 + t_2)$, and so on. Sequences that share prefixes thus share paths in the Trie and their cached KV blocks.

Cache entries are stored in fixed-size blocks, typically corresponding to a fixed number of tokens (e.g., 512). A cached sequence may span multiple blocks. This block granularity simplifies space accounting and eviction, as we track cache size at the block level rather than per-token.

3.3 Eviction Strategies

We implement and compare three eviction strategies. The first is Least Recently Used (LRU), which removes the sequence that has not been accessed for the longest time. Each Trie node tracks its last access timestamp, and eviction targets the least-recently used leaf path.

The second strategy is Least Frequently Used (LFU), which evicts the sequence with the lowest access count. Counters are incremented on each lookup, and eviction removes the least-used entry.

To balance multiple reuse signals, we introduce a Hybrid strategy that combines access frequency, sequence length, and recency into a composite score:

$$\text{value} = \frac{\text{usage} \times \text{length}}{1 + \text{recency}}.$$

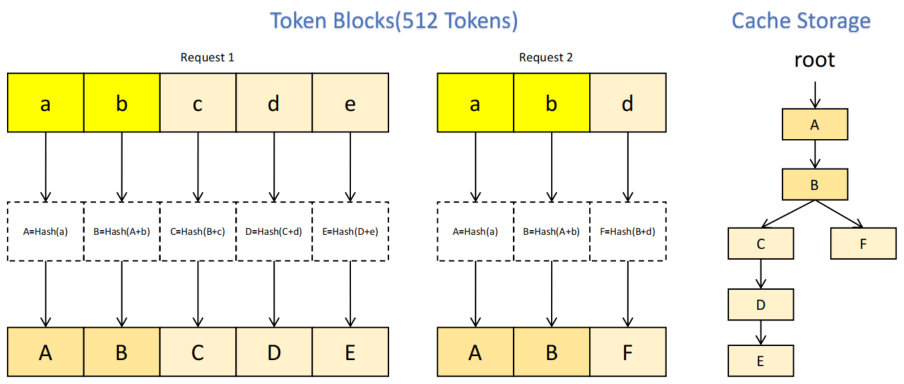

Figure 3: Example of prefix hashing and Trie construction for two input sequences. Request 1: $[a, b, c, d, e]$, and Request 2: $[a, b, d]$ are recursively hashed and inserted into a Trie structure. Nodes represent hash values of prefixes. The shared prefix $[a, b]$ is stored once, allowing efficient memory reuse and lookup. This structure supports fast prefix matching and compact representation of overlapping token sequences in the KV cache.

Sequences with the lowest value are evicted first. This formulation favors entries that are frequently accessed, long, and recently used—properties that indicate high potential for reuse.

Eviction is triggered whenever inserting new KV blocks would exceed the cache's total capacity. The affected entries are removed from both the Trie and the underlying block storage.

3.4 Request Scheduling Strategies

In concurrent serving scenarios, the order in which requests are processed can impact cache locality. To investigate this, we implement four scheduling strategies.

The default is FIFO, where requests are served in arrival order. We also experiment with length-based strategies: Shortest First, which prioritizes short sequences to quickly release cache resources; and Longest First, which prioritizes longer sequences that may populate the cache with reusable prefixes.

Finally, we design a Reuse-Aware scheduler. Before processing, it estimates prefix overlap between pending requests and the current cache state using prefix lookup. Requests with higher overlap are prioritized, increasing the chance of immediate cache reuse. Detailed algorithm is shown in the appendix 1.

It is important to note that scheduling only influences request order, not eviction decisions. However, by improving temporal locality among similar requests, it can indirectly raise hit rates.

4 Experiments

We conduct extensive simulations to evaluate the effectiveness of different KV cache eviction and request scheduling strategies. Our experiments aim to answer three core questions: (1) Which eviction strategy achieves the highest cache hit rates across diverse workloads? (2) Does request scheduling meaningfully impact cache efficiency? (3) How do cache size and workload characteristics influence performance?

4.1 Datasets and Setup

We evaluated our system on both synthetic and real-world datasets, summarized below.

LooGLE LongDep-QA [2] is a synthetic benchmark composed of long-form question-answering pairs with an average prefix reuse rate of 91%. Sequences are randomly shuffled and grouped into batches to simulate concurrent requests. MoonCake Traces [4] include realistic token traces extracted from deployed LLM serving platforms. Table 1 lists the details about the datasets that we care most about.

Table 1: Description of datasets used in the experiments. Prefix reuse rates are estimated based on trace analysis.

Dataset Name	**Description**	**Prefix Reuse (%)**
`LooGLE LongDep-QA`	Synthetic long-form question-answering pairs	91%
`conversation_trace`	Interactive user sessions from MoonCake	40%
`toolagent_trace`	Multi-turn tool-using agents from MoonCake	59%
`synthetic_trace`	Diverse generated workloads from MoonCake	66%
`mooncake_trace`	A mixed aggregate of all above types	50%

Each request is treated as a token sequence. Cache performance is measured by block-level hit ratio:

$$\text{Hit Ratio} = \frac{\text{KV blocks reused}}{\text{total KV blocks requested}}.$$

We evaluate all combinations of eviction (LRU, LFU, Hybrid) and scheduling (FIFO, Shortest, Longest, Reuse-aware) strategies, under multiple cache sizes (100, 1k, 10k, 100k blocks), yielding over 300 configurations.

4.2 Eviction Strategy Evaluation

Figure 4 shows cache hit rates for LRU, LFU, and Hybrid policies across datasets and cache sizes. Several trends emerge.

First, LRU and Hybrid consistently outperform LFU, especially under small cache budgets. LFU struggles to adapt to the dynamic and bursty reuse patterns typical in LLM serving, often achieving less than half the hit rate of LRU.

Second, while LRU offers strong baseline performance, our Hybrid strategy slightly improves over LRU at large cache sizes, particularly on traces with longer sequences and more frequent reuse. The inclusion of usage frequency and sequence length in Hybrid provides finer control over eviction prioritization.

Third, the impact of cache size is significant. As capacity increases, all strategies improve, but Hybrid gains faster, suggesting it makes better use of the additional memory.

4.3 Scheduling Strategy Evaluation

We next isolate the effect of scheduling by comparing strategies under fixed eviction policies. Figure 5 presents results on the `conversation_trace` dataset.

Overall, scheduling has a limited impact on cache hit rates in our single-node simulator. Across all settings, differences between FIFO, Shortest, Longest, and Reuse-aware scheduling were typically under 5%.

Nonetheless, the Reuse-aware scheduler shows marginal gains in many cases. By prioritizing requests with high prefix overlap, it improves hit rates especially when request arrival is bursty and the cache is moderately sized.

4.4 Summary of Best Results

Table 2 summarizes the best hit rates achieved on each dataset under small (100-block) and large (10k-block) cache configurations. In all cases, the Hybrid eviction policy, often paired with FIFO or Reuse-aware scheduling, yields the highest hit rates.

These results reinforce that cache strategy effectiveness depends not only on algorithmic design but also on trace characteristics. While eviction policy dominates overall performance, scheduling may become more important in multi-node or distributed scenarios.

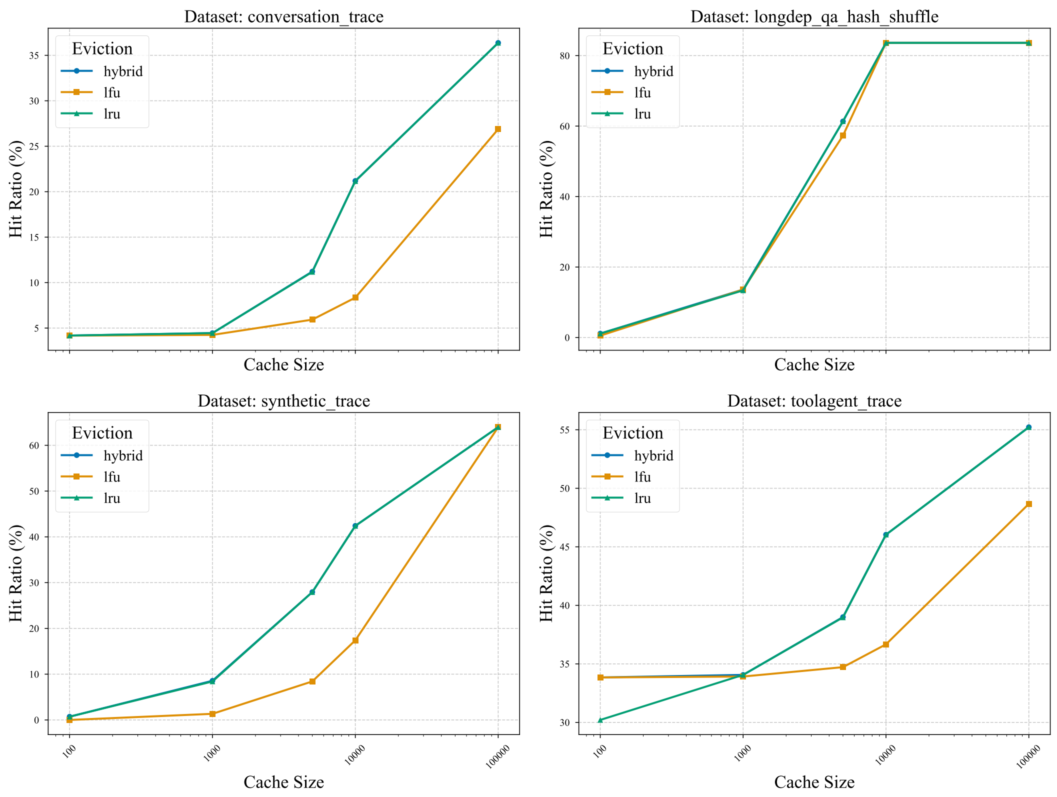

Figure 4: Cache hit rate comparison of LRU, LFU, and Hybrid eviction strategies across four workloads and multiple cache sizes. Hybrid consistently matches or exceeds LRU, while LFU lags behind, especially in smaller cache regimes.

Table 2: Best-performing strategy combinations and hit ratios (%) for small and large cache sizes.

Dataset	Small Cache (100)		Large Cache (10k)	
	Strategy (Evict/Sched)	Hit (%)	Strategy (Evict/Sched)	Hit (%)
`conversation_trace`	hybrid/reuse	4.19	hybrid/reuse	21.25
`LooGLE LongDep-QA`	hybrid/longest	3.18	hybrid/fifo	83.62
`mooncake_trace`	hybrid/reuse	33.87	hybrid/reuse	46.10
`synthetic_trace`	hybrid/fifo	0.72	hybrid/fifo	42.42
`toolagent_trace`	hybrid/fifo	33.84	hybrid/reuse	46.09

5 Conclusion

We presented a systematic evaluation of KV cache management strategies for Transformer-based LLM inference. Using a modular simulation framework, we compared classic (LRU, LFU) and hybrid eviction policies, as well as multiple request scheduling algorithms. Experiments across both synthetic and real-world traces showed that eviction strategy has the dominant effect on cache efficiency, while scheduling offers only marginal gains in single-node settings.

Among eviction policies, LRU provides strong baseline performance, and our proposed Hybrid strategy slightly improves hit rates in larger cache regimes by incorporating usage frequency and sequence length. LFU consistently underperforms due to its slow adaptation to changing reuse patterns.Request scheduling strategies, including length-based and reuse-aware variants, had limited impact on hit rates. While reuse-aware scheduling showed some benefit under bursty workloads, the effect remained secondary to eviction logic.

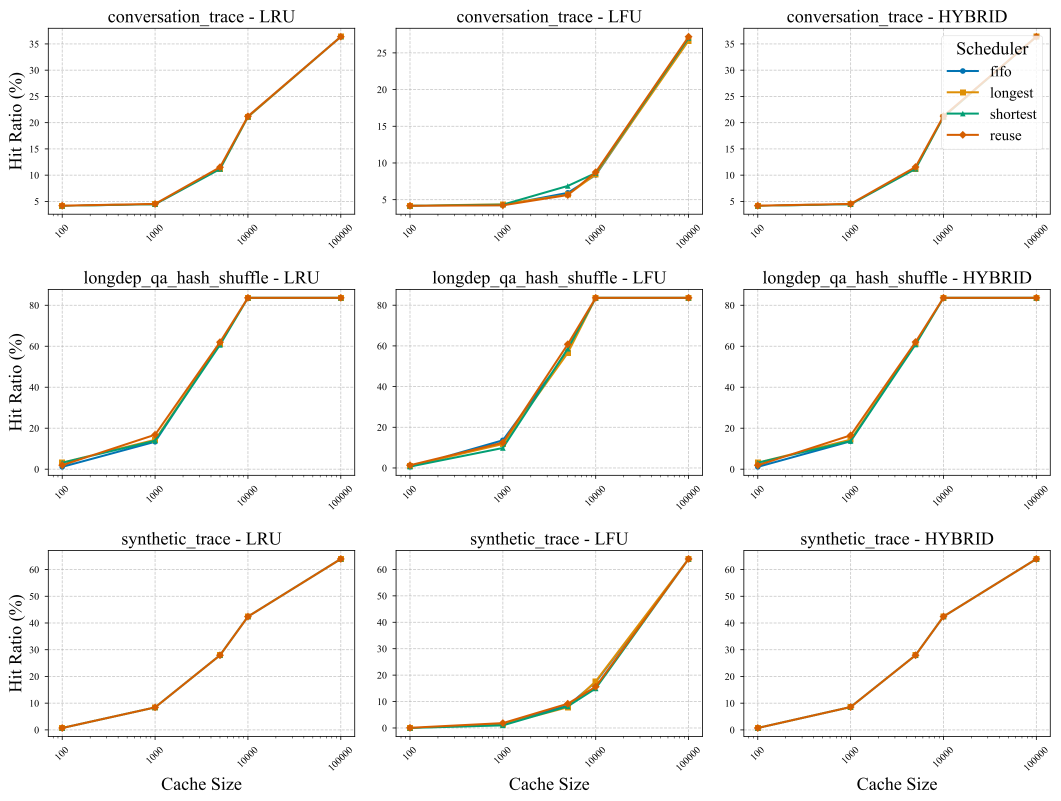

Figure 5: Impact of scheduling strategies under fixed eviction policies. The Reuse-aware scheduler slightly outperforms naive orderings (FIFO, Shortest, Longest), though absolute gains are modest in the single-node setup.

This study provides practical insights for the deployment of KV-cached LLMs under memory constraints and highlights the need to optimize policies based on workload characteristics.

Limitations. Our experiments assume a single-node cache shared by all requests. In distributed serving settings, where the cache is partitioned or partially replicated, scheduling and reuse patterns may interact more complexly. In addition, the hybrid score formulation, currently heuristic, may benefit from further optimization or learning-based tuning.

Future directions. One natural extension is to explore adaptive eviction policies that adjust behavior based on online access statistics. Another is to integrate learned predictors, for example, lightweight models that estimate the future reuse of KV sequences. Finally, moving beyond simulation, evaluating cache policies in live-serving systems with real latency measurements would offer deeper practical validation.

References

[1] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Jean Kossaifi, Anima Anandkumar, and Christos Kozyrakis. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles*, SOSP '23, page 411–426, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. Loogle: Can long-context language models understand long contexts? *arXiv preprint arXiv:2311.04939*, 2023.

[3] Reiner Pope, Sholto Zhu, Adebayo Adebayo, Myle Gottardi, Tim Feng, Shuming Shen, and Abhay Patwardhan. Efficiently scaling transformer inference. In *Proceedings of the 6th MLSys Conference*, 2023.

[4] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation — a KVCache-centric architecture for serving LLM chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 155–170, Santa Clara, CA, February 2025. USENIX Association.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, volume 30, 2017.

[6] Xupeng Yu, Zirui Geng, Min Li, Yuxuan Tian, Ziyu Zhang, Zhe Chen, and Shucheng Zheng. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, page 665–682. USENIX Association, 2022.

Appendix

To provide a clearer understanding of our approach, we include the pseudocode for both the reuse-aware scheduler and the hybrid cache management strategy in this appendix. These algorithms illustrate the core mechanisms used to score and schedule incoming requests based on cache reuse potential, as well as the trie-based structure employed to manage and evict cached blocks efficiently.

Algorithm 1: Reuse-Aware Scheduler

Input : Request queue Q, Cache strategy $cache$
Output : Scheduled request order

```
Initialize pending set: $P \leftarrow \emptyset$;
while new request $r$ arrives do
    $P \leftarrow P \cup \{r\}$;
    $bestScore \leftarrow -\infty$, $bestReq \leftarrow \emptyset$;
    foreach $req \in P$ do
        $reuse \leftarrow$ prefix length of $req.sequence$ in cache;
        $waiting \leftarrow$ current time $-req.arrival$;
        $score \leftarrow \alpha \cdot reuse + \beta \cdot waiting$;
        if $score > bestScore$ then
            $bestScore \leftarrow score$;
            $bestReq \leftarrow req$;
        end
    end
    if $bestReq \neq \emptyset$ then
        schedule $bestReq$;
        $P \leftarrow P \setminus \{bestReq\}$;
    end
end
```

Algorithm 2: Hybrid Cache Management via Trie

```
Input  : Maximum cache blocks $B$
Output : Cache hit statistics
Initialize prefix trie $T$;
$total \leftarrow 0$;
Function Insert($seq$):
    Traverse and insert $seq$ into $T$;
    UpdateStats for visited nodes;
    while $total > B$ do
        EVICT();
    end
Function Evict():
    $minVal \leftarrow \infty, target \leftarrow \emptyset$;
    foreach leaf node $n$ in $T$ do
        $v \leftarrow \frac{usage(n) \cdot length(n)}{1 + recency(n)}$;
        if $v < minVal$ then
            $minVal \leftarrow v, target \leftarrow n$;
        end
    end
    Remove $target$ and recursively clean up;
    $total \leftarrow total - 1$;
```

Also, We present the detailed experimental results.

Table 3: Detailed cache hit ratios (%) across all datasets, policies, and cache sizes.

Dataset	Eviction Policy	Scheduler	100	1k	5k	10k	100k
`conversation_trace`	Hybrid	FIFO	4.18	4.45	11.22	21.19	36.37
		Longest	4.17	4.48	11.22	21.11	36.37
		Shortest	4.17	4.47	11.21	21.15	36.37
		Reuse	4.19	4.52	11.55	21.25	36.37
	LFU	FIFO	4.17	4.25	5.93	8.35	26.88
		Longest	4.17	4.37	5.67	8.46	26.65
		Shortest	4.18	4.32	6.87	8.63	27.02
		Reuse	4.17	4.23	5.63	8.75	27.20
	LRU	FIFO	4.18	4.45	11.18	21.16	36.37
		Longest	4.17	4.48	11.20	21.08	36.37
		Shortest	4.17	4.47	11.18	21.16	36.37
		Reuse	4.19	4.52	11.51	21.22	36.37
`LooGLE LongDep-QA`	Hybrid	FIFO	1.13	13.62	61.34	83.62	83.62
		Longest	3.18	14.40	61.09	83.62	83.62
		Shortest	3.18	13.76	60.84	83.62	83.62
		Reuse	1.93	16.52	62.05	83.62	83.62
	LFU	FIFO	0.54	13.61	57.30	83.62	83.62
		Longest	1.08	11.76	56.52	83.62	83.62
		Shortest	0.68	9.82	58.60	83.62	83.62
		Reuse	1.34	12.43	60.79	83.62	83.62
	LRU	FIFO	1.13	13.38	61.38	83.62	83.62
		Longest	3.18	14.33	61.01	83.62	83.62
		Shortest	3.18	13.89	60.66	83.62	83.62
		Reuse	1.95	16.81	61.99	83.62	83.62
`mooncake_trace`	Hybrid	FIFO	33.86	34.08	39.02	46.05	55.22
		Longest	33.85	34.08	39.07	46.08	55.22

Table 3 – continued from previous page

Dataset	Eviction Policy	Scheduler	100	1k	5k	10k	100k
		Shortest	33.85	34.09	39.01	46.06	55.22
		Reuse	33.87	34.13	39.28	46.10	55.22
	LFU	FIFO	29.64	33.93	34.86	36.78	48.64
		Longest	2.67	33.91	35.31	37.09	48.63
		Shortest	33.86	33.99	34.82	36.55	48.60
		Reuse	2.67	33.97	34.87	36.98	48.60
	LRU	FIFO	30.23	34.08	38.99	46.05	55.22
		Longest	29.66	34.08	39.04	46.08	55.22
		Shortest	29.66	34.08	38.97	46.07	55.22
		Reuse	31.60	34.13	39.28	46.10	55.22
`synthetic_trace`	Hybrid	FIFO	0.72	8.56	27.94	42.42	63.96
		Longest	0.72	8.56	27.94	42.42	63.96
		Shortest	0.72	8.56	27.94	42.42	63.96
		Reuse	0.72	8.56	27.94	42.42	63.96
	LFU	FIFO	0.00	1.34	8.42	17.40	63.96
		Longest	0.02	1.33	7.84	17.68	63.96
		Shortest	0.01	0.99	8.18	14.97	63.96
		Reuse	0.07	1.85	9.18	15.62	63.96
	LRU	FIFO	0.71	8.41	27.93	42.39	63.96
		Longest	0.71	8.41	27.93	42.39	63.96
		Shortest	0.71	8.41	27.93	42.39	63.96
		Reuse	0.71	8.41	27.93	42.39	63.96
`toolagent_trace`	Hybrid	FIFO	33.84	34.06	39.00	46.04	55.22
		Longest	33.83	34.06	39.04	46.06	55.22
		Shortest	33.83	34.07	38.98	46.05	55.22
		Reuse	33.84	34.11	39.25	46.09	55.22
	LFU	FIFO	33.84	33.92	34.72	36.67	48.67
		Longest	29.63	33.89	34.98	36.62	48.64
		Shortest	33.84	33.86	34.87	36.92	48.55
		Reuse	33.84	33.96	34.59	36.90	48.58
	LRU	FIFO	30.21	34.06	38.97	46.04	55.22
		Longest	29.64	34.06	39.02	46.06	55.22
		Shortest	29.65	34.06	38.95	46.06	55.22
		Reuse	31.58	34.11	39.25	46.09	55.22