CACHE ENGINEERING FOR AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Model (LLM) agents are rapidly evolving to handle complex tasks, navigating long sequences of observations and actions. However, as agent trajectories grow longer, Key-Value (KV) cache storage and I/O become dominant costs. Long KV caches can also degrade model performance, especially when an LLM's context window limit is exceeded. Prior work uses *context engineering* to reduce cache costs and support agent trajectories longer than the LLM's context window. They typically trim or summarize information irrelevant to the current round, thereby reducing the cache used in subsequent attention computations. However, modifying the prefix invalidates all subsequent cached states and triggers costly re-computation of the cache for the remaining context. Compact summarizations can also directly lose critical information, reducing accuracy in long-horizon tasks. To mitigate these limitations, we propose a new *cache engineering* scheme. It lets agents proactively drop selected KV cache while directly reusing retained ones, without relying on context engineering as a proxy. The key challenge is the history dependence of cache states: the same token sequence can yield different caches under different cache-operation histories, making token-prefix matching insufficient to determine whether a cache can be reused. Thus, we propose AKV, an efficient serving engine to support cache engineering. It provides a simple, stateless interface compatible with existing API-based serving, and hides the complexity of stateful cache management inside. We further extend the radix tree and attention masking to maximize cache reuse, enable timely eviction, and efficiently restore missing cache states. Experiments show that AKV enables a simple rolling-drop cache engineering strategy that effectively improves benchmark scores by 1.7 points while reducing the average cost by 27.7% across various LLMs and agent benchmarks. It further enables recursive self-improvement of agents' cache-engineering strategy, expanding the cost reduction to 17.8%.

1 INTRODUCTION

Large Language Model (LLM) agents are capable of increasingly long tasks. They are generally composed of both an LLM for reasoning and tool calling, and a harness that orchestrates tool execution and context. Through repeated reasoning, tool use, and interaction with environments, agents can carry out work spanning over 500 rounds (Ye et al., 2025; Team et al., 2026; Liu et al., 2025).

As sequences grow longer, Key-Value (KV) cache becomes an increasingly important bottleneck for both the cost and the quality of agent execution. In terms of cost, retaining and repeatedly accessing accumulated KV states incurs substantial storage and I/O overhead (Wu et al., 2026; Anthropic, 2026). As shown in Figure 13, KV cache reads take up 83.6% of the total cost. In terms of quality, a growing cache can exceed the context window length at which the LLM performs reliably (Hsieh et al., 2024; Chen et al., 2026; Zhou et al., 2025), leading to severe performance degradation. Efficient and reliable agent execution therefore requires control over the active KV cache during inference.

Prior work reduces the growing cache via *context engineering*. These methods trim or summarize historical messages that have already guided subsequent generations and may no longer need direct attention. However, editing earlier messages changes the prefix, invalidating the entire subsequent KV cache and triggering recomputation. This re-prefill cost discourages frequent reductions, making it expensive to keep the active cache consistently short.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

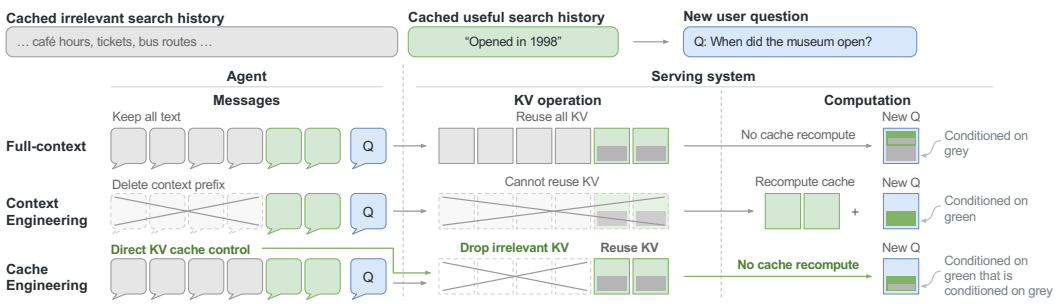


Figure 1: Full-context inference, context engineering, and cache engineering. Full-context inference reuses all historical KV. Context engineering removes irrelevant text and recomputes the retained history. Cache engineering keeps the text history but drops selected KV entries, reusing retained states without recomputation. Nested color blocks illustrate how retained KV can still carry information from dropped history.

Thus, we introduce *cache engineering*, which gives agents direct control over their active KV cache. As shown in Figure 1, it preserves the logical token sequence and reuses retained KV states, avoiding the recomputation required by text edits. Alongside the usual append operation, we introduce two additional cache operations: drop, which removes selected KV entries from future attention, and repos, which adjusts the positions encoded in retained keys. Throughout execution, the LLM and harness choose when and how to apply these operations, repeatedly interleaving them with append to shape the active cache.

However, the history-dependent nature of KV states makes it challenging to serve these sequences of cache operations. With standard KV caching, the token prefix determines the cached states. In contrast, the same token prefix and retained positions can result in different cache states under different cache operation histories, as shown in Figure 5. The serving system must therefore account for both token and cache operation histories to correctly track or reuse cache states.

Thus, we propose AKV, a serving system that exposes a cache engineering interface to agents while handling cache reuse, eviction, and restoration internally. Following the principle of *stateless interface, stateful serving*, agents submit messages and cache operations without tracking which KV states remain resident on the server. An extended radix tree uses both histories to support correct reuse and timely eviction, while segment-batch attention reconstructs missing states in a single prefill pass. The system supports various harness policies while allowing agents to search, evaluate, and refine cache engineering strategies through recursive self-improvement (RSI).

- **Cache engineering formulation.** We formalize cache engineering through explicit cache operations and their trajectories, providing a programmable interface for cache engineering in agent workloads.
- **Serving system for cache engineering.** We propose the first serving system for cache engineering that enables agents to control their caches, which were previously managed solely by the serving system. We design an extended radix tree and segment-batch attention to support efficient cache reuse, eviction, and restoration.
- **Improving efficiency and benchmark performance.** Across four agent benchmarks, a simple rolling-drop strategy reduces average cost by 27.7% relative to Standard context engineering while improving task performance by 3.5%. An RSI-based strategy further reduces costs by 17.8%. The AKV server achieves up to 2.67× the output throughput of Standard and a 7.51× speedup in P95 request latency.

2 Related Work

Agent context engineering. Context engineering supports agent trajectories beyond the model's context window. Existing methods trim, summarize, or manage the context to accommodate subsequent interactions (Lindenbauer et al., 2025; Wu et al., 2025; Lu et al., 2025; Kang et al., 2025;

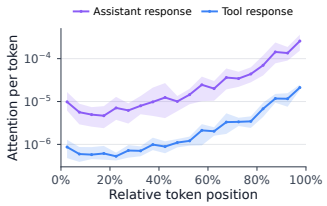


Figure 2: Attention over history. Per-token attention to assistant and tool responses, averaged over 10 BCP requests.

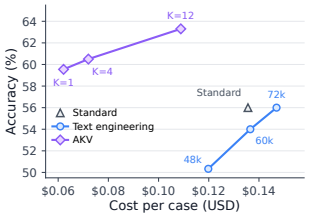


Figure 3: Accuracy–cost tradeoff. GPT-OSS-120B on the first 100 BCP questions. K denotes retained tool rounds.

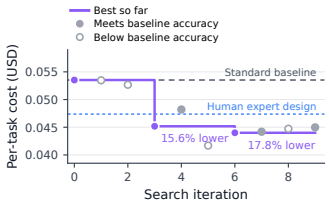


Figure 4: Agentic policy search on BCP with MiniMax-M2.7. The human expert uses rolling drop ($K = 12$).

Ye et al., 2025; OpenAI, n.d.; Anthropic, n.d.). Careful context management can also improve agent performance by removing redundant or distracting history (Tang et al., 2025; Zhang et al., 2026). However, summary-based context engineering can disrupt the agent's workflow and degrade its overall performance (Sun et al., 2025). Furthermore, modifying earlier context invalidates subsequent cached states and requires re-prefill. Some designs reduce this cost by limiting edits to the suffix (Sun et al., 2025). Others delay and batch history edits to reduce the frequency of re-prefilling (Xu et al., 2026c), at the cost of reduced flexibility. Systems can also overlap re-prefill with ongoing execution to hide its latency, but still incur the underlying computational cost (Pan et al., 2026). AKV instead lets agents discard selected KV entries directly and flexibly, enabling context management without re-prefilling the retained context.

KV cache compaction and retention. Direct KV cache reduction uses compact representations or selective retention. Compaction reduces the size or the number of KV entries through low-rank compression, quantization, sharing, and aggregation (Liu et al., 2024a;b; Shazeer, 2019; Ainslie et al., 2023; Xu et al., 2026b;a; Fu et al., 2025b), but the cache still grows with accumulated history. Retention policies limit this growth by evicting entries based on position, recency, or attention scores (Xiao et al., 2024; Zhang et al., 2023; Li et al., 2024; Fu et al., 2025a). For agent trajectories, retention policies aggregate queries across different execution phases to identify KV entries likely to be useful later (Liu et al., 2026; Li et al., 2026). SideQuest (Kariyappa & Suh, 2026) lets models express retention decisions directly.AKV provides serving support for agent-managed KV cache reduction while reusing retained states.

KV cache reuse in LLM serving. LLM serving systems reuse previously computed KV states to avoid repeated computation. Prefix trees and cache-aware scheduling enable KV reuse across requests with shared prefixes (Zheng et al., 2024; Ye et al., 2024; Zheng et al., 2026; Srivatsa et al., 2025). Edits to earlier context break prefix matches, motivating KV reuse after history changes. Position-aware caching adapts positional encodings to reuse KV states after history truncation (Gao et al., 2024) or segment relocation (Wang et al., 2025; Ma et al., 2026b; Ordonez & Parnell, 2026). Explicit cache-editing mechanisms support removing or replacing selected spans while retaining downstream KV states (Ma et al., 2026a). AKV integrates cache engineering into a stateless serving interface, supported by an extended radix tree that maximizes reuse of retained states.

3 Cache Engineering

Cache engineering adds a control dimension orthogonal to context engineering: agents decide which KV cache remains active without changing the logical token sequence.

3.1 Motivation

Previous *context engineering* methods typically drop or summarize historical messages to reduce KV cache. Dropping older tool responses can improve agent performance by keeping the active context within the model's context window (Zhang et al., 2026; Tang et al., 2025), but frequent prefix deletion incurs substantial re-prefill costs. Figure 2 shows that older tool responses receive roughly two orders of magnitude less attention than recent assistant messages. However, deleting them requires re-prefilling the entire retained suffix. For example, we find that for GPT-OSS-120B

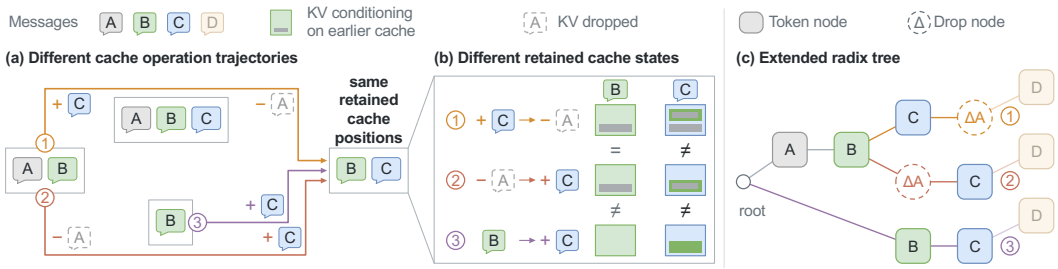


Figure 5: Cache operation trajectories and history-dependent KV states. (a) ① appends C to A and B, then drops A's KV; ② drops A's KV before appending C; ③ starts from B alone and appends C. All three retain B and C. (b) B's KV is identical in ① and ②, both conditioned on A, but differs in ③, where B has no prefix. C's KV differs in all three: ① is computed with A and B; ② and ③ cannot attend to A but use different B caches. (c) The extended radix tree shares the A–B prefix of ① and ②, then branches according to drop timing, distinguishing the cached states used to generate D.

on BrowseComp-Plus, text drop that retains the latest 12 rounds of tool responses incurs $1.85\times$ the total inference cost over its full-context counterpart. Summarization reduces re-prefill cost by leaving only a shorter suffix, but can discard useful information. As shown in Figure 3, lower summarization trigger thresholds save cost at the expense of an accuracy drop.

In contrast, *cache engineering* enables direct control of active cache states without repeated re-prefill. For example, rolling tool-response drop gradually removes older tool KV while reusing the remaining cache states (Figure 1). These caches can also carry information from the dropped history (Xiao et al., 2024; Fu et al., 2026; Zou et al., 2025), potentially preserving more information over re-prefilling. This policy achieves higher accuracy at lower cost than the summarization settings in Figure 3. It motivates us to formalize and explore cache engineering for agent tasks.

3.2 Cache Operations

An autoregressive LLM predicts the next token conditioned on the preceding token sequence:

$$p_\theta(x_{t+1} \mid x_{1:t}). \tag{1}$$

Modern LLM serving uses KV caching to avoid recomputing earlier tokens' states. The next-token prediction is then conditioned on the current token x_t and the preceding tokens' cache:

$$p_\theta(x_{t+1} \mid x_t, C_{1:t-1}), \tag{2}$$

where $C_{1:t-1}$ denotes the KV states computed from tokens $x_{1:t-1}$.

Conventional agent-serving engines keep KV cache management entirely internal. Agents and harnesses cannot directly select which KV states remain available to future attention. *Context engineering* is therefore expressed as changes to the text sequence. Editing an earlier prefix invalidates subsequent KV states and requires recomputation.

Cache engineering gives agents or their harnesses direct control over the active cache. In AKV, we define three operations: append, drop, and repos. For a logical sequence $x_{1:n}$, let $\mathcal{I} \subseteq \{1, \ldots, n\}$ index the active tokens and $\mathcal{C} = \{C_i = (K_i, V_i) : i \in \mathcal{I}\}$ denote their cache. We use $\pi(i)$ for token i's encoding position and q for the next appended token's position. Initially, $n = q = 0$ and $\mathcal{I} = \varnothing$.

Append. Processing x_{n+1} against the active cache at position q produces its KV state C_{n+1}. Append(x_{n+1}) adds this state to $\mathcal{C}$, sets $\pi(n+1) = q$, and then increments both n and q. Tokens from the prompt, tool responses, and model generation all enter the cache through append. Without cache engineering, inference consists of a sequence of append operations.

Drop. For a set $P \subseteq \mathcal{I}$ of active logical token indices, drop removes their KV states from future attention:

$$\text{drop}(P): \qquad \mathcal{C} \leftarrow \mathcal{C} \setminus \{C_i : i \in P\}, \qquad \mathcal{I} \leftarrow \mathcal{I} \setminus P. \tag{3}$$

The logical token sequence, the remaining KV states and their positions, and q are unchanged. In particular, drop does not retroactively recompute states produced while the dropped entries were available. It reduces the active cache without re-prefilling the retained history (Figure 1).

```
{  "model": "gpt-oss-120b",
   "messages": [
      {"role": "user", "content": "Search ICLR"},      ▷ append(m_0)
      {"role": "assistant", "content": "Searching"},   ▷ append(m_1)
      {"role": "tool", "content": "ICLR is..."}        ▷ append(m_2)
   ], "cache": {
      "drop": {"1": [0],                               ▷ drop(m_0)
               "2": [1]},                              ▷ drop(m_1)
      "reposition": [2]                                ▷ repos
} }
```

Figure 6: Illustrative AKV request. Right-hand annotations show cache operations; m_i denotes message i.

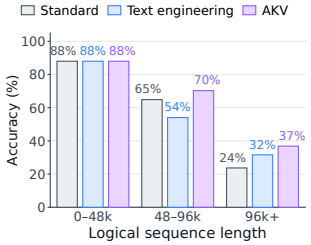


Figure 7: BCP accuracy by length with GPT-OSS-120B.

Repos. Let $i_1 < \cdots < i_r$ be the active token indices in logical order. Repos assigns their cached states consecutive positions and sets the next position accordingly:

$$\text{repos}: \qquad \pi(i_j) \leftarrow j - 1 \quad (1 \le j \le r), \qquad q \leftarrow r. \tag{4}$$

This operation removes gaps in the active positional indices after dropping states. For models using rotary position embeddings (RoPE) (Su et al., 2023), a cached key previously rotated at position i can be moved to position j by applying the rotation $R(j - i)$. The cached values and previously computed hidden states remain unchanged. Repos therefore avoids a full model forward pass, but does not produce the states that would result from prefilling a shortened text sequence.

These operations specify the cache visible to model computation. The serving system handles physical storage, eviction, and reuse separately in Section 4.

3.3 History-dependent Cache Trajectory

With drop and repos cache operations, the next-token probabilities cannot be solely determined by the token sequence. Thus, we introduce the concept of cache operation sequence to describe the entire process to reach current cache states:

$$o_{1:t} = (o_1, \ldots, o_t), \qquad o_i \in \{\text{append}(x_i), \text{drop}(p_i), \text{repos}\}. \tag{5}$$

The key property of cache engineering is history dependency, which means that the final retained cache positions cannot fully represent the retained cache states. As shown in Figure 5, while both retain B and C, in the first trajectory C is computed with A's KV available, whereas in the second trajectory it is not, leading to different cache states. Repos additionally changes the positional versions used by later computations (Figure 8). Consequently, a request's computation is specified by its token and operation sequences, rather than its final retained tokens alone.

We use *sequence length* for the logical sequence length, counting all tokens, including those whose KV has been dropped. *Active context length* counts retained historical tokens and new input tokens, regardless of cache hits or misses.

4 AKV Serving System

AKV follows the principle of *stateless interface, stateful serving*. Agents specify desired cache operations through a simple interface (Section 4.1), while the server handles cache reuse, eviction, and restoration internally (Section 4.2).

4.1 Serving Interface

The stateless interface lets agents specify desired cache operations without tracking server-side cache state. As shown in Figure 6, it extends the standard `messages` field with optional `cache` instructions. The `messages` field retains the full message history, with each token implicitly specifying an append cache operation.

The optional `cache` field specifies which operations to apply and when, using an *operate-after* grammar. For each `drop`, the key identifies the message after which to apply drop, and the value

lists the indices of messages whose KV entries are dropped. The `reposition` field lists the message indices after which to apply repos.

In common multi-turn scenarios, clients continue a conversation by extending the message and operation histories. They append new messages, such as assistant responses, tool responses, or user inputs, and optionally add desired cache operations. Together, these histories fully specify the current cache states, as discussed in Section 3, allowing the server to reuse resident KV or reconstruct missing states without client involvement.

4.2 Cache State Management

AKV uses each request's token and operation histories to manage KV reuse, eviction, and restoration efficiently while preserving request isolation.

4.2.1 Design Objectives and Challenges

AKV aims to preserve three properties of modern serving systems built around append-only cache management: (1) reusing all and only reusable cache states, (2) evicting unused cache states without compromising request isolation, and (3) restoring evicted cache states within one prefill pass.

Preserving these properties becomes challenging with drop and repos cache operations. (1) Token prefixes alone no longer determine cache states, so ordinary prefix matching can incorrectly reuse states from different cache operation histories (Figure 5(b)). (2) KV cache dropped by one request may still be needed by another, so immediate eviction can violate request isolation. (3) Ordinary causal prefill cannot reproduce historical drops and repositions, while replaying them step by step requires multiple forward passes. AKV addresses these challenges with an extended radix tree, drop-aware eviction, and segment-batch attention, respectively.

4.2.2 Drop-aware KV Cache Management

Reuse with drop node. AKV extends radix-tree prefix matching with drop nodes. Each drop node Δ marks the trigger point and records the dropped token indices. Each path records the token and cache operation histories that determine its cache states. A drop operation only affects later computation, so this placement preserves prefix sharing while distinguishing subsequent states. Prefix matching is then applied on this extended radix tree, as shown in Figure 5. It reuses the respective B and C caches when processing D, maximizing reuse without mixing incompatible states.

Evict with drop-awareness. AKV decouples logical cache dropping from physical eviction. In Figure 5, A's KV becomes evictable once no active computation or restoration needs it, even while requests ① and ② are still running. Eviction frees the KV storage but retains the tree node, keeping cached descendants reachable through prefix matching.

Restore with segment-batch attention. AKV restores cache states by encoding the drop history in *segment-batch attention* in one prefill pass. Queries are split at drop boundaries, and each segment selects the KV entries visible at that stage of the original computation. Each segment uses standard causal attention, and the segments are batched within each layer. This restores the required KV states in one prefill pass without separate forward passes at each drop (Appendix A.1.1).

4.2.3 Reposition-aware KV Cache Management

Repos changes cached K's positional encodings. Unlike drop, a repos operation affects both preceding and subsequent KV caches.

Reuse with branch copy. AKV records the encoded position of each radix-tree node and only directly reuses a cached node when its position matches. If the position does not match, AKV copies the mismatched branch and repositions the Key caches in the copied branch. For example, in Figure 8(c), after repositioning at message C, AKV creates a new branch in which B is placed at position 0 and C is at position 1. Similar to drop, AKV then appends a reposition node to the new branch, annotated with C as the reposition trigger point. Appendix A.1.2 explains the rationale behind this annotation and how the corresponding KV entries are adjusted.

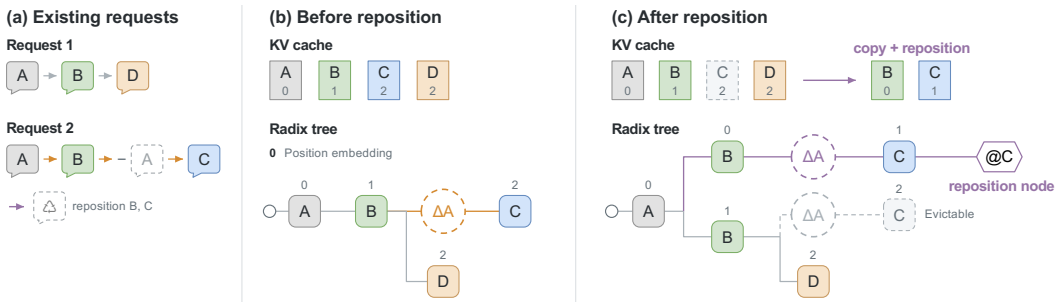


Figure 8: Repos in the KV pool and radix tree. (a) Two requests share A and B; Request 2 drops A before appending C. (b) B and C initially occupy positions 1 and 2. (c) Repos creates copies at positions 0 and 1 by re-encoding K and copying V. The original B remains shared with Request 1, while the original C becomes evictable. The tree branches at B to distinguish positional versions and records a reposition marker after C. Labels p denote position p.

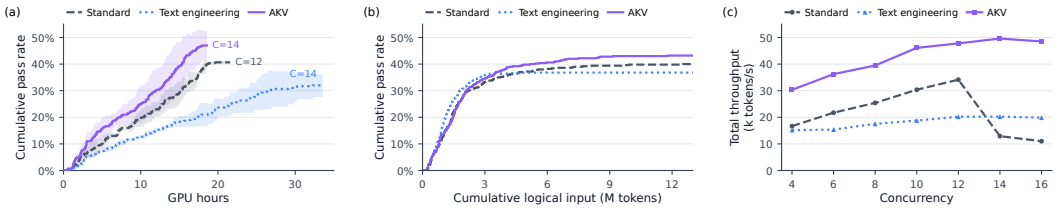


Figure 9: Task performance and serving throughput with GPT-OSS-120B on SWE-bench Verified. (a) Cumulative pass rate versus GPU hours over 100 tasks. (b) Cumulative pass rate versus cumulative logical input length over 500 tasks. (c) Total serving throughput versus concurrency.

Restore with temporary KV recovery. Similar to drop, AKV uses segment-batch attention to restore the required KV states in a single prefill pass. It partitions queries at the repos trigger point so that each segment shares the same positional encoding over its attended KV caches. AKV then constructs the required KV representations for each segment temporarily and batches all segments into one attention computation. The eviction policy with repos is the same as that of drop.

5 Experiment

5.1 Setup

Baselines. We compare AKV against two context engineering baselines. Following common practice, Standard retains the accumulated history and triggers summarization at 75% of the context window. We set the maximum context length to 128K tokens, giving Standard a summarization threshold of 96K tokens. Text Engineering summarizes context when the context length reaches 48K tokens. AKV applies our rolling-drop policy to the KV cache, retaining the latest 12 tool responses. Both the baselines deploy the model using SGLang.

Models and benchmarks. We evaluate AKV across different model families and scales, including MiniMax-M2.7, GPT-OSS-120B, and MiroThinker-1.7-Mini. Our evaluation benchmarks cover a range of long-horizon agentic tasks. For deep research, we choose BrowseComp-Plus (Chen et al., 2025) (BCP), xbench-DeepSearch-2510 (xbench, 2025), and GAIA (Mialon et al., 2023). For software engineering, we choose SWE-bench Verified (OpenAI, 2024).

Metrics. We report success rate and average API-equivalent inference cost per task, measured in cents unless otherwise specified. For each task, the cost is calculated based on the active input lengths and generated output lengths. For cost accounting, we count repositioned tokens as cache hits, since the rotary adjustment described in Section 3.2 incurs little computational overhead. We use Groq's API rates for GPT-OSS and MiniMax's official API rates for MiniMax-M2.7. For MiroThinker-1.7-Mini, we use the official API rates of its base model, Qwen3-30B-A3B.

Table 1: Benchmark scores (pass rate, %) and costs (cents) by model and management method. Bold indicates the highest score and lowest cost within each model, including ties.

Method	BCP		GAIA-Text		SWE-Verified		xbench-DS		Average	
	Score	Cost	Score	Cost	Score	Cost	Score	Cost	Score	Cost
MiniMax-M2.7										
Standard	58.9	5.2	35.0	**2.4**	**75.2**	7.0	**40.5**	5.5	52.4	5.0
Text Engineering	**62.3**	5.3	36.9	2.7	72.0	9.4	40.0	5.2	52.8	5.7
AKV	58.9	**4.8**	**40.8**	2.6	74.4	**5.5**	**40.5**	**4.7**	**53.7**	**4.4**
GPT-OSS-120B										
Standard	57.5	13.1	68.0	5.4	40.0	20.0	**38.5**	6.2	51.0	11.2
Text Engineering	52.7	12.6	65.1	6.8	36.8	15.8	**38.5**	5.4	48.3	10.2
AKV	**61.9**	**11.4**	**69.9**	**4.1**	**43.2**	**9.7**	37.5	**3.8**	**53.1**	**7.3**
MiroThinker-1.7-Mini (30B activate 3B)										
Standard	60.0	25.2	68.0	**9.2**	**7.8**	33.6	40.5	13.5	44.1	20.4
Text Engineering	45.7	17.6	66.0	11.0	4.2	28.4	32.5	15.3	37.1	18.1
AKV	**63.5**	**15.3**	**68.9**	9.5	7.4	**26.3**	**43.5**	**8.1**	**45.8**	**14.8**

Hardware. We evaluate the serving efficiency of AKV and the baselines. All measurements are conducted on 8 NVIDIA A800-80GB GPUs, with 4 GPUs for prefill and 4 for decode, each using a tensor parallelism size of 4. Since A800 GPUs lack native MXFP4 compute support, all methods use model weights that are dequantized to BF16. Detailed settings are in A.1.3.

5.2 Agent Benchmark Results

Main results. As shown in Table 1, AKV achieves the highest average success rates across three models, improving the average success rate by 1.3–2.1% and 0.9–8.7% compared with two baselines, respectively. AKV also reduces inference cost by 27.7% relative to Standard and 21.9% relative to Text Engineering, averaged across all four models and four benchmarks.

Source of improvements. We calculate the accuracy of AKV and the baselines on GPT-OSS-120B over BrowseComp-Plus, grouped by logical sequence lengths. As shown in Figure 7, all three methods achieve comparable accuracy on short trajectories (< 48K tokens). However, as the logical sequence length increases, the accuracy of both baselines degrades substantially faster than that of AKV. We attribute this gap to the information loss introduced by summarization in the baselines.

5.3 Serving Efficiency

Main results. Table 2 shows that AKV improves serving efficiency over Standard, with larger gains at higher concurrency. Across different concurrency levels, AKV achieves 1.46–3.54$\times$ and 1.59–7.51$\times$ speedups in mean and P95 request latency, respectively. Input and output throughput reach 1.45–4.68$\times$ and 1.08–2.67$\times$ the baseline levels. At high concurrency ($C \geq 14$), KV-cache pressure causes Standard to spend more time on re-prefill, leaving fewer requests decoding concurrently. This results in lower TPOT, but at the cost of higher end-to-end latency and lower throughput.

Source of improvements. Figure 12 in Appendix A.3.1 shows that the speedup comes mainly from reduced full-attention time. After KV entries are dropped, each subsequent decode step attends only to the retained active context. This reduces the KV data loaded from GPU memory and the attention computation performed over historical tokens. Detailed settings are in Appendix A.1.4.

5.4 Cache Engineering Strategies

Drop targets and reposition. Table 3 shows that dropping tool KV with repos achieves the best pass rate. Without repos, position indices grow until the context-window limit is reached, preventing

Table 2: Serving performance (latency, s; TTFT, s; TPOT, ms/token; throughput, tokens/s) of GPT-OSS-120B on Top-80 SWE-Bench across concurrency (C) levels.

C	Method	Request latency		TTFT		TPOT		Throughput	
		mean	P95	mean	P95	mean	P95	Input	Output
8	Standard	12.16	43.57	3.18	8.42	26.57	31.38	25200	217
	Text Engineering	12.84	43.57	4.94	22.99	**23.23**	**25.68**	17313	228
	AKV	**7.79**	**25.93**	**1.87**	**5.99**	23.50	29.42	**39227**	**245**
10	Standard	12.82	45.03	3.93	9.76	26.47	30.83	30108	254
	Text Engineering	15.17	50.27	7.35	34.07	**22.57**	**25.09**	18533	246
	AKV	**8.13**	**27.11**	**2.22**	**7.36**	23.20	29.86	**46002**	**295**
12	Standard	12.39	43.16	2.97	9.99	28.00	32.78	33832	313
	Text Engineering	16.83	52.82	8.86	37.98	23.24	**25.95**	19945	264
	AKV	**8.47**	**27.18**	**2.60**	**8.17**	**23.06**	29.23	**48923**	**339**
14	Standard	30.16	108.86	21.36	88.29	**24.50**	29.85	12767	146
	Text Engineering	19.59	60.67	11.19	47.36	**24.50**	**27.23**	19992	267
	AKV	**9.09**	**28.49**	**2.92**	**8.62**	**24.50**	31.56	**51837**	**369**
16	Standard	36.50	229.90	27.71	223.57	24.83	29.25	10866	138
	Text Engineering	23.00	69.96	14.48	52.71	**24.44**	**27.56**	19595	267
	AKV	**10.32**	**30.63**	**4.16**	**13.02**	24.84	32.41	**50817**	**369**

Table 3: Drop targets and repos with GPT-OSS-120B.

Strategy	Repos	BCP		SWE	
		Score	Cost	Score	Cost
Standard	✗	56.0	13.5	**41.0**	19.1
Tool drop	✗	55.0	7.1	**41.0**	9.4
Reasoning drop	✓	56.5	8.2	36.5	14.2
Both drop	✓	56.5	8.7	34.5	7.8
Tool drop	✓	**62.0**	10.9	**41.0**	9.5

Table 4: Agentic cache reduction on BCP. CAT means Cache-As-Tool. Costs in cents.

Model	Metric	Method		
		Standard	Text Eng.	CAT
GPT-OSS-120B	Score	52.1	50.0	**61.0**
	Cost	13.4	13.9	**12.1**
MiniMax-M2.7	Score	56.0	61.0	**64.0**
	Cost	**5.4**	5.5	5.9

further exploration. Dropping reasoning KV, either alone or together with tool KV, offers little accuracy benefit over Standard and can even hurt agent performance.

Recursive self-improvement (RSI). Figure 4 shows that RSI discovers lower-cost policies than our manually designed rolling-drop policy ($K = 12$) while maintaining Standard's accuracy. On BCP, MiniMax-M2.7 iteratively refines cache reduction policies, reducing per-task cost by 15.6% at iteration 3 and 17.8% at iteration 6 relative to Standard.

Cache management as a tool. Table 4 shows that agentic cache reduction achieves the highest BCP score: 61.0% for GPT-OSS-120B and 64.0% for MiniMax-M2.7. It also lowers GPT-OSS-120B's cost to 12.1 cents, while MiniMax-M2.7's cost rises slightly to 5.9 cents as it executes more turns.

6 Conclusion

We introduce cache engineering, enabling agents to directly manage active KV states while reusing retained caches without recomputation. We implement AKV serving system to support this through a stateless interface, an extended radix tree, and segment-batch attention for efficient cache reuse, eviction, and restoration. Across agent benchmarks, cache engineering improves average task performance while reducing inference costs, and AKV increases serving throughput and lowers latency.

Limitations. More proactive exploration of cache engineering strategies, agentic model training under this scheme, and serving system support for linear attention models remain future work.

AI USE STATEMENT

We used generative AI tools to assist in polishing this paper. We take responsibility for the final content of this work, including the text, claims, and artifacts produced with the assistance of generative AI.

ETHICS STATEMENT

This work bridges agent-side context management and serving-side KV cache management through a stateless interface that lets agents control cache retention and removal while the serving system handles reuse, restoration, and eviction. By reducing KV cache memory requirements and attention overhead, AKV improves serving throughput and reduces request latency, potentially lowering inference costs and making long-horizon agent applications more accessible to users with limited computational resources. However, inappropriate retention decisions may remove safety-relevant context, potentially causing factual errors, instruction-following failures, or inappropriate actions. Deployment should therefore include safeguards for preserving access to critical information, task-specific evaluation of retention policies, and appropriate human oversight.

REPRODUCIBILITY STATEMENT

We describe the KV cache operations and serving mechanisms underlying AKV in Sections 3 and 4.2. The benchmark datasets and evaluation protocol are presented in Section 5, as well as the design of the serving system, in 4.2. All datasets and models used in our experiments are publicly available. Detailed information on the evaluation setting is also provided in 5 and A. To facilitate reproducibility, we also release our source code along with detailed guidelines.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 4895–4901, 2023.

Anthropic. Introducing claude fable 5.1 and claude mythos 5.1, 2026. URL https://www.anthropic.com/claude-fable-and-mythos-5-1.

Anthropic. How Claude Code uses prompt caching. Claude Code documentation, n.d. URL https://code.claude.com/docs/en/prompt-caching. Accessed September 22, 2026.

Guoxin Chen, Zile Qiao, Xuanzhong Chen, Donglei Yu, Haotian Xu, Wayne Xin Zhao, Ruihua Song, Wenbiao Yin, Huifeng Yin, Liwen Zhang, Kuan Li, Minpeng Liao, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. Iterresearch: Rethinking long-horizon agents with interaction scaling, 2026. URL https://arxiv.org/abs/2511.07327.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

earendil-works. pi: AI agent toolkit: unified LLM API, agent loop, TUI, coding agent CLI. GitHub repository, 2026. URL https://github.com/earendil-works/pi. Commit ce5ec9ca355a852fb1f25c088cf409df47a95213; accessed September 26, 2026.

Tianyu Fu, Haofeng Huang, Xuefei Ning, Genghan Zhang, Boju Chen, Tianqi Wu, Hongyi Wang, Zixiao Huang, Shiyao Li, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Moa: Mixture of sparse attention for automatic large language model compression, 2025a. URL https://openreview.net/forum?id=konDsSUSqg.

Tianyu Fu, Tengxuan Liu, Qinghao Han, Guohao Dai, Shengen Yan, Huazhong Yang, Xuefei Ning, and Yu Wang. Framefusion: Combining similarity and importance for video token reduction on large vision language models. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 22654–22663. IEEE, 2025b.

Tianyu Fu, Zihan Min, Hanling Zhang, Jichao Yan, Guohao Dai, Wanli Ouyang, and Yu Wang. Cache-to-cache: Direct semantic communication between large language models. In *International Conference on Learning Representations*, volume 2026, pp. 43130–43158, 2026.

Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. {Cost-Efficient} large language model serving for multi-turn conversations with {CachedAttention}. In *2024 USENIX annual technical conference (USENIX ATC 24)*, pp. 111–126, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Sanjay Kariyappa and G Edward Suh. Sidequest: Model-driven kv cache management for long-horizon agentic reasoning. *arXiv preprint arXiv:2602.22603*, 2026.

Junjie Li, Jiong Lou, and Jie Li. Intentkv: Cross-turn intent-aware kv cache pruning for agent inference. *arXiv preprint arXiv:2606.09916*, 2026.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as llm summarization for agent context management. *arXiv preprint arXiv:2508.21433*, 2025.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024a.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents, 2025. URL https://arxiv.org/abs/2512.22087.

Taowen Tony Liu, Jeffrey TH Wong, Can Xiao, Bowen Yang, Hao Mark Chen, and Yiren Zhao. Agentkv: Phase-aware kv eviction for agentic llms. *arXiv preprint arXiv:2609.14872*, 2026.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024b.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Scaling llm multi-turn rl with end-to-end summarization-based context management. *arXiv preprint arXiv:2510.06727*, 2025.

Bole Ma, Jan Eitzinger, and Harald Koestler. Leyline: Kv cache directives for agentic inference. *arXiv preprint arXiv:2606.01065*, 2026a.

Bole Ma, Jan Eitzinger, and Harald Köstler. Irminsul: Mla-native position-independent caching for agentic llm serving. *arXiv preprint arXiv:2605.05696*, 2026b.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

OpenAI. Introducing SWE-bench Verified. OpenAI, 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Updated February 24, 2025.

OpenAI. Developer commands. OpenAI documentation, n.d. URL https://learn.chatgpt.com/docs/developer-commands?surface=cli. Accessed September 22, 2026.

Nathan Ordonez and Thomas Parnell. Minipic: Flexible position-independent caching in¡ 100loc. *arXiv preprint arXiv:2606.13126*, 2026.

Zaifeng Pan, Qianxu Wang, Zhengding Hu, Chang Chen, Yue Guan, Yanbo Zhou, Steven Swanson, and Yufei Ding. Smoothagent: Efficient long-horizon llm-based agent serving with lookahead context engineering. *arXiv preprint arXiv:2607.00151*, 2026.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving. In *International conference on learning representations*, volume 2025, pp. 37057–37082, 2025.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Qiaoyu Tang, Hao Xiang, Le Yu, Bowen Yu, Yaojie Lu, Xianpei Han, Le Sun, WenJuan Zhang, Pengbo Wang, Shixuan Liu, et al. Beyond turn limits: Training deep search agents with dynamic context window. *arXiv preprint arXiv:2510.08276*, 2025.

MiroMind Team, Song Bai, Lidong Bing, Carson Chen, Guanzheng Chen, Yuntao Chen, Zhe Chen, Ziyi Chen, Jifeng Dai, Xuan Dong, Wenhan Dou, Yue Deng, Yunjie Fu, Junqi Ge, Chenxia Han, Tammy Huang, Zhenhang Huang, Jerry Jiao, Shilei Jiang, Tianyu Jiao, Xiaoqi Jian, Lei Lei, Ruilin Li, Gen Luo, Tiantong Li, Xiang Lin, Ziyuan Liu, Zhiqi Li, Jie Ni, Qiang Ren, Pax Sun, Shiqian Su, Chenxin Tao, Bin Wang, Wenhai Wang, Haonan Wang, James Wang, Jin Wang, Jojo Wang, Letian Wang, Shizun Wang, Weizhi Wang, Zixuan Wang, Jinfan Xu, Sen Xing, Chenyu Yang, Hai Ye, Jiaheng Yu, Yue Yu, Muyan Zhong, Tianchen Zhao, Xizhou Zhu, Yanpeng Zhou, Yifan Zhang, and Zhi Zhu. Mirothinker: Pushing the performance boundaries of open-source research agents via model, context, and interactive scaling, 2026. URL https://arxiv.org/abs/2511.11793.

Qian Wang, Zahra Yousefijamarani, Morgan Lindsay Heisler, Rongzhi Gu, Bai Xiaolong, Shan Yizhou, Wei Zhang, Wang Lan, Ying Xiong, Yong Zhang, et al. Mepic: Memory efficient position independent caching for llm serving. *arXiv preprint arXiv:2512.16822*, 2025.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Yongtong Wu, Shaoyuan Chen, Yinmin Zhong, Rilin Huang, Yixuan Tan, Wentao Zhang, Liyue Zhang, Shangyan Zhou, Yuxuan Liu, Shunfeng Zhou, et al. Dualpath: breaking the storage bandwidth bottleneck in agentic llm inference. *arXiv preprint arXiv:2602.21548*, 2026.

xbench. xbench-DeepSearch-2510. Hugging Face dataset, 2025. URL https://huggingface.co/datasets/xbench/DeepSearch-2510. Accessed September 24, 2026.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Anyi Xu, B Li, Bangcai Lin, Bing Xue, BingCheng Xian, Bingzheng Xu, Bochao Wu, Bowei Zhang, Boyi Deng, CC Yu, et al. Deepseek-v4. 1-flash: Pushing the limits of kv cache compression. *arXiv preprint arXiv:2609.19969*, 2026a.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026b.

Buqiang Xu, Zirui Xue, Dianmou Chen, Chenyang Fu, Chiyu Wu, Caiying Huang, Chen Jiang, Jizhan Fang, Xinle Deng, Yijun Chen, et al. Tokenpilot: Cache-efficient context management for llm agents. *arXiv preprint arXiv:2606.17016*, 2026c.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11608–11620, 2024.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, et al. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025.

Haoxiang Zhang, Qixin Xu, Zhuofeng Li, Lei Zhang, Pengcheng Jiang, Yu Zhang, and Julian McAuley. Masking stale observations helps search agents–until it doesn't: A regime map and its mechanism. *arXiv preprint arXiv:2606.00408*, 2026.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in neural information processing systems*, 36:34661–34710, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. Batchllm: Optimizing large batched llm inference with global prefix sharing and throughput-oriented token batching. *Proceedings of Machine Learning and Systems*, 8:1742–1756, 2026.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Jinhua Zhao, Bryan Kian Hsiang Low, and Paul Pu Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents, 2025. URL https://arxiv.org/abs/2506.15841.

Jiaru Zou, Ruizhong Qiu, Gaotang Li, Xiyuan Yang, Katherine Tieu, Pan Lu, Ke Shen, Hanghang Tong, Yejin Choi, Jingrui He, et al. Latent collaboration in multi-agent systems. *arXiv preprint arXiv:2511.20639*, 2025.

A Appendix

A.1 AKV System Design Details

A.1.1 Restore with Segment-batch Attention

Segmentation. Building on Sections 4.2.2 and 4.2.3, AKV restores missing KV states by partitioning queries whenever a drop or repos operation changes the visible history or its position mapping. Each segment selects the corresponding KV pages and applies standard causal attention, encoding historical visibility without constructing a dense custom mask.

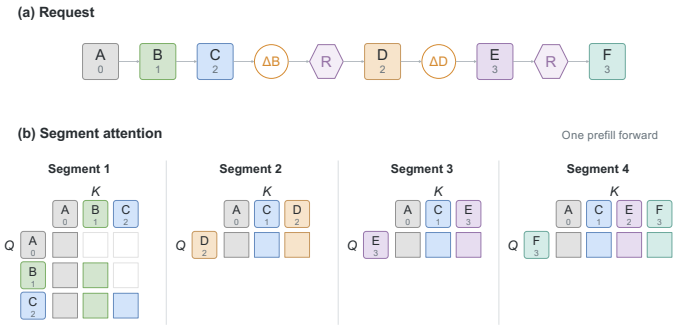


Figure 10: **Restore with segment-batch attention.** The request in (a) produces the four attention segments in (b). Numbers denote query/key positions; colored cells indicate allowed attention.

Example. Figure 10 illustrates a complete restoration, treating each letter as one token and writing X_p for token X at position p. Segment 1 computes A_0, B_1, C_2 with causal attention. After dropping B and repositioning, Segment 2 computes D_2 using keys (A_0, C_1, D_2). Dropping D starts Segment 3: E_3 attends to (A_0, C_1, E_3). Although Segments 2 and 3 share a position mapping, they require separate KV views because D is no longer visible to E; drop alone does not compact positions. The second repos moves E from position 3 to 2, so Segment 4 computes F_3 using (A_0, C_1, E_2, F_3).

Batched restoration. Within each layer, AKV writes the newly computed KV states and constructs the additional position versions, such as C_1 and E_2, by re-encoding K and copying V. The segment-specific page lists and query ranges are then submitted together to batched attention. Thus, all four segments execute within a single model forward pass, without recomputing hidden states for each position version or replaying separate forward passes at operation boundaries.

A.1.2 Reposition Scheme

Design objective. Reposition-aware cache management must distinguish positional versions while reusing their shared computational states. We therefore first identify directly reusable caches via exact matching, then use retrigger matching to find prefixes that are reusable via positional transformation, avoiding unnecessary hidden-state recomputation.

Token-node representation. Each token node is represented by a pair $T = (x, p)$, where x is the token ID and p is the position currently encoded in its cached K. The position label describes the stored cache, not necessarily the position at which the token was originally computed. For example, C is initially computed at position 2 in all three trajectories in Figure 11. After repos, trajectories 1 and 3 use $(C, 0)$, while trajectory 2 uses $(C, 1)$. Control nodes along the path preserve the operation history that governs computation.

Why position labels and R nodes are needed.

Position labels distinguish cache representations that cannot be used interchangeably. For example, C has the same original computation history in trajectories 1 and 2, but is repositioned to 0 and 1, respectively. Its K encodings therefore differ and require positional transformation before reuse.

Reposition nodes distinguish states that share final positions but arise from different computations. Trajectories 1 and 3 both drop B after ABC, but trajectory 3 executes R_C before computing D.

When D is computed, the positions of A, C, and D are respectively $(0, 2, 3)$ and $(0, 1, 2)$. The relative distance between D and A changes, while that between D and C remains unchanged, potentially producing different hidden states and KV entries for D. Both trajectories subsequently drop A and execute R_D to move C and D to positions 0 and 1. This later transformation cannot erase the computational difference already established in D. The path must therefore retain R_C before D to prevent incorrect reuse of D and subsequent cached states.

Exact and retrigger matching. Exact matching compares all token-node fields:

$$T(x, p) \equiv T(x', p') \iff x = x' \wedge p = p',$$

and requires identical control nodes and their parameters. Retrigger matching ignores only token positions while preserving all operation records:

$$\pi(T(x, p)) = T(x), \qquad \pi(\Delta S) = \Delta S, \qquad \pi(R) = R.$$

Two prefixes are history-compatible if their projections are identical.

The system initializes the result with the longest valid exact prefix, then searches compatible branches from the root. A candidate replaces the current result only if it is valid and matches strictly more real tokens, preserving exact matches in a tie. Retrigger matching stops at the first token or operation mismatch. Selected source caches are copied and positionally transformed as needed before use.

Correctness of retrigger matching. Under fixed model and positional-encoding configurations and exact arithmetic, identical projected prefixes preserve the token and operation history preceding each token's original computation. They therefore produce the same hidden states and unrotated KV content. Repos operations applied afterward change only K's positional representation, allowing the target version to be constructed as

$$K_t = \mathcal{R}(p_t)\mathcal{R}(p_s)^{-1}K_s, \qquad V_t = V_s,$$

without recomputing hidden states.

Preserving control nodes prevents this transformation from crossing a divergence in computational history. For example, trajectories 1 and 3 disagree at R_C before D, so retrigger matching cannot reuse D across them. Finite-precision transformations may introduce rounding error; this correctness argument does not require bitwise equivalence to recomputation.

A.1.3 Serving Configuration

AKV is implemented on SGLang v0.5.19. Table 5 summarizes the configuration used for the serving experiments. Both Original Context and AKV use GPT-OSS-120B with weights expanded to BF16 on eight A800 GPUs, with separate prefill and decode workers. Both workers enable radix caching; Original Context uses standard LRU eviction, while AKV enables Drop-aware eviction.

Table 5: **Serving configuration.**

Setting	Value
Model and weight precision	GPT-OSS-120B, BF16
Hardware	Eight NVIDIA A800 GPUs, 80 GB each
Parallelism	Prefill: TP4; decode: TP4
KV transfer	Mooncake over intra-node NVLink
Attention backend	Triton
MoE backend	`triton_kernel`
Static memory fraction	0.85
KV page size	One token per page
Prefill chunk size	8,192 tokens
Maximum prefill tokens	8,192
Prefix caching	Radix cache enabled on both workers
Eviction	Original Context: LRU; AKV: Drop-aware eviction

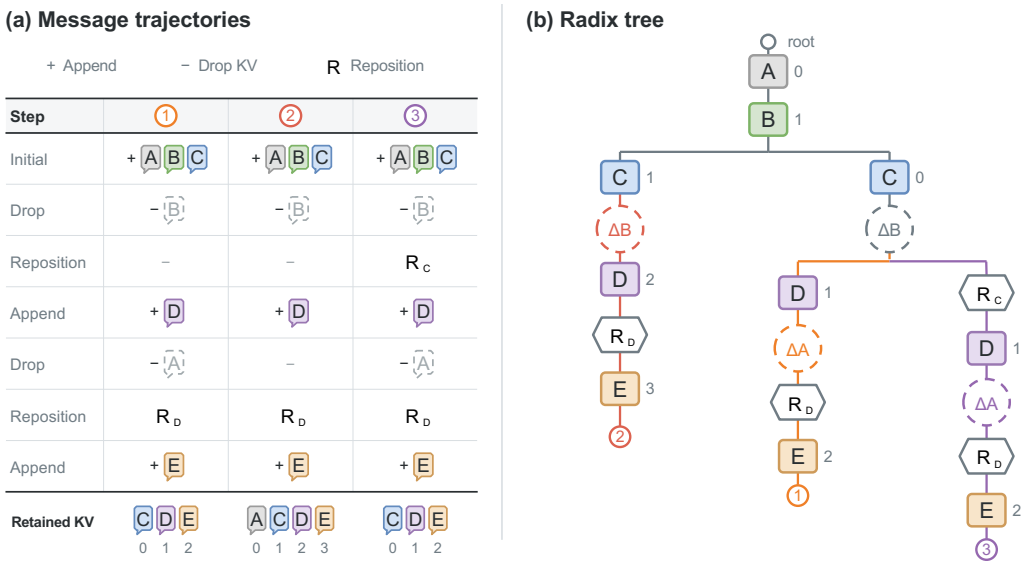


Figure 11: **Encoding drop and reposition histories in the extended radix tree.** (a) Three message trajectories interleave token appends, KV drops, and reposition operations. Although their retained tokens overlap, applying the operations at different points produces different KV states. (b) AKV records each operation in the radix tree, so trajectories share a prefix only while their cache states remain reusable. Each tuple next to an appended token denotes its position within the reposition epoch; $*$ denotes the original positioning.

A.1.4 Workload Replay and Evaluation Protocol

Task selection. To emulate realistic long-horizon agent workloads with substantial accumulated context, we select the 80 SWE-bench Verified tasks with the largest context lengths in the recorded Full Context trajectories. We rank tasks by this peak length and shuffle the selected tasks with seed 42. Both methods use the same task and ordering. Each method replays trajectories collected under its own context-management policy, so the number of turns and the output budgets may differ between methods.

Trajectory replay. For each turn, we reconstruct the output token budget from the complete recorded Harmony assistant response, including reasoning and tool-call content, after subtracting the generation prefix already present in the input. We use temperature 0 and `ignore_eos=True` to generate the prescribed number of tokens. The newly generated assistant response is retained in the history for subsequent turns, while tool responses are replayed from the recorded trajectory rather than re-executing the tools. Recorded terminal inputs without an available assistant response are excluded from replay.

Concurrency and measurement windows. We maintain C client task concurrency and execute turns sequentially within each task. The task-count targets for the measurement windows are given in Table 2; measurements include requests from supplemental tasks used to maintain concurrency. For AKV at $C \in \{14, 16\}$, we truncate the archived runs at the completion of the first 30 tasks in completion order, counting all requests within the resulting window rather than only requests from those 30 completed tasks. For Full Context at large concurrency, KV cache capacity pressure causes repeated prefill, resulting in severe serving degradation. We therefore report these three runs over a fixed one-hour window, with some requests still unfinished at the cutoff.

A.2 Context Engineering Baseline Details

Our Context engineering baselines use a Pi-derived summarization policy (earendil-works, 2026). We retain Pi's prompts, conversation serialization, cut-point selection, and default budgets for

retained context and summary output. Compaction is triggered when the prompt reaches an experiment-specific exact token-count threshold. The summary call uses the same model and reasoning settings as the agent. Its output replaces older dialogue while preserving system and developer instructions and the recent messages selected for retention. Unlike Pi, our implementation rejects an empty summary; a summary failure follows the benchmark error protocol without retry.

Pi defines four prompt templates: a shared system prompt, an initial-summary prompt, an update-summary prompt, and a turn-prefix prompt. For brevity, we reproduce the shared system and turn-prefix prompts below. The turn-prefix prompt is appended to the serialized history in the summary request's user message.

Text 1. Shared system prompt

```
You are a context summarization assistant.  Your task is to read a
conversation between a user and an AI assistant, then produce a
structured summary following the exact format specified.

Do NOT continue the conversation.  Do NOT respond to any questions
in the conversation.  ONLY output the structured summary.
```

Text 2. Turn-prefix prompt

```
This is the PREFIX of a turn that was too large to keep.  The
SUFFIX (recent work) is retained.

Summarize the prefix to provide context for the retained suffix:

## Original Request
[What did the user ask for in this turn?]

## Early Progress
- [Key decisions and work done in the prefix]

## Context for Suffix
- [Information needed to understand the retained recent work]

Be concise.  Focus on what's needed to understand the kept suffix.
```

A.3 Additional Experiment Results

A.3.1 Speedup Breakdown

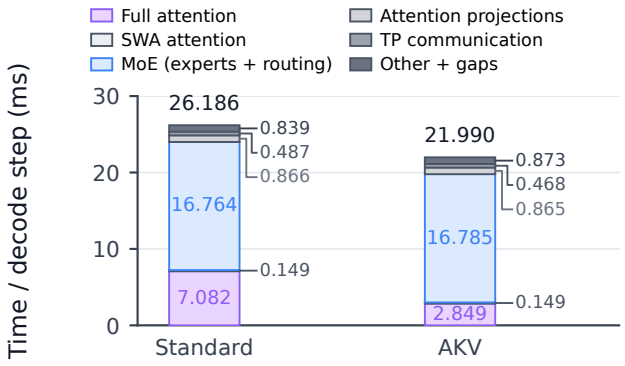


Figure 12: **Speedup breakdown.** Decode-step time for GPT-OSS-120B at the same batch size, broken down by operation.

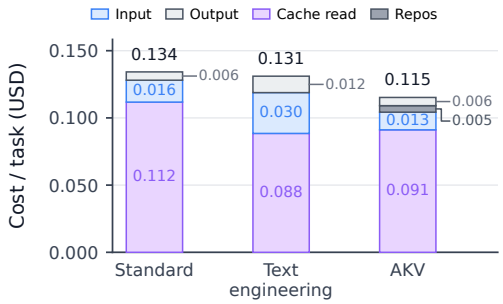


Figure 13: **Cost breakdown.** Average cost per task on BCP for Standard, Text Engineering, and AKV.

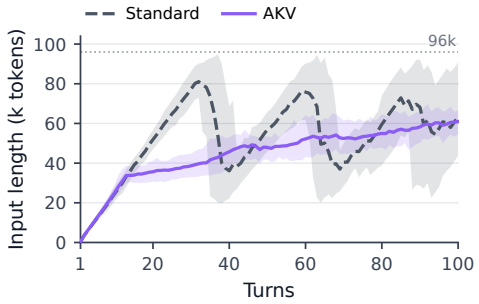


Figure 14: **Active context length by turn.** Standard and AKV on BCP over 100 turns. The dotted line marks 96k tokens.

A.3.2 Cost Breakdown and Active Context Length

Cost breakdown. AKV lowers costs by reducing cache reads and repeated prefill. As shown in Figure 13, dropping older tool-response KV reduces the amount of cached history read by subsequent requests, lowering cache-read costs by approximately 19% relative to Standard. Compared with Text Engineering, AKV reuses retained KV directly without the re-prefill required when summarization rewrites the history. This reduces extend computation and lowers input costs by approximately 57%.

Active context length. Figure 14 shows how AKV slows the growth of active context. Once tool responses exceed the retention budget of K rounds, AKV continually drops older tool-response KV while retaining the most recent K rounds and reasoning KV. The active context therefore continues to grow as reasoning accumulates, but at a substantially slower rate. In contrast, Standard accumulates history until summarization triggers a sharp reduction in context length.

A.3.3 Attention Across Message History

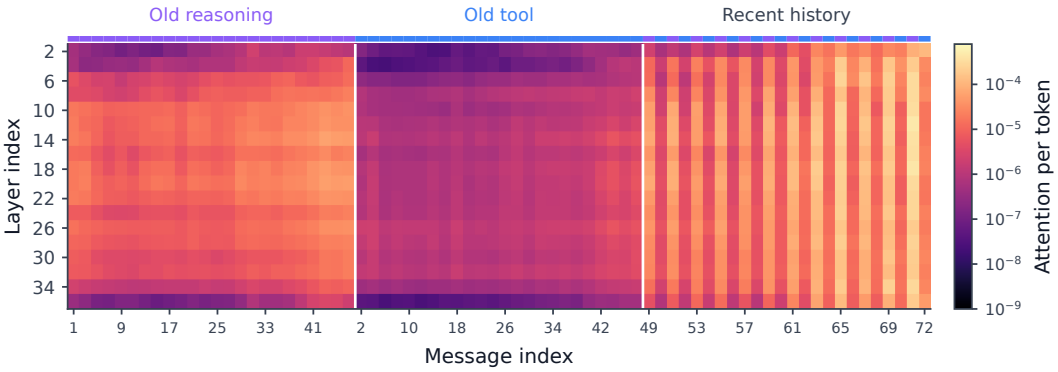


Figure 15: **Attention across message history.** Attention per token by layer and message index for an example BrowseComp-Plus request, divided into old reasoning, old tool messages, and recent history.

To explain why AKV prioritizes dropping older tool-response KV entries, we analyze the attention allocated to different types of historical messages. Figure 15 shows attention per token across layers for an example BrowseComp-Plus request. Older tool responses receive less attention than older reasoning in most displayed layers, while recent history generally receives more attention. This pattern suggests that direct attention to historical messages depends on both recency and message type, providing empirical support for retaining recent tool responses and reasoning while preferentially dropping older tool-response KV entries. Together with the task performance and cost results reported earlier, this observation helps explain how AKV reduces KV-cache access while maintaining task performance.

Compatibility with Harness. To assess whether AKV improves an existing harness, we integrate it with Pi (earendil-works, 2026) and evaluate the combination on BrowseCompPlus. For both GPT-OSS 120B and GPT-OSS 20B, adding AKV improves accuracy by 3% over the native Pi harness without compaction and 4% over the Pi harness with compaction.

A.3.4 ADDITIONAL MODELS

Table 6: Benchmark scores (pass rate, %) and costs (cents) of GPT-OSS-20B across different management methods. Bold indicates the highest score and lowest cost, including ties.

Method	BCP		GAIA-Text		SWE-Verified		xbench-DS		Average	
	Score	Cost	Score	Cost	Score	Cost	Score	Cost	Score	Cost
Standard	49.0	6.5	60.2	3.5	41.8	12.5	**25.0**	2.4	44.0	6.2
Text Engineering	43.0	**4.4**	**65.1**	3.0	25.6	**8.8**	19.0	2.0	38.2	**4.6**
AKV	**50.5**	6.0	62.1	**2.5**	**44.2**	8.9	24.0	**1.8**	**45.2**	4.8

We further evaluate AKV on GPT-OSS-20B, a smaller model from the same model family as GPT-OSS-120B, and compare its pass rate and cost against the Standard and Text Engineering baselines. As shown in Table 6, AKV achieves the highest average pass rate of 45.2%, while maintaining a comparable average cost of 4.8 cents. In particular, AKV achieves the best pass rate on BCP and SWE-Verified, and the lowest cost on GAIA-Text and xbench-DS. On several metrics where AKV does not outperform the baselines, we attribute the smaller gains to the more limited capability of GPT-OSS-20B: compared with GPT-OSS-120B, the smaller model is less able to effectively leverage the retained information under cache engineering, making it more sensitive to cache modifications.