ASYNCOMPACT: EFFICIENT CONTEXT COMPACTION FOR LONG-HORIZON LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Context compaction supports long-horizon LLM agents but can interrupt execution during summary generation and KV-cache reconstruction. We present a compaction framework that reduces these interruptions through asynchronous execution and optional KV grafting. The agent continues its task while a background branch generates a summary, then adopts the compacted history at a safe step boundary. For self-hosted inference, KV grafting further reuses cached states for the summary and retained context, with positional rebasing, instead of fully recomputing the compacted context. This reuse is approximate and requires no model retraining. Experiments on three benchmarks show broadly comparable task performance. Controlled boundary replays show that asynchronous compaction reduces mean explicit summary blocking by 73.7–84.5%. In a separate Qwen3.8-27B replay with one active request per replica, KV grafting reduces mean post-compaction resumption latency by 53.2%, including graft-specific preparation costs. These results demonstrate reduced waiting at compaction boundaries, supporting more responsive agent execution without relying on consistent full-task speedups.

1 INTRODUCTION

Large language model (LLM) agents (Wang et al., 2024) increasingly perform long-horizon tasks through sustained interaction with external environments. In applications such as repository-level software development (Zhang et al., 2024), deep research (Comanici et al., 2025), and mathematical discovery (Tzachristas et al., 2026), agents accumulate reasoning traces, tool calls, and observations over many interaction steps. These growing trajectories place pressure on context capacity, effective context utilization (An et al., 2025), and KV-cache memory. Context compaction addresses this problem by replacing older history with a concise summary while retaining recent information needed for continued execution.

Compaction, however, can interrupt the agent's progress. A synchronous agent must wait while an LLM reads the history and generates a summary. Adopting that summary introduces a second source of delay: rewriting the context prefix invalidates exact prefix-cache reuse for the affected tokens, requiring the serving system to rebuild their KV states before generation resumes. These interruptions delay the agent's next response or action, even when their contribution to total task runtime is small. For interactive applications, reducing such pauses is valuable independently of whether the complete task finishes sooner. We therefore focus on **responsiveness at compaction boundaries**: reducing explicit summarization waits and the delay before the agent resumes generation after a context switch.

Prior work addresses compaction latency through parallelism. Some of which partition a long history into blocks that can be summarized concurrently (Cim et al., 2026; Li et al., 2026), while others overlap summary generation or cache preparation with foreground execution (Pan et al., 2026; Chen et al., 2026). These approaches motivate a complementary question: when the agent adopts a compacted context, can it reuse states already computed instead of rebuilding them?

Our starting observation is that both branches already contain potentially useful KV states. The background branch computes states while generating the summary, and the foreground branch holds states for previously processed recent context and subsequent interaction tokens. However, these

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.

fragments do not form an exact cache for the compacted context. They were computed under different causal histories, and replacing the original prefix also shifts token positions. Simply concatenating them therefore differs from recomputing the compacted context (Gim et al., 2024; Yao et al., 2025). This distinction motivates *approximate* state reuse: correct positional offsets where possible, reuse available representations, and evaluate the resulting effect on model outputs and performance.

We introduce **AsynCompact**, a context-compaction framework that combines asynchronous execution with optional KV grafting. At the agent level, it appends a compaction instruction to the existing history, preserving prefix-cache reuse opportunities, and generates the summary in a background branch while the foreground agent continues working. At a safe boundary between agent steps, it adopts the summary together with retained recent context and messages accumulated during compaction. This protocol operates through standard model APIs. For self-hosted models, AsynCompact additionally supports **KV grafting**: composing cached states from the background and foreground branches, with RoPE-based positional rebasing where applicable, to avoid fully recomputing the compacted context. Grafting is an approximation to recomputation and requires serving-runtime support, but no model retraining.

We evaluate the framework on DeepSWE, NL2Repo, and Terminal Bench 2.1, using DeepSeek-v4.1-Flash through its API and locally served Qwen3.8 models. Task-level experiments show broadly comparable performance, while controlled boundary replays isolate the latency effects. Asynchronous compaction reduces mean explicit summary blocking by 73.7–84.5% across the two replayed models. In a separate Qwen3.8-27B experiment with one active request per replica, KV grafting reduces mean post-compaction resumption latency by 53.2%, including graft-specific preparation costs. These gains concern interruptions around compaction; full-task runtime does not improve consistently across benchmarks. Our contributions can be summarized as follows:

- We develop AsynCompact, an asynchronous compaction protocol that overlaps summarization with ongoing agent execution and adopts the compacted history at safe boundaries.
- We introduce a KV-grafting runtime for self-hosted inference, supporting cache slicing, composition, and positional rebasing to reuse existing states when switching to a compacted context.
- We evaluate task performance and compaction-boundary latency across API and self-hosted settings. Controlled replays demonstrate shorter summarization waits and faster resumption, while representation and output analyses characterize the approximation introduced by KV grafting.

2 METHODOLOGY

Context compaction introduces two sources of delay for long-horizon LLM agents (Figure 1): one arises from the summarization step before the agent resumes its task, and the other from KV state re-computation when adopting the compacted context. In response to the former, we develop an asynchronous compaction protocol that generates the summary in a background branch while the foreground agent continues its task. For the latter, we propose KV grafting, which initializes a new KV cache by reusing, positionally rebasing, and concatenating KV states from the summary and foreground branches rather than recomputing them through canonical replay. In this section, we first formalize the agent harness and context compaction, then describe our asynchronous compaction protocol, and finally present KV grafting for different LLM architectures.

2.1 AGENT HARNESS AND CONTEXT COMPACTION

An agent harness places an LLM in an execution loop that maintains a persistent trajectory, exposes tools, executes actions, and returns observations to the model. For concreteness, we consider a minimal coding-agent harness with a shell tool and a context manager (Yang et al., 2024), which isolates context management while supporting tasks such as repository inspection, code editing.

Although modern LLMs support increasingly long context windows, they do not ensure reliable use of all available information. Model performance can depend strongly on where relevant information appears in the context (Liu et al., 2024). Longer contexts also increase inference cost and cache occupancy. Compaction addresses these issues by replacing older interactions with a textual summary while retaining recent interactions that contribute the most to the current subtasks.

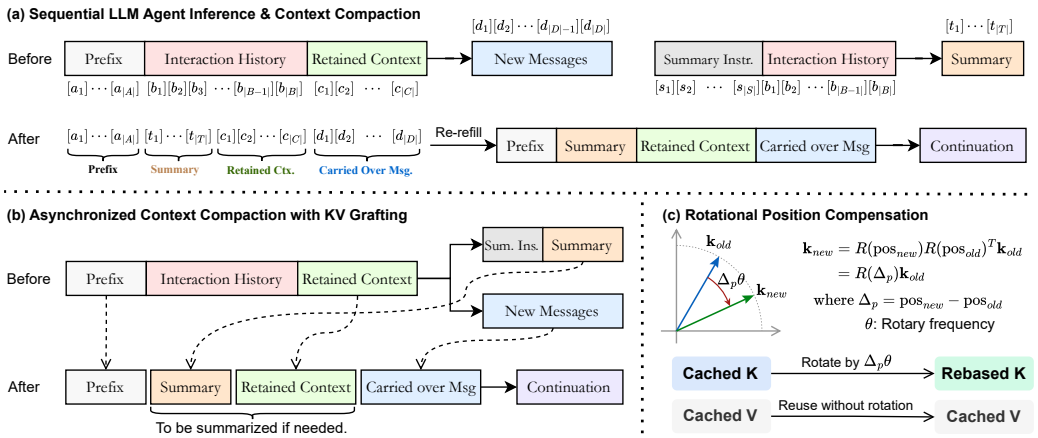

Figure 1: Context compaction for a long-horizon agent. Older interactions are replaced by a summary while the stable task prefix and recent interactions are retained.

Figure 1 (a) shows a conventional context compaction process. At the end of a completed agent interaction turn, the reasoning trajectory can be partitioned as:

$$X = A \| B \| C \,, \tag{1}$$

where $\|$ denotes concatenation, A is a stable prefix containing the system instructions and task descriptions, B is the history selected for compaction, and C is the retained recent history. The boundary between B and C is placed at such a turn boundary, ensuring that no tool actions and their outcomes are split across the two segments. As task execution proceeds, the boundary between B and C moves forward. At each subsequent compaction, B consists of the previous summary together with the newly accumulated history moved from C, thereby preserving a cumulative summary of the earlier trajectory.

Compaction is triggered when the estimated context length exceeds a predefined threshold. An individual tool output that is too large can also be compacted separately, without waiting for the overall context to reach this threshold. Both operations update the active context while preserving the raw trajectory.

2.2 Asynchronous Context Compaction

Summarization introduces two sources of overhead: the prefill cost of processing the interaction history and the decoding latency of generating the summary. To mitigate these costs, our protocol makes two complementary design choices (Figure 1 (b)).

Appended summarization request. A summarization call that replaces the original system prompt with a compaction instruction changes the serialized prefix for the interaction history, thereby requiring recomputation of the KV states for that history. To mitigate such re-prefill cost, we instead preserve $A \| B$ from task execution and append a summarization instruction U. The instruction redirects the model to preserve task-relevant facts, constraints, observations, and unresolved issues. Formally, we have:

$$A \| B \| U \longrightarrow T \,, \tag{2}$$

where T is the generated summary.

This construction makes the existing $A \| B$ prefix eligible for cache reuse. Actual reuse requires identical tokenization and prompt serialization, including system instructions and tool definitions, as well as a resident cache entry. We therefore retain the main agent's request format and tool policy; any tool calls emitted by the summarizer are ignored.

Concurrent execution and merge In parallel with summary generation, the main agent continues task execution from its current context and produces additional interactions D:

$$A \| B \| C \longrightarrow D. \tag{3}$$

The summary request uses a fixed snapshot of $A\|B$; it does not include C or the evolving D. Once T is ready and the main agent reaches a later completed turn boundary, subsequent model calls use

$$X' = A\|T\|C\|D. \tag{4}$$

All interactions completed in the meantime are thus preserved. We allow at most one pending compaction per agent to avoid overlapping replacements of the same history. A synchronous control uses the same summarization policy but waits for the summary before the next main-agent call.

Asynchronous execution hides summary-generation latency behind model calls and tool execution. Its end-to-end benefit depends on the amount of overlap, serving contention, and merge cost. In particular, replacing B with T changes the prefix of $C\|D$, so ordinary prefix caching still requires those tokens to be processed under the new context. KV grafting addresses this remaining cost.

2.3 KV GRAFTING

Replacing history B with its summary T changes the prefix of $C\|D$, even though these messages remain unchanged. Conventional compaction therefore recomputes their KV states under the new prefix $A\|T$. We refer to recomputation from the compacted text as *canonical replay*.

We propose *KV grafting* to avoid this repeated prefill (Figure 1 (c)): reuse the KV states computed for $C\|D$ during task execution and adjust their positional encodings to match the compacted sequence.

The reused states remain conditioned on the original history B, so they are generally different from those obtained by canonical replay. We hypothesize that **their retained dependence on B can preserve task-relevant information omitted from the textual summary T.**

Reusing attention state. For a causal attention head (Vaswani et al., 2017),

$$\mathrm{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}, \tag{5}$$

where d_k is the key dimension and $\mathbf{M}$ is the causal mask. Autoregressive inference caches the keys and values of processed tokens. Grafting transfers selected entries to the new sequence layout and updates their positional encoding without recomputing their underlying hidden representations.

For rotary position embeddings (RoPE) (Su et al., 2024), let $\mathbf{k}_i^{\mathrm{old}} = R(p_i^{\mathrm{old}})\bar{\mathbf{k}}_i$ be a cached key, written as a column vector. Here $R(p)$ is the model's rotary transformation at position p and $\bar{\mathbf{k}}_i$ is its pre-rotation representation. Moving this entry to position p_i^{new} gives

$$\begin{aligned} \mathbf{k}_i^{\mathrm{graft}} &= R(p_i^{\mathrm{new}})R(p_i^{\mathrm{old}})^{-1}\mathbf{k}_i^{\mathrm{old}}, \\ \mathbf{v}_i^{\mathrm{graft}} &= \mathbf{v}_i^{\mathrm{old}}. \end{aligned} \tag{6}$$

For standard RoPE with fixed frequencies, the key transformation reduces to $R(p_i^{\mathrm{new}} - p_i^{\mathrm{old}})$ and can be applied as elementwise two-dimensional rotations. Only the rotary dimensions are transformed; the remaining key dimensions and all value entries are preserved. This corrects the explicit positional encoding of a donor key, but does not undo its dependence on the donor's earlier context.

We reuse the KV states of T from the summarization branch and those of $C\|D$ from the main branch. Each span is repositioned independently to form the compacted context $A\|T\|C\|D$.

2.4 BEYOND FULL ATTENTION

Modern LLMs use a range of attention designs to improve long-context inference efficiency. We consider two representative approaches: *hybrid attention* and *sparse attention*. Their implementations vary across models, so we use Qwen3.8-27B and Qwen3.8-Flash-Next as concrete examples to illustrate how KV grafting adapts to the state maintained by each design.

Hybrid attention Qwen3.8-27B interleaves three Gated DeltaNet (GDN) layers (Yang et al., 2025) with one full-attention layer. For a GDN head, we write its recurrence using $\mathbf{S}_t \in \mathbb{R}^{d_k \times d_v}$:

$$\mathbf{S}_t = \alpha_t\left(\mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top\right)\mathbf{S}_{t-1} + \beta_t\mathbf{k}_t\mathbf{v}_t^\top, \tag{7}$$

where α_t controls state decay and β_t controls the delta update. Unlike a full-attention cache, the recurrent matrix has a size independent of sequence length. Reconstructing it under the compacted text nevertheless requires processing that text through the model.

We apply Eq. (6) to the full-attention layers and initialize each GDN layer from the main branch's terminal state after D:

$$\mathbf{S}_{\ell}^{\text{graft}} \leftarrow \mathbf{S}_{\ell,\,|A\|B\|C\|D|}^{\text{main}}, \tag{8}$$

where ℓ indexes a GDN layer. We also reuse the associated convolution state from the main branch at the end of D. Together, these states allow the GDN layers to resume without replay.

Sparse attention Qwen3.8-Flash-Next combines GDN with Qwen Sparse Attention (QSA) (Qiu et al., 2026). Its indexer uses multiple query heads and a shared key head to select small blocks of tokens. Let $\mathbf{z}_j$ denote an unrotated indexer key, distinct from the main attention key. For a complete block of r tokens starting at $p_b = b \cdot r$, the cached index key is

$$\widehat{\mathbf{k}}_b = R_{\text{idx}}(p_b)\ \text{RMSNorm}\left(\frac{1}{r}\sum_{j=p_b}^{p_b+r-1}\mathbf{z}_j\right), \tag{9}$$

where R_{idx} acts on the indexer's rotary dimensions. For a query at position i, eligible complete blocks satisfy $p_b + r - 1 \le i$. Their scores are

$$s_{ib} = \frac{1}{\sqrt{d_{\text{idx}}}}\sum_{h=1}^{H_{\text{idx}}}\text{ReLU}\Big((\widehat{\mathbf{q}}_i^{\,h})^{\top}\widehat{\mathbf{k}}_b\Big), \tag{10}$$

where $\widehat{\mathbf{q}}_i^{\,h}$ is a normalized, position-encoded indexer query and d_{idx} is its dimension. The highest-scoring blocks are expanded into token indices; attention is evaluated over those tokens and the visible incomplete tail. Our supported configuration uses $r = 4$.

Grafting relocates both the main attention keys and the compressed index keys, using token positions and block-start positions, respectively. Because each compressed index key represents r consecutive tokens, we align the splice to block boundaries to preserve the grouping of reused tokens. For a target prefix P and a donor suffix $Z[s:]$, the aligned target is

$$Y = P\big[0 : r\lfloor |P|/r \rfloor\big] \| Z\big[r\lceil s/r\rceil : |Z|\big]. \tag{11}$$

Here, slices use zero-based, half-open indices. The rule removes at most $r - 1$ tokens from each side of the splice, leaving the retained donor blocks intact.

Due to the limited space, the engineering details are provided in Appendix A.1.

3 Experiment

In this section, we conduct experiments to verify our proposed method. We conducted two experiments: one for the use of close-source LLM agents, i.e., Online API service; the other for agent serving system. We use API to verify the benefit from sync to asycn. We use open-source LLMs to verify the benefit of KV grafting.

3.1 API-Based LLM Agents

We evaluate asynchronous compaction in an API-only setting using DeepSeek-v4.1-Flash through its official API. Experiments cover DeepSWE (Huang et al., 2026), NL2Repo (Ding et al., 2025), and Terminal Bench 2.1 (Merrill et al., 2026). We report task-level pass rates for DeepSWE and Terminal Bench 2.1. For NL2Repo, we report the macro-average test pass rate: the fraction of tests passed within each task, averaged across tasks.

Table 1: DeepSeek API performance (%; mean ± SD over three independent runs).

Benchmark	Sync	Async
DeepSWE	63.95 ± 2.84	66.37 ± 3.17
NL2Repo	58.19 ± 1.59	58.16 ± 1.80
TB2.1	74.16 ± 2.13	76.40 ± 1.97

Table 2: Compaction-boundary replay on 50 DeepSWE trajectories, with three paired repetitions per boundary (150 pairs per model). Times are condition means; reductions are computed from the unrounded means. Blocking time measures explicit waiting for the summary. Boundary-window latency covers summary generation, three main-model requests, and recorded tool and controller delays, allowing overlap under async. Qwen history-KV reconstruction is excluded.

	Compaction blocking time (s)		Boundary-window latency (s)	
Model	Sync → Async	Reduction	Sync → Async	Reduction
DeepSeek-v4.1-Flash API	$11.77 \rightarrow 1.82$	**84.5%**	$57.50 \rightarrow 50.79$	**11.7%**
Qwen3.8-27B	$26.47 \rightarrow 6.97$	**73.7%**	$99.37 \rightarrow 81.46$	**18.0%**

Task performance and runtime Table 1 shows broadly comparable performance under synchronous and asynchronous compaction. Async achieves higher mean scores on DeepSWE and Terminal Bench 2.1, with nearly identical scores on NL2Repo.

The agent-runtime ECDFs in Figure 5, drawn from one run per condition, show no consistent runtime advantage across benchmarks. Task runtime also depends on the generated trajectory, and tool execution. We therefore use controlled compaction-boundary replays to isolate the latency benefit of asynchronous compaction.

Latency results We replay 50 compaction boundaries from 50 DeepSWE trajectories, with three paired sync/async repetitions per boundary. Each window covers summary generation, three main-model requests, and recorded tool and controller delays. Sync waits for the summary, whereas async overlaps its generation with ongoing execution. Both conditions share captured histories, reference summaries, tool outputs, and delays.

To control variability in API outputs and serving latency, we additionally replay the same boundaries with Qwen3.8-27B under controlled local serving using fixed output token sequences. Before each condition, we reconstruct the pre-boundary history KV cache, excluding this preparation from timing. We measure *compaction blocking time*, the explicit wait for summary completion, and *boundary-window latency*, the total elapsed time including any remaining summary wait.

Table 2 shows that async reduces compaction blocking time by 84.5% for DeepSeek and 73.7% for Qwen. Boundary-window latency decreases by 11.7% and 18.0%, saving 6.71 and 17.91 seconds per window, respectively. Across three-repeat averages, async is faster on 49/50 DeepSeek boundaries and all 50 Qwen boundaries. These results support a latency benefit at compaction boundaries under both API access and controlled local replay, without implying full-task speedup.

3.2 Open-Source LLM Agents: KV Grafting

We evaluate KV grafting in a self-hosted agent system using Qwen3.8-27B and Qwen3.8-Flash-Next. The former is a dense model with 64 layers, while the latter uses a 48-layer mixture-of-experts backbone. Both interleave three linear-attention layers with one attention layer, using full attention in Qwen3.8-27B and Qwen sparse attention in Qwen3.8-Flash-Next. These models allow us to examine grafting across dense and MoE architectures with different attention implementations.

We use the same agent implementation and asynchronous compaction policy in both conditions, enabling or disabling graft reuse while keeping the remaining settings fixed within each model–benchmark comparison. Table 5 summarizes the benchmark, generation, compaction, and serving configurations. For every model, benchmark, and graft setting, we conduct three independent complete benchmark runs and report the mean and standard deviation of task accuracy.

3.2.1 Performance

Task performance Table 3 reports accuracy across the three benchmarks. KV grafting improves mean accuracy in four of the six settings. The two decreases occur on NL2Repo with Qwen3.8-27B (2.29 percentage points) and Terminal Bench 2.1 with Qwen3.8-Flash-Next (1.20 points). Across all settings, changes range from -2.29 to $+2.35$ percentage points, suggesting broadly comparable task performance with effects that depend on the model and benchmark.

Table 3: Task performance and compaction statistics. Acc. is accuracy (%), reported as mean ± standard deviation over three independent complete benchmark runs per setting. CR is the percentage of tasks triggering compaction. RT is cumulative prefill token positions reused across successful graft commits, in millions. CR and RT describe individual graft runs rather than three-run averages; RT measures gross reuse without deducting preparation computation.

	DeepSWE			NL2Repo			Terminal Bench 2.1		
	Acc.	CR	RT	Acc.	CR	RT	Acc.	CR	RT
Qwen3.8-27B									
w/o KV graft	41.15 ± 0.62	–	–	44.42 ± 2.84	–	–	65.12 ± 0.73	–	–
w/ KV graft	42.93 ± 0.61	91	8.1	42.13 ± 1.53	91	11.4	67.42 ± 0.69	21	1.3
Qwen3.8-Flash-Next									
w/o KV graft	56.64 ± 1.25	–	–	53.62 ± 2.30	–	–	78.73 ± 1.98	–	–
w/ KV graft	58.41 ± 2.50	93	5.4	55.97 ± 1.41	97	12.5	77.53 ± 1.54	17	0.7

Post-compaction latency We evaluate the latency benefit of KV grafting by replaying DeepSWE compaction boundaries with Qwen3.8-27B, both across tasks and under different concurrency levels. Our primary metric is *resumption latency*: the elapsed time from the start of the replayed compaction-boundary operation to the first token of the next response. This includes graft-specific rendering, cache warmup, commit, checkpoint release, and queueing, while excluding common history reconstruction and final experimental session cleanup.

We first replay 81 boundaries from 48 trajectories, with three repetitions per boundary, yielding 243 paired comparisons. Each pair uses the same compacted input and generates 128 tokens per condition; output length is fixed, but generated content may differ. Eight independent replicas process boundaries in parallel (TP$=1$, DP$=8$), with at most one model request per replica and both conditions of each pair evaluated on the same replica. Mean resumption latency decreases from 12.273 to 5.746 seconds (53.18%). The mean time from the boundary operation to completion of the 128-token response also decreases, from 12.963 to 6.452 seconds (50.22%).

We then examine the effect of serving load by varying client concurrency over 1, 2, 4, and 8 on a single GPU, with each graft/no-graft pair evaluated on the same GPU.[1] Segments are issued in synchronized waves whose size equals the concurrency level. In addition to resumption latency, we report *burst throughput*, defined as completed segments divided by the sum of wave durations, including preparation, queueing, and the fixed continuations. Table 4 shows that grafting reduces mean resumption latency by 70.8–79.4% at concurrency levels 1–4, including preparation overhead, while increasing burst throughput by 3.25–$4.09\times$.

Grafting benefits depend on the availability of reusable donor state: cache eviction can require recomputation and reduce latency savings. At concurrency 8, the evaluated warmup-based implementation times out before any graft commit, with diagnostics showing high KV-cache occupancy and stalled request processing. We therefore report no latency reduction for this condition. The successful runs demonstrate reduced post-compaction waiting, including graft-specific preparation costs. These measurements characterize compaction-boundary latency and finite burst throughput, rather than full-task end-to-end speedup or steady-state serving throughput.

3.2.2 Why Does KV Graft Work?

We investigate why KV grafting can preserve predictive behavior despite differences in cached representations. Using saved trajectories from eight DeepSWE tasks with Qwen3.8-27B, we select one compaction event from the saved trajectory and reconstruct the corresponding grafted (G) and recomputed cache (R) states.

Differences in cached representations We quantify representation differences using relative L2: $d(X^G, X^R) = \frac{\|X^G - X^R\|_F}{\|X^R\|_F}$, where squared norms are pooled over the positions, layers, and heads

[1] To hold workload composition constant across concurrency levels, we use the same fixed set of eight compaction boundaries, each from a distinct trajectory in the capture collection.

Table 4: Concurrency study with Qwen3.8-27B. Each condition replays the same eight compaction segments once. **Latency stands for the mean resumption latency**. Latency reductions and burst-throughput ratios are relative to the no-graft baseline and include graft-specific preparation costs.

Concurrency	Latency w/o graft (s)	Latency w/ graft (s)	Latency reduction	Throughput ratio
1	12.20	2.51	79.4%	3.25×
2	18.87	4.55	75.9%	3.71×
4	29.68	8.66	70.8%	4.09×

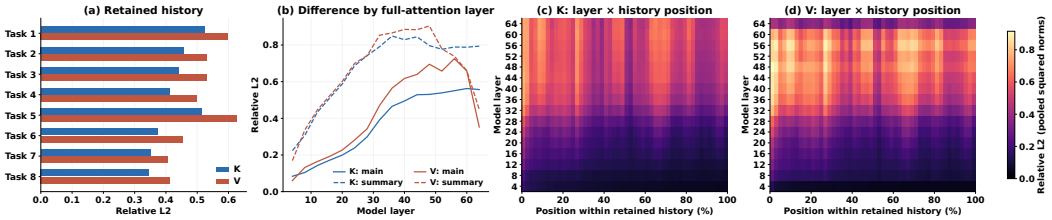

Figure 2: Relative L2 differences between grafted and recomputed KV. (a) Retained-history differences by task. (b) Layerwise differences for retained-history and summary KV. (c–d) Differences in K and V across layers and relative positions within the retained history.

included in each comparison. Figure 2 shows substantial differences between grafted and recomputed KV. Within the retained history, V differs more than K across all eight tasks. Differences generally increase with depth, although V exhibits a pronounced decline at the final layer. Summary KV has larger pooled relative differences than retained-history KV. The heatmaps further reveal considerable variation across token positions, indicating that these differences are unevenly distributed. KV differences alone, however, do not determine predictive behavior. Recomputed KV provides a reference execution, not a ground-truth target for generation. We therefore examine how these representation differences affect the model's conditional output distributions.

Differences in output distributions To isolate the effect of cache differences on next-token predictions, we compare the two executions using identical input tokens. For each selected compaction event, we first generate a 128-token reference sequence by greedy decoding under R. We then replay this sequence under both R and G using teacher forcing. At each position t, both executions receive the same compacted prompt and reference prefix, and we record their next-token distributions before supplying the next reference token. Let P_t^R and P_t^G denote these distributions, obtained by applying softmax to the full-vocabulary logits at temperature 1. We measure their divergence as $D_{\mathrm{KL}}(P_t^R \| P_t^G) = \sum_{v \in \mathcal{V}} P_t^R(v) \log \frac{P_t^R(v)}{P_t^G(v)}$. As shown in Figure 3, most positions exhibit small distributional differences, with occasional larger deviations. Across all 1,024 positions, mean and median $D_{\mathrm{KL}}(P^R \| P^G)$ are 0.128 and 0.0077, respectively, and top-1 agreement is 89.26%. Restricting the analysis to the 1,018 positions through the first EOS yields a mean KL of 0.118 and agreement of 89.39%. Thus, substantial KV differences can co-exist with close predictive agreement at most

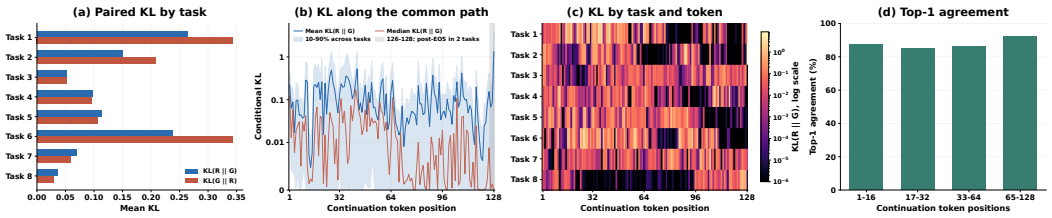

Figure 3: Predictive differences along shared 128-token continuations. (a) Mean KL divergence in both directions by task. (b) Forward KL across continuation positions. (c) Forward KL by task and position. (d) Top-1 agreement within position intervals. Gray shading in (b) marks positions after EOS in two tasks.

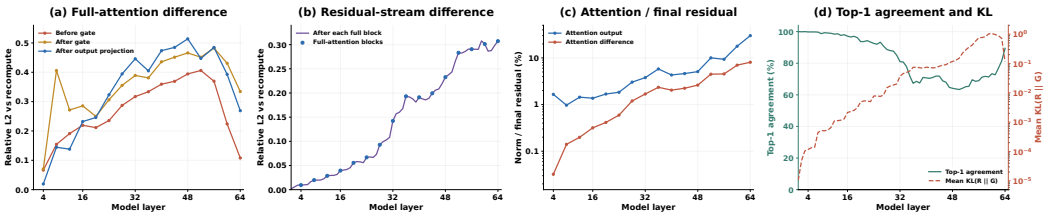

Figure 4: Representation and predictive differences across layers. (a) Relative L2 differences in full-attention outputs before gating, after gating, and after output projection. (b) Relative L2 differences in the residual stream after each block. (c) Attention-output and attention-difference norms normalized by the final pre-RMSNorm residual norm. (d) Layerwise top-1 agreement and mean KL, using the final RMSNorm and language-model head. All panels use the same 1,018 positions.

positions. These measurements characterize conditional predictions along a shared history, rather than agreement between independently generated trajectories.

Propagation through the residual stream We next examine whether residual connections (He et al., 2016) help explain this behavior. An attention sublayer updates the residual stream as $r_\ell^+ = r_\ell + a_\ell$, where a_ℓ is the attention output after gating and output projection. This motivates examining both the relative change in the attention output and its magnitude compared with the residual stream. **Figure 4 (a–b)** shows that relative L2 differences reach approximately 0.51 for projected attention outputs and 0.31 for the final residual. To compare their scales, panel (c) reports $\frac{\|a_\ell^R\|_F}{\|r_{\text{final}}^R\|_F}$ and $\frac{\|a_\ell^G - a_\ell^R\|_F}{\|r_{\text{final}}^R\|_F}$, where r_{final}^R denotes the final residual before RMSNorm. At layer 48, the attention-output difference is 51.4% of the reference attention norm but only 2.61% of the final residual norm. Thus, a substantial relative change within an attention branch can be modest on the scale of the final representation. Panel (d) shows that predictive agreement improves in the final layers. From layer 60 to layer 64, mean KL decreases from 1.048 to 0.118, while top-1 agreement increases from 73.38% to 89.39%. These intermediate-layer readouts are diagnostic: low KL at earlier layers can partly reflect diffuse distributions and should not be interpreted as evidence of accurate predictions.

Substantial KV differences can coexist with close next-token predictions under a shared token history. The relative scale of attention perturbations and the recovery of predictive agreement in the final layers offer a plausible explanation for this robustness, although these observations do not establish a causal role for residual connections.

4 Related Work

Long-horizon agent harness manages growing trajectories through structured memory (Hu et al., 2025) and context compaction (Li et al., 2026). Recent work further advances context compaction efficiency by parallelizing the summarization of history blocks (Cim et al., 2026; Li et al., 2026) and overlapping compaction with task execution (Pan et al., 2026; Chen et al., 2026). Complementary work on KV-cache management reuses KV states under evolving contexts (Srivatsa et al., 2025). CacheBlend (Yao et al., 2025) combines precomputed KV caches for retrieved RAG chunks, selectively recomputing token states to recover cross-chunk attention; ProphetKV prioritizes such recomputation according to query relevance (Wang et al., 2026). Unlike these RAG-oriented methods, our work targets online agent context compaction: it constructs the KV cache of a compacted trajectory by transforming and concatenating live KV states, avoiding re-prefill latency at context-switch time.

5 Conclusion

We presented a context compaction framework that reduces interruptions in long-horizon LLM agents. Asynchronous compaction overlaps summarization with ongoing execution, while KV grafting reuses existing states to accelerate context adoption in self-hosted inference. Experiments demonstrate shorter compaction waits and faster resumption with broadly comparable task performance. These benefits improve responsiveness at compaction boundaries, even when they do not translate into consistent full-task speedups.

AI USE STATEMENT

In this work, we used generative AI tools for polishing writing and running experiments. We have not used generative AI tools for generating research idea, and [the rest of the required disclosure tasks] are not applicable to this work. Additionally, we used generative AI tools for generate tables and figures. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Chenxin An, Jun Zhang, Ming Zhong, Lei Li, Shansan Gong, Yao Luo, Jingjing Xu, and Lingpeng Kong. Why does the effective context length of LLMs fall short? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=eoln5WgrPx.

Zhuofu Chen, Rui Pan, Yinwei Dai, and Ravi Netravali. Slipstream: Trajectory-grounded compaction validation for long-horizon agents, 2026. URL https://arxiv.org/abs/2605.08580.

Musa Cim, Burak Topcu, Chita Das, and Mahmut Taylan Kandemir. Parallel context compaction for long-horizon llm agent serving, 2026. URL https://arxiv.org/abs/2605.23296.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, and et. al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference. In P. Gibbons, G. Pekhimenko, and C. De Sa (eds.), *Proceedings of Machine Learning and Systems*, volume 6, pp. 325–338, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/file/a66caa1703fe34705a4368c3014c1966-Paper-Conference.pdf.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1575. URL https://aclanthology.org/2025.acl-long.1575/.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Tianjian Li, Jingyu Zhang, William Jurayj, Xi Wang, Chuanyang Jin, Mehrdad Farajtabar, Eric Nalisnick, and Daniel Khashabi. Self-compacting language model agents, 2026. URL https://arxiv.org/abs/2606.23525.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

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, 2026. URL https://arxiv.org/abs/2607.00151.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, et al. On the design of qwen3. 8-next architecture: Evaluation, efficiency, and training stability. *arXiv preprint arXiv:2608.30320*, 2026.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=meKEKDhdnx.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Ioannis Tzachristas, Georgios Tzachristas, and Aifen Sui. Open problems solved by LLMs? a survey of verifiable mathematical discovery. In Yanai Elazar, Allyson Ettinger, Nora Kassner, and Sebastian Ruder (eds.), *Proceedings of The Big Picture v2: Crafting a Research Narrative*, pp. 10–21, San Diego, CA, USA, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-416-3. doi: 10.18653/v1/2026.bigpicture-main.2. URL https://aclanthology.org/2026.bigpicture-main.2/.

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

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL https://doi.org/10.1007/s11704-024-40231-1.

Shihao Wang, Jiahao Chen, Yanqi Pan, Hao Huang, Yichen Hao, Xiangyu Zou, Wen Xia, Chongyang Qiu, Wentao Zhang, and PengFei Wang. ProphetKV: User-query-driven selective recomputation for efficient KV cache reuse in retrieval-augmented generation. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=SwGwmKIgUb.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *International Conference on Learning Representations*, volume 2025, pp. 29687–29707, 2025.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 94–109, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696098. URL https://doi.org/10.1145/3689031.3696098.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13643–13658, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.737. URL https://aclanthology.org/2024.acl-long.737/.

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.

Algorithm 1 KV grafting as radix-tree path splicing

Require: Target leaf a represents A; isolated paths t and c represent T and $C\|D$

function KVGRAFT(a, t, c)
  $p_t \leftarrow$ STARTPOSITION(t)
  $p_c \leftarrow$ STARTPOSITION(c)
  DETACHPATHS(t, c) ▷ Preserve the KV slots
  REBASEKEYS($t, p_t, |A|$)
  REBASEKEYS($c, p_c, |A| + |T|$)
  CONNECT($a, \mathrm{Head}(t)$)
  CONNECT($\mathrm{Tail}(t), \mathrm{Head}(c)$)
  return a
end function

A Appendix

A.1 Engineering Implementation

Modern inference engines reuse KV caches across requests with matching prefixes. vLLM (Kwon et al., 2023) implements automatic prefix caching through hash-based block indexing, while SGLang (Zheng et al., 2024) uses a radix tree. We build on SGLang. Each radix-tree node stores a variable-length token segment and references to its physical KV slots; a root-to-node path therefore represents a cached prefix.

Radix-tree grafting. A contiguous token span in a radix tree may occupy one node or a path of nodes. If a graft boundary falls inside an edge, we split its token label and KV-slot references at the same offset. This exposes the desired path segment without recomputing its KV states.

KV grafting then splices these path segments together: the summary path T is attached after A, followed by the main-branch path $C\|D$, yielding $A\|T\|C\|D$. The nodes retain their token labels and KV-slot references, while their keys are rebased to the new positions using Eq. equation 6. Algorithm 1 illustrates this operation on isolated donor paths t and c, with a denoting the end of the target prefix A. Message delimiters are omitted for clarity.

Head and Tail return the first and last nodes of a path. $\mathrm{Connect}$ adds a parent–child link. The tree operation changes how the cached segments are connected; the positional update changes their keys while preserving their values.

Paged KV cache. The physical cache is allocated in fixed-size pages, so logical path splicing must also preserve a valid mapping from token positions to physical slots. Complete pages can be reassigned when source and target page offsets match. Boundary fragments and shifts in page offsets require repacking the transferred KV entries using bounded scratch space. This avoids allocating a second full target cache, although repacking still incurs memory traffic. Values are preserved, while keys receive the positional transformations described above.

For a target of n tokens and page size p, we publish the first $p\lfloor n/p \rfloor$ tokens and replay the remaining $n \bmod p$ tokens on continuation. Hybrid-model checkpoints must correspond to that same published boundary. Thus, the final continuation tail is shorter than one page; obtaining a missing donor checkpoint may require additional processing. Physical page alignment is separate from QSA's compression-block alignment. For QSA, any boundary trimming is validated against the rendered messages and persisted in subsequent requests; graft and replay controls use the same edited target. The donor's terminal incomplete compression group and pending indexer state are retained for continuation.

Speculative decoding. Our Qwen implementations use native multi-token prediction (MTP) for speculative decoding, which maintains a draft KV cache in addition to the main model's cache. We transfer the draft KV for the committed token prefix alongside the main-model KV, using the same logical token mapping and the appropriate positional offsets for each model.

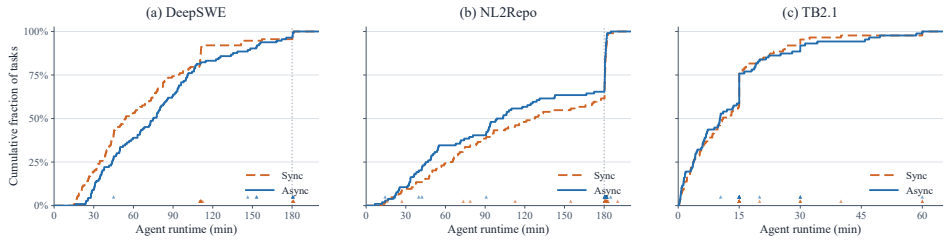

Figure 5: Agent-runtime ECDFs for DeepSeek-v4.1-Flash with synchronous and asynchronous compaction. Each panel compares the same tasks in both conditions: 113 for DeepSWE, 104 for NL2Repo, and 87 for Terminal Bench 2.1, using one run per condition. Curves include observed timeout and error terminations, marked by triangles. Vertical dashed lines indicate the 180-minute task budget for DeepSWE and NL2Repo.

Native MTP shifts its token inputs by one position, so the draft state at a splice boundary also depends on the token that follows it. We materialize this boundary with the known next token in the target sequence; for example, the summary's closing delimiter must be paired with the first retained token. This boundary processing prepares the draft cache without adding an extra token to the target text. Uncommitted draft proposals and request-local verification buffers are regenerated for the next decoding round. Speculative decoding then resumes with the grafted main-model state used for verification.

A.2 Experimental Settings

	DeepSWE		NL2Repo		Terminal Bench 2.1	
Setting	27B	Flash-Next	27B	Flash-Next	27B	Flash-Next
Benchmark and execution						
Tasks per run	113		104		89	
Independent runs per setting	3		3		3	
Execution environment	E2B		E2B		E2B / Harbor	
Agent concurrency	16		16		16	
Internal agent limit (s)	3,240		3,240		3,600	
External agent limit (s)	3,300		3,300		$\min(t_{\text{official}}, 3{,}600)$	
Tool-command timeout (s)	300		300		300	
Model-request timeout (s)	900		900		900	
Generation						
Maximum output tokens	32,768		65,536		65,536	
Temperature / top-p / top-k	1.0 / 0.95 / 20		1.0 / 0.95 / 20		1.0 / 0.95 / 20	
Min-p / presence / repetition penalty	0 / 0 / 1		0 / 0 / 1		0 / 0 / 1	
Thinking / thinking preservation	On / On		On / On		On / On	
Requested reasoning effort	`xhigh`		`xhigh`		`xhigh`	
Streaming transport	On	Off	On	Off	On	Off
Compaction						
Execution mode	Asynchronous		Asynchronous		Asynchronous	
Context window (tokens)	262,144		262,144		262,144	
Reserved context budget (tokens)	131,072		131,072		131,072	
Recent-history budget (tokens)	32,768		32,768		32,768	
Preserved header messages	2		2		2	
Tool-output compaction threshold	65,536 tokens		65,536 tokens		65,536 tokens	
Maximum compaction failures	2		2		2	
Serving						
Inference engine	SGLang with KV-graft extensions					
Tensor / data / expert parallelism	1 / 8 / –	4 / 2 / 4	1 / 8 / –	4 / 2 / 4	1 / 8 / –	4 / 2 / 4
Checkpoint / KV precision	FP8 / FP8 E4M3		FP8 / FP8 E4M3		FP8 / FP8 E4M3	
Recurrent-state precision	BF16	FP16	BF16	FP16	BF16	FP16
KV page size (tokens)	64		64		64	
Prefill chunk size (tokens)	4,096		4,096		4,096	
MTP steps / draft tokens	3 / 4		3 / 4		3 / 4	
Ordinary prefix caching	On		On		On	

Table 5: Configurations for task-performance evaluation. 27B and Flash-Next denote Qwen3.8-27B and Qwen3.8-Flash-Next. Paired conditions share these settings; the graft condition enables history, summary, and MTP-draft KV reuse. Terminal Bench uses CPU single-container tasks with official CPU and memory allocations; t_{official} is the official per-task agent limit, and verification has a separate official limit. Replay experiments use the distinct protocols described below.