CHARACTERIZING AND OPTIMIZING CONTEXT COMPACTION IN PRODUCTION CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-running coding agents rely on context compaction to overcome finite LLM context windows. Existing harnesses typically perform compaction synchronously, placing expensive summarization on the critical path and disrupting prefix-cache reuse. We deep-dive into compaction behavior and costs by analyzing **13.5M** GitHub Copilot agentic coding sessions from **3.2M** users, comprising **760M** LLM calls. Although only 7.8% of sessions trigger compaction, they account for 44.2% of total token consumption. Compaction also causes substantial foreground delay and sharply reduces prefix-cache reuse.

To enable systematic evaluation, we first introduce **COMPACTIONBENCH**, a benchmark that models software evolution as persistent, multi-turn coding sessions. It enables controlled evaluation of repeated compaction while preserving executable task-level evaluation and reflecting context-growth patterns observed in production. We then present **APEC**, an asynchronous progressive compaction mechanism that prepares fine-grained, indexed summaries in the background and appends them to a stable prefix, moving compaction work off the critical path while preserving prefix reuse. APEC is harness-agnostic and treats summarization as a black box. Evaluated across frontier models, APEC improves task quality while reducing total cost by up to 25.4% and agent execution time by 9.6%.

1 INTRODUCTION

Coding agents are increasingly used for software development tasks that require long-running, iterative development, build, and testing lifecycles. As repository state, tool outputs, and conversation history accumulate across LLM calls, these agents eventually exceed their context windows. *Context compaction* enables continued execution: when the context approaches a configured threshold, the agent asks an LLM to summarize the accumulated history, replaces the original messages with the summary, and resumes from the compressed state (Anthropic, 2026; GitHub, 2026).

Motivation. Prior work primarily treats compaction as a semantic problem: how to reduce context while retaining information needed by the agent. However, *when and how* compaction is performed can be equally consequential. Existing systems typically invoke summarization synchronously and replace a large portion of the prompt at once. This stalls foreground execution and disrupts prefix-cache reuse, forcing subsequent LLM requests to rebuild KV-cache state. Consequently, compaction can introduce substantial latency and serving overhead that offsets its intended efficiency benefits.

Our Work. Our production traces show that these systems costs matter in practice. Across **13.5M** GitHub Copilot agentic coding sessions from **3.2M** users, comprising **760M** LLM calls, compaction appears in 7.8% of sessions, yet those sessions account for 44.2% of total token consumption. A compaction event reduces the prompt by 72.9% to allow the session to continue. But the default synchronous implementation comes with foreground delay and drops prefix-cache hit rate by 57.8 pp.

This motivates a systematic, end-to-end study of context compaction. Surprisingly, we find that existing frontier coding-agent benchmarks (e.g., Terminal Bench 4.0) barely trigger even a **single** compaction when paired with modern LLMs (Section 3). This exposes a fundamental evaluation challenge: compaction requires workloads that sustain context growth over sufficiently long horizons while retaining meaningful, executable task outcomes. Simply increasing task length is insufficient, as it can induce unrealistic interaction patterns or make task outcomes difficult to reproduce and evaluate. We introduce **COMPACTIONBENCH**, a benchmark that turns software evolution into persistent, multi-turn coding sessions. Rather than evaluating isolated tasks, it repeatedly evolves the same repository through *dependent* user requests, naturally growing the agent context over time.

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.

Each session retains executable repository-level tests, providing a consistent task outcome while allowing us to control session length and exercise repeated compaction. The resulting context-growth behavior also more closely matches the distribution observed in production.

We further develop **APEC**, an asynchronous, progressive compaction mechanism for coding agents. Instead of synchronously rewriting the entire context, APEC identifies completed trajectory segments and summarizes them incrementally in the background. Completed summaries are appended to an append-only compacted prefix, while the active tail remains available for continued execution. This structure moves summarization off the critical path and preserves the reusable prefix across compaction boundaries, avoiding repeated KV-cache reconstruction.

We evaluate APEC on CompactionBench across multiple frontier models. APEC improves task quality while reducing input tokens by 33.5% and total cost by 25.3% relative to Sync-H, and end-to-end time by 9.6% relative to Sync-L. APEC is harness-agnostic and treats summarization as a black box, complementary to existing efforts on LLM context compression (e.g., (Li et al., 2026a)).

In summary, our contributions are:

- **Production evidence for compaction as a systems bottleneck.** We present the largest characterization of context compaction across 13.5M coding-agent sessions from GitHub Copilot.
- **Long-horizon compaction evaluation.** We introduce CompactionBench, a production-representative compaction benchmark to evaluate the costs and benefits of compaction policies.
- **Asynchronous progressive compaction.** We design APEC, a cost-efficient compaction mechanism that preserves task quality by incrementally preparing summaries off the critical path.
- **Open-source** the first context compaction production traces and benchmarks.[1]

2 Context Compaction in Production Coding Agents

2.1 Dataset Overview

We collected agent trajectories from GitHub Copilot coding-agent sessions in Visual Studio and VS Code during the first week of June 2026. The trace records timestamps, durations, model names, token counts, cache-hit percentages, tool names, and status metadata. Table 1a presents a summary of the dataset scale. Temporal patterns and workload statistics are presented in Appendix A.

A *session* contains one or more user turns. Each turn expands as `user` → `LLM` → `[tool` → `LLM]*` → `response`. Later model calls consist of context and cached prefixes accumulated by earlier calls in the same session. Table 1b shows an overview of per-session statistics.

Table 1: Scale and workload statistics for the June 2026 production trace.

(a) Dataset scale.

Metric	Valu	Metric	Value
Sessions	13.5M	Users	3.2M
Turns	82.4M	Models / tools	27+ / 45+
LLM calls	760.5M	Tool calls	774.7M
Prompt tokens	44.9T	Completion tokens	39.3B

(b) Per-session statistics. "Skew" is mean/median.

Metric	Median	P75	P90	Mean	Skew
User turns	3	7	15	6.1	2.0×
LLM calls	15	42.8	100.5	56.3	3.8×
Tool invocations	13	45.6	111.4	57.4	4.4×
Duration (min)	4.2	39.5	177.8	62.6	14.9×

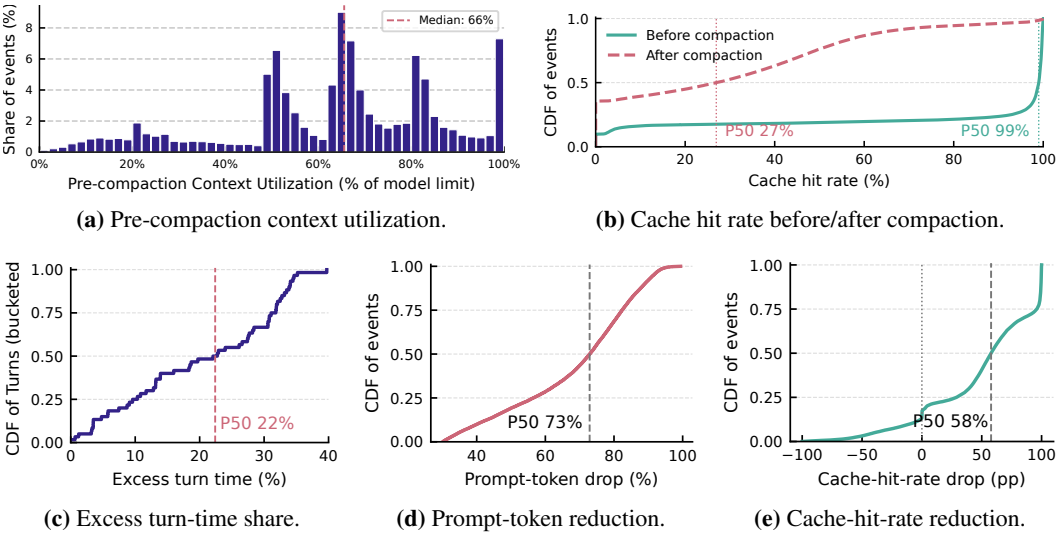
2.2 Characterizing Context Compactions

As an agentic session runs long, extensive tool output, and accumulated turn history eventually push the prompt toward the model's context limit. As context reaches the threshold, the coding agent performs compaction (Anthropic, 2026): it issues another LLM call with instructions to summarize the history and to room for the session to continue. Here we zoom into the sessions where at least one compaction happens from the production trace, more details in section A.

Trigger Behavior and Recurrence. The context utilization at which compaction fires is governed by static thresholds. Figure 1a shows the distribution of *pre*-compaction context utilization. Rather than concentrating around one value, the distribution indicates multiple thresholds, with pronounced peaks near 50%, 65–66% (the overall median), 80%, and near-100% utilization. This pattern reflects heterogeneity in compaction policies due to a mixture of model-specific triggering rules.

Substantial Impact of Context Compaction. Figure 1b-e separates three major consequences from compaction. Figure 1c reports the CDF of compaction duration as a percentage of total turn execu-

[1] https://anonymous.4open.science/r/APEC-A6D4



(a) Pre-compaction context utilization. **(b)** Cache hit rate before/after compaction.

(c) Excess turn-time share. **(d)** Prompt-token reduction. **(e)** Cache-hit-rate reduction.

Figure 1: Compaction trigger points and costs.

tion time across all observed compaction events. We find that compaction overhead is substantial and highly variable: the median compaction consumes roughly 22% of the turn's total execution time. The tail is heavy where P90 sits at approximately 34%. This overhead suggests that compaction is not a lightweight housekeeping step in agent sessions, motivating the need for optimizations like incremental compaction or overlapping strategies.

Beyond the added latency, compaction also perturbs the KV cache: the summarized history invalidates the previously cached prefix, forcing a costly cache-cold prefill on the next turn and eliminating the reuse benefits that prefix caching would otherwise provide. Figure 1b shows that the median cache hit rate is 99.0% before compaction and 27.0% afterward. When compaction does occur, it is aggressive: the median event drops 72.9% of prompt tokens, with the middle half of events (P25 to P75) removing between 58% and 81% of tokens, and 6.1% of events dropping 90% or more (Figure 1d). This rewriting comes at a steep cache cost. Sessions typically enter compaction with a near-fully cached prompt, but the rewritten prompt shares little prefix with what came before, and the cache hit rate on the first call after compaction results in a median drop of 57.8% (Figure 1e); 34.3% of compaction events erase >90% cache hit rate, and 21% almost empties the cache.

3 CompactionBench

Evaluating context compaction in agentic coding requires workloads that repeatedly build sufficient context pressure to trigger compaction throughout a session. Surprisingly, our study of more than ten frontier coding benchmarks finds that they barely trigger even a single compaction. We therefore transform suitable coding workloads into persistent, multi-turn sessions that mimic production usage and sustain context growth. To our knowledge, CompactionBench is the first benchmark specifically designed for end-to-end evaluation of repeated context compaction in coding agents.

3.1 Limitations of Existing Benchmarks

Table 2 summarizes our findings; Section D.1 describes the execution setup and measurement protocol. We identify three interaction patterns that limit repeated-compaction coverage, illustrated by the context trajectories in Figure 3. An *agentic turn* may contain many LLM calls and tool interactions; an *LLM turn* consists of one model call and its tool calls.

- **Front-loaded context.** Existing long-context benchmarks concentrate most context in the initial prompt, creating high pressure early but little subsequent growth. While this pattern may trigger an initial compaction, it fails to capture the sustained growth-compaction cycles observed in production. Figure 3a illustrates this behavior from LongSWE-Bench.
- **Single-turn autonomous execution.** Many coding benchmarks issue a task once and let the agent execute autonomously until termination. Although these trajectories may contain many LLM and tool calls, they lack continued user interaction, and often fail to generate sufficient context for repeated compaction. Figure 3b shows an example of Terminal-Bench 4.0.

Table 2: Context growth and compaction behavior across existing benchmarks. LLM and tool token counts are mean output tokens per LLM turn and per text-bearing tool call, respectively.

Benchmark	#Tasks	#LLM Tokens	#Tool Tokens	#Turns	#Compactions
(1) Front-loaded context					
A. LongSWE-Bench [256k] (Rando et al., 2025)	100	0.25K	0.62K	14.3	1
B. LoCoBench [single turn] (Qiu et al., 2025a)	174	8.90K	0K	1.0	None
(2) Single-turn autonomous execution					
C. SWE-bench (Jimenez et al., 2024)	89	0.49K	0.62K	23.5	None
D. SWE-EVO (Le et al., 2025)	27	1.04K	—	109.9	None
E. Terminal-Bench 2.1 (Merrill et al., 2026)	87	0.26K	2.01K	12.5	None
F. Terminal-Bench 4.0 (Terminal-Bench Team, 2026)	61	0.38K	1.08K	21.1	None
G. ViBench [build] (Zhong et al., 2026)	24	0.87K	0.37K	14.0	None
H. DeepSWE (Datacurve, 2026)	108	0.24K	0.73K	53.1	None
(3) Passive multi-turn continuation					
I. SWE-Marathon (Desai et al., 2026)	14	0.51K	0.66K	20.3	None
J. LHTB (Li et al., 2026b)	44	0.30K	0.34K	136.1	None
K. LoCoBench-Agent (Qiu et al., 2025b)	30	1.18K	1.79K	8.7	None

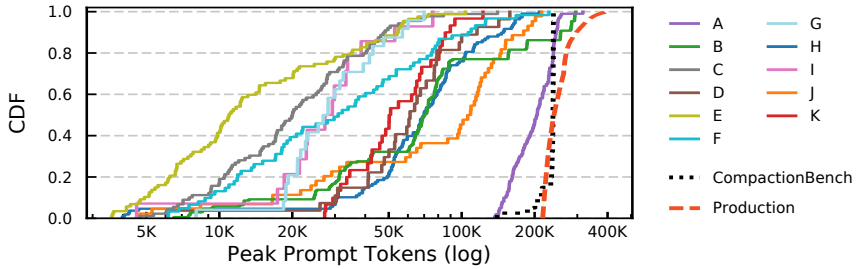
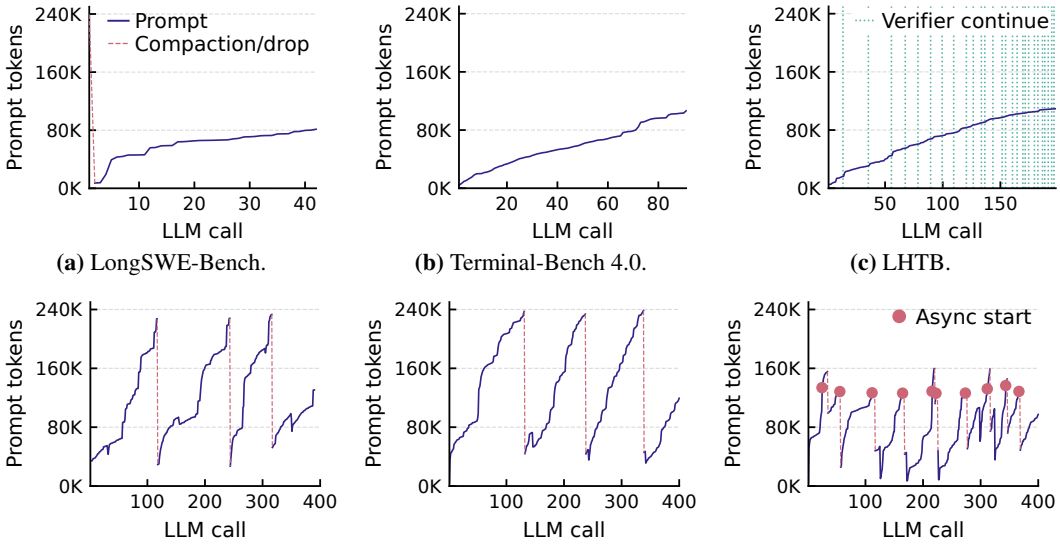


Figure 2: Observed peak prompt tokens across tasks in existing SWE benchmarks (Table 2) vs. production sessions with compactions. CompactionBench uses the eight retained tasks, with five Sync-H trials per task. Letters A-K identify the benchmark rows in Table 2.



(a) LongSWE-Bench. **(b)** Terminal-Bench 4.0. **(c)** LHTB.

(d) Production traces. **(e)** CompactionBench (Sync.) **(f)** CompactionBench (APEC)

Figure 3: Prompt-token trajectories across benchmark and production examples. CompactionBench is able to replay production-like trajectories where other frontier benchmarks fail.

- **Passive multi-turn continuation.** Recent benchmarks extend execution through *verifier* feedback, asking the agent to continue when a task remains unresolved. While verification creates additional agentic turns, it does not necessarily sustain context growth. In Figure 3c, LHTB repeatedly resumes the same task after verifier feedback; however, the number of LLM calls within each successive agentic turn gradually decreases. This trajectory stagnation produces increasingly limited new context and does not resemble sustained production interaction.

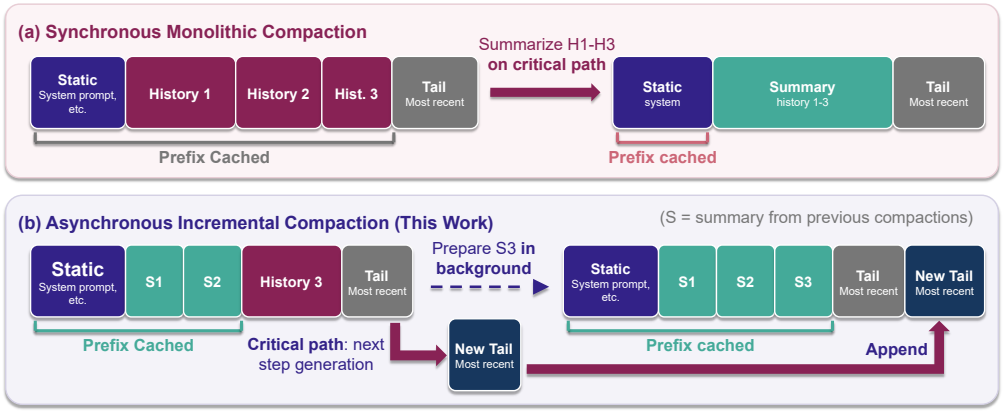


Figure 4: Synchronous compaction (top) vs. APEC (bottom). (a) A monolithic foreground compaction replaces the cached history with one summary. (b) Incremental summaries leave a stable prefix as the context grows, while the next summary is prepared in the background.

3.2 Constructing Realistic Long-Horizon Workloads

The limitations above suggest that repeated compaction requires not merely long execution, but continual introduction of new information. We therefore construct multi-turn coding sessions in which each turn introduces a new, dependent requirement to the same evolving repository, sustaining context growth while approximating iterative user-agent interaction observed in production.

Constructing such workloads requires realistic, reproducible, and evaluable task sequences. In particular, follow-up requests in the task sequence must be fixed in advance rather than generated from an individual agent trajectory, ensuring consistent workloads across benchmark evaluations.

Repository evolution across software releases naturally provides this structure. We decompose a repository's release history into changelog features and concatenate them semantically: each turn introduces a predetermined feature that extends the codebase produced by previous turns. This approach provides trajectory-independent follow-up requests while naturally modeling iterative feature development in production software engineering. It requires repositories with sufficiently long release histories, rich changelogs, and executable evaluations.

Inspired by repositories in SWE-EVO (Le et al., 2025), we select repositories with sufficiently long release spans and rich changelogs, and convert their release evolutions into multi-turn task sequences to build CompactionBench. As shown in Figures 3e and 3f, these workloads sustain repeated context growth and compaction cycles, closely resembling the production trajectory in Figure 3d.

4 APEC: Asynchronous Progressive Compaction

Motivated by measurements observed in Section 2, we design APEC to address three challenges: (1) *Blocking latency*, as rewriting the full history stalls the agent, inflating **end-to-end latency**. (2) *Cache disruption*, as replacing the prompt prefix invalidates its KV cache, raising **serving cost**. (3) *Information loss*, as truncation or summarization may discard details needed for later steps in the same agent session, degrading **task accuracy**.

APEC addresses these challenges with *asynchronous, progressive compaction*, inspired by LSM trees for transactional logging systems (O'Neil et al., 1996) (Figure 4, Algorithm 3). Section C describes the OpenClaude integration and the interfaces to port APEC to other agent harnesses.

Fine-Grained Context Segmentation. Synchronous compaction (Algorithm 2) rewrites the entire history into a single summary (Line 5) and keeps a recent tail within the remaining history budget (Lines 6–7), yielding $[sysPrompt,\ summary,\ tail]$ (Line 8). The KV cache is invalidated at each compaction. APEC instead replaces this monolithic rewrite with *progressive compaction*. As shown in Algorithm 3, APEC maintains a $prefix$ list (Line 1) that accumulates summaries across compactions rather than replacing the $prefix$ list. Each compaction snapshots a segment (the new raw history) and generates a summary using the existing prefix as context. The segment summary is appended to $prefix$ list while only messages after the snapshot remain in the $tail$ (Line 5): $[sysPrompt,\ prefix,\ newTail]$. The prefix is append-only so its KV-cache entries remain valid across compaction boundaries. APEC partitions the history into *contiguous segments*: each segment is present either in

Algorithm 1 Agent Loop.

```
1: procedure AGENTLOOP(msg, mode)
2:   while true do
3:     if mode = SYNC then
4:       msg ← SYNCCOMPACTCHECK(msg)
5:     else
6:       msg ← ASYNCCOMPACTCHECK(msg)
7:     end if
8:     resp ← CALLLLM(msg);
       STREAMTOUSER(resp)
9:     results ←
       EXECUTETOOLSIFANY(resp.toolCalls)
10:    msg ← msg ‖ resp ‖ results
11:  end while
12: end procedure
```

Algorithm 2 Synchronous Compaction.

```
1: procedure SYNCCOMPACTCHECK(msg)
2:   if COUNTTOKENS(msg) < threshold then
3:     return msg
4:   end if
5:   summary ← SUMMARIZE(msg, S_cap)
6:   H ← max(0, B − COUNTTOKENS(summary))
7:   tail ← RECENTTAIL(msg, H)
8:   return [sysPrompt, summary, tail]
9: end procedure
```

Algorithm 3 APEC.

```
1: work ← NONE; prefix ← []
2: procedure ASYNCCOMPACTCHECK(msg)
3:   tokens ← COUNTTOKENS(msg)
4:   if work.status = READY then
5:     prefix ← prefix ‖ work.summary
6:     tail ← msg[work.snapEnd+1 ..]
7:     work ← NONE
8:     return [sysPrompt, prefix, tail]
9:   end if
10:  if tokens ≥ τ_hard then
11:    summary ← SUMMARIZE(msg, S_cap)
12:    H_reset ←
       max(0, B − COUNTTOKENS(summary))
13:    tail ← RECENTTAIL(msg, H_reset)
14:    prefix ← [summary]; work ← NONE
15:    return [sysPrompt, prefix, tail]
16:  end if
17:  if tokens ≥ τ_soft
       and COUNTTOKENS(prefix) < τ_soft
       and work = NONE then
18:    (segment, tail, snapEnd) ←
       SPLITFORCOMPACTION(msg, prefix, H)
19:    work ←
       ASYNCSUMMARIZE(prefix,
         segment, S_cap)
20:    work.snapEnd ← snapEnd
21:  end if
22:  return msg
23: end procedure
```

original messages (tail) or by its summary (prefix list). Context segmentation exposes explicit controls over compaction granularity: the snapshot size, early-trigger threshold τ_{soft}, and tail budget.

Budget Notation. Let S denote actual summary-output tokens and S_{cap} the configured output cap. Let H be the retained raw-tail token budget, excluding summaries, and B the combined budget for one new summary and its raw tail, excluding the fixed prefix, earlier immutable summaries, and post-snapshot growth. If the retained summary occupies S_{ret} tokens, a filled reconstruction has $B = S_{\text{ret}} + H$. Synchronous reconstruction derives $H = \max(0, B - S_{\text{ret}})$; progressive compaction controls H independently. The pseudocode passes S_{cap} to summary generation and H to tail selection. SPLITFORCOMPACTION leaves up to H tokens of recent raw history outside the summarized segment; the application step also preserves messages generated after the snapshot.

Proactive, Non-Blocking Compaction. Synchronous compaction invokes summarization only after the context reaches its threshold, forcing the next LLM call to wait for the rewrite (Algorithm 1). APEC instead starts compaction proactively with an early threshold τ_{soft} (Algorithm 3). The runtime snapshots the current history and launches summarization in the background, while the foreground agent continues executing LLM calls and tools on the unmodified context. When the background summary becomes available (Line 4), the runtime appends it to the persistent prefix and retains only the messages generated after the snapshot as the new tail (Line 5). Thus, summarization is overlapped with foreground execution rather than placed on the critical path (Figure 4).

The threshold τ_{soft} trades *preparation slack* against *compaction frequency*: A smaller threshold provides more time for background summarization but triggers more often, while a larger threshold reduces frequency at the risk of falling onto the foreground critical path. We set $\tau_{soft} = 50\%$ and $\tau_{hard} = 90\%$ to leave sufficient slack while avoiding excessive background work.

Prefix List Compaction. The summary prefix itself consumes context space. When its size reaches τ_{soft}, APEC stops launching background compactions and allows the context to grow until τ_{hard}. It then performs a synchronous full compaction and starts a new append-only epoch.

Cost Modeling. We formulate an analytical model that quantifies the cost of APEC compared to synchronous compaction. Consider N foreground LLM calls with mean output growth g tokens per call and an initial prompt of b tokens. For this simplified model, assume retained-summary size equals output size, $S_{\text{ret}} = S$, and each compaction retains H raw-tail tokens, so $B = S + H$.

Synchronous compaction triggers at T tokens, whereas APEC snapshots $L_{\text{snap}} = S$ tokens and applies the resulting summary after ℓ foreground calls. Let π_{pf}, π_{hit}, and π_{dec} denote uncached-input, cached-input, and output prices. We assume full retention budgets, shared prefix caching, call-aligned triggers, and a single append-only epoch; fixed request-instruction overhead is omitted. The resulting compaction counts are:

$$m_{\text{s}} = \max\left\{0, \left\lfloor \frac{g(N-1) - S - H}{T - b - S - H} \right\rfloor\right\}, \qquad m_{\text{a}}(t) = \max\left\{0, \left\lfloor \frac{g(N-1-t) - H}{L_{\text{snap}}} \right\rfloor\right\}.$$

Here, $m_{\text{a}}(0)$ and $m_{\text{a}}(\ell)$ count summaries started and applied by the end, respectively. Subtracting the two total-cost expressions in Appendix E.1 cancels their common no-compaction cost $\mathcal{C}_0$. For compactness, write $D = T - b - B$, $L = L_{\text{snap}}$, $m = m_{\text{s}}$, $a = m_{\text{a}}(\ell)$, and $j = m_{\text{a}}(0)$. The exact saving $\Delta\mathcal{C} = \widehat{\mathcal{C}}_{\text{sync}} - \widehat{\mathcal{C}}_{\text{apec}}$ simplifies to

$$\begin{aligned}\Delta\mathcal{C} = \pi_{\text{hit}} \Bigg[& a(L-S)\left(N - \ell - \frac{2H + (a+1)L}{2g}\right) \\ & - mD\left(N - \frac{2B + (m+1)D}{2g}\right) + mT - j(b+L) - \frac{S\,j(j-1)}{2}\Bigg] \\ & + (\pi_{\text{pf}} - \pi_{\text{hit}})\left[(m-a)B - a(\ell-1)g\right] + \pi_{\text{dec}} S(m-j). \end{aligned} \tag{1}$$

Positive $\Delta\mathcal{C}$ favors APEC. The started and applied counts remain distinct, so this expression also charges summaries that finish after the foreground workload ends.

For a representative example, use $S = 6$K, approximating the main evaluation's mean APEC summary output, not its cap $S_{\text{cap}} = 20$K. With the reconstructed-history budget $B = 40$K, set $H = B - S = 34$K. Keep $N = 500$, $g = 1$K, $b = 4$K, $T = 240$K, $L_{\text{snap}} = 60$K, and $\ell = 2$, with prices $(\pi_{\text{pf}}, \pi_{\text{hit}}, \pi_{\text{dec}}) = (5, 0.5, 25)$ dollars per million tokens. Then $m = 2$ and $a = j = 7$, and collecting terms by per-token price gives

$$\Delta\mathcal{C} = 19{,}713{,}000\,\pi_{\text{hit}} - 207{,}000\,\pi_{\text{pf}} - 30{,}000\,\pi_{\text{dec}}.$$

Using the stated per-million-token prices, $\Delta\mathcal{C} = 19.713(0.5) - 0.207(5) - 0.030(25) = \$8.0715$. Estimated costs are **$46.50** for synchronous compaction and **$38.43** for APEC, a **17.4% saving**.

5 Evaluation

5.1 Setup

Agentic Benchmarks and Metrics. We use CompactionBench tasks for the main experiment and report averages over tasks across five repeated trials. We focus on three end-to-end metrics: (M1) task accuracy, (M2) total token cost, and (M3) task completion time. We additionally report token usage, caching, context pressure, and compaction events.

Harness. We use OpenClaude (Gitlawb, 2026) as the coding-agent harness and implement compaction on top of its agent loop, following the design by representative harnesses (e.g., Codex (OpenAI, 2026)). By default each summary has output cap $S_{\text{cap}} = 20$K tokens, and combined summary-and-tail budget $B = 40$K, the raw-tail allowance is $H = 40\text{K} - S$ (at most 20K), where S is the retained summary size. We evaluate parameter sensitivity in Section 5.3.

Models. We use GPT-5.6 Sol for the main evaluation. We use extra-high reasoning effort and otherwise default API settings. We evaluate other models in the model-family sensitivity study.

Context Limit. We use a popular 400K-token context capacity and 128k output capacity. We follow the triggering rule used by Codex and other harnesses, with compaction triggered at $(400\text{K} - 128\text{K}) \times \rho$, where ρ is the trigger ratio. We also include 1M setup in the sensitivity study.

APEC Variants and Baselines. We use H and L to denote high (90%) and low (50%) compaction trigger ratios, respectively. They include: (1) **Sync-H** and (2) **Sync-L** perform synchronous compaction. (3) **Prune-H** and (4) **Prune-L** augments Sync with a conservative, rule-based policy that deduplicates file read outputs and compresses formatting-incurred token bloat. (5) **Async-L** applies normal compaction asynchronously at the low threshold. (6) **PEC** applies blocking, progressive compaction at the low threshold while (7) **APEC** additionally applies asynchronous compactions.

Table 3: End-to-end evaluation on COMPACTIONBENCH with GPT-5.6 Sol. **Top-2** approaches for each of the end-to-end metrics are highlighted in **bold**: APEC dominates among all three metrics

Metric (Avg. across tasks & trials)	**Sync-H**	**Sync-L**	**Async-L**	**Prune-H**	**Prune-L**	**PEC**	**APEC**
[M1] Task accuracy (%)	69.29	73.04	70.70	72.40	71.52	**76.74**	**77.95**
Compactions/trial	1.10	3.35	3.15	0.95	3.15	3.05	2.90
Max. compactions	3	9	8	3	8	6	7
Total input tokens (M)	34.008	22.864	22.957	32.704	22.773	20.874	22.627
Total cached input tokens (M)	33.518	22.187	22.279	32.260	21.669	20.356	21.922
Total output tokens (K)	48.936	55.145	55.226	47.471	54.408	49.638	47.393
Token growth/LLM turn (K)	1.032	1.043	1.094	0.989	1.010	1.011	0.973
Compaction cost ($)	1.50	2.68	2.38	1.26	2.51	**0.74**	**0.70**
[M2] Total cost ($)	21.93	18.54	18.30	20.80	20.22	**14.75**	**16.37**
[M3] Task completion time (min)	29.25	30.77	28.99	**25.55**	33.24	29.89	**27.82**
Compaction time (s)	60.02	176.76	156.88	49.81	178.87	160.38	156.17
Overlapped compaction (%)	0.00	0.00	98.65	0.00	0.00	0.00	85.34

5.2 END-TO-END EXPERIMENT

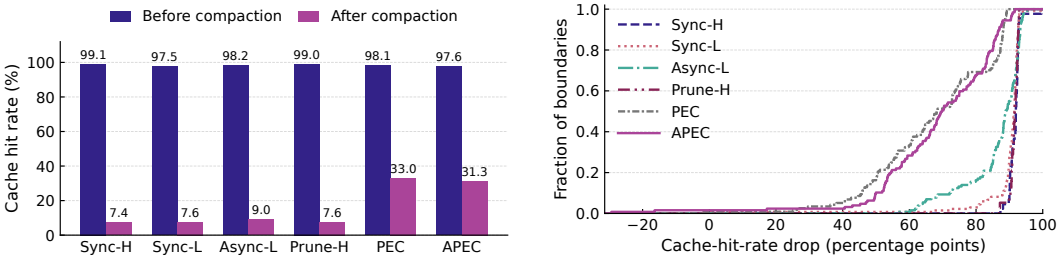
We evaluate all approaches end-to-end on COMPACTIONBENCH and report averages across tasks and 5 repeats trialTable 3. Overall, APEC is the only method that ranks in the top two on all three metrics. It achieves the highest accuracy at 77.95%, improving over Sync-H, Sync-L, and Async-L by 8.66, 4.91, and 7.25 points, respectively. It also reduces total cost by 25.3% relative to Sync-H and by 11.7% and 10.5% relative to Sync-L and Async-L. Progressive compaction lowers compaction cost from $2.68 (Sync-L) to $0.70, a 73.8% reduction, even though APEC triggers a comparable number of compactions (2.90 vs. 3.35 per trial). On latency, APEC finishes tasks in 27.82 minutes on average, 6.9% faster than its synchronous counterpart PEC (29.89 min). Compared with the other baselines, Prune-H is fastest (25.55 min) but has lower accuracy (72.40%) and higher cost ($20.80) than APEC, and PEC attains the lowest total cost ($14.75) but is slower than APEC.

Cache Hit Rate. We zoom into cache hit rate from the main result run. We measure the cache hit rate on the last foreground LLM call immediately before each compaction event and the first foreground call immediately after, measuring how much prefix structure is kept after a compaction. Figure 5(a) reports the average cache hit rate, and Figure 5(b) reports the C.D.F of cache hit rate drop. All plotted methods have cache hit rates near 100% just before compaction, as a result of sequential progressing, but monolithic compaction reduces this to 7.39–8.96% afterward (only system prompt cached after compaction). Progressive compaction (PEC and APEC) retains 33.01% and 31.28% respectively, a 3.5–4.2× improvement over the monolithic methods. Both variants preserve more prefix reuse by retaining accumulated summaries rather than rewriting the entire summary prefix.

5.3 ABLATION AND SENSITIVITY STUDY

More Models. We additionally evaluate GPT-5.4 model and observed similar trends, showing the design improvement from APEC is consistent. (Table 4).

Compaction Parameters. Using GPT-5.6 Sol, we vary APEC's summary-output allowance $S_{\text{cap}} \in \{20, 40\}$K and raw-tail budget $H \in \{20, 40, 60\}$K, keeping the accumulated-summary cap fixed at 136K. Table 5 reports the results. Top configurations achieves the highest accuracy while also being among the cheapest and fastest, yielding the best overall accuracy-per-cost-per-time efficiency. Most H/S_{cap} still out performs baselines in Table 3, showing the design of APEC is generalizable.



(a) Cache-hit rate before and after compaction **(b)** Cache-hit-rate drop distribution

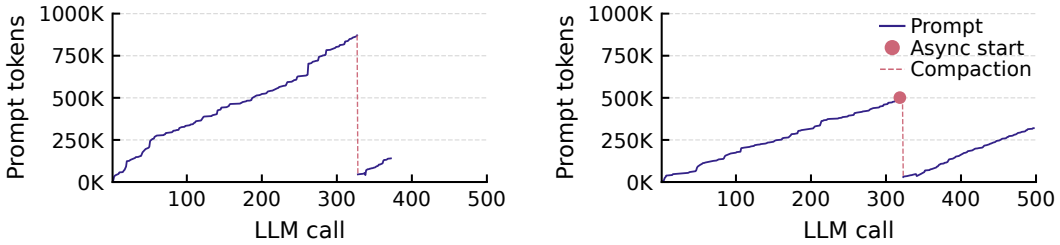
Figure 5: Cache behavior at compaction boundaries.

Table 4: Model sensitivity study on GPT-5.4.

Model	Method	Acc. (%)	Num. Comps.	Input (M)	Cached input (M)	Output (K)	Total time (min)	Compaction time (s)	Overlap (%)	Total cost ($)
GPT-5.4	Sync-H	56.70	2.44	42.706	40.164	184.67	144.47	471.47	0.00	41.06
	Async-L	52.57	7.44	35.400	33.024	302.54	117.13	710.86	98.15	42.32
	APEC	**60.88**	6.11	31.729	28.337	196.68	**61.53**	617.35	94.78	**40.64**

Table 5: APEC sensitivity to summary-output cap (S_{cap}) and raw-tail budget (H). **Top-3** configurations are highlighted in **bold** in each end-to-end metric.

S_{cap}	H	Acc. (%)	Comp.	Input (M)	Cached (M)	Output (K)	Agent (min)	Compaction (s)	Overlap (s)	Cost($)
20K	20K	77.41	3.00	24.794	23.836	57.93	81.47	140.57	139.77	18.16
20K	40K	**78.93**	2.94	22.790	22.096	49.59	**63.37**	155.79	135.86	**15.76**
20K	60K	78.36	5.22	28.606	27.289	51.25	100.93	203.10	194.22	21.51
40K	20K	**85.81**	2.57	20.346	19.610	46.50	**69.58**	123.18	121.51	**14.65**
40K	40K	**82.12**	2.67	24.409	23.425	52.43	81.98	125.61	124.37	17.94
40K	60K	68.36	3.44	26.496	25.937	54.03	**69.37**	156.68	149.68	**17.11**



(a) Sync-H. **(b)** APEC.

Figure 6: Context trajectories on an example CompactionBench task with a 1M context window.

1M Context Limit. Several heavy tasks in CompactionBench can trigger compaction even with 1M context limit, as we show a sample in Figure 6. We note that as model capabilities and context windows grow, maintaining compaction coverage may require updating the benchmark with longer, heavier tasks. We therefore call for more attention on long-horizon benchmarks.

6 Related Work

Long-horizon Agentic Coding Evaluation. Recent benchmarks increasingly target long-horizon agent interactions, including those summarized in Table 2. Additionally, StaminaBench (Sobal et al., 2026) constructs long-horizon interactions using synthetically generated API requests. CompactionBench instead grounds workloads in real code repository evolution and emulates production workload distributions, yielding realistic trajectories that support repeated-compaction studies under controlled runtime configurations.

Coding Harness and Runtime Compaction. LLM-based context compaction is widely implemented in coding harnesses (OpenAI, 2026; Google, 2026; OpenCode, 2026; Pi, 2026; Anthropic, 2026; GitHub, 2026). Most existing implementations perform blocking compaction with harness-specific heuristics. APEC instead provides a principled, drop-in design that preserves information flow while maintaining prefix caching friendliness for lower costs.

Context Compression and Management. Prior work explores masking, pruning, learned compression, and parallel summarization (Lindenbauer et al., 2025; Xiao et al., 2026; Wang et al., 2026; Xu et al., 2026; Kang et al., 2026; Li et al., 2026a; Cim et al., 2026; Nguyen et al., 2026). These techniques primarily improve the compression operator itself, e.g., through retraining, auxiliary models, or heuristics. APEC is orthogonal to these approaches and can incorporate different compression operators, while our evaluation retains the production-style summarization operator.

7 Conclusion

We characterize context compaction in production coding agents, introduce CompactionBench for evaluating compaction in long-horizon workloads, and present APEC, an asynchronous progressive compaction mechanism that moves summary preparation off the critical path. We open-source the production traces and benchmark to facilitate further research on efficient context management.

AI USE STATEMENT

We used AI tools to assist with writing, code implementation, and evaluation. The authors directed the work, reviewed the AI-assisted outputs and experimental results, and take full responsibility for the final paper and artifacts.

ETHICS STATEMENT

Our production analysis uses aggregated operational metrics, including token counts, latency, cache usage, and tool-call statistics. The production telemetry available to us does not contain prompt content; our analysis focuses on system performance rather than the content of user interactions.

REPRODUCIBILITY STATEMENT

The appendices describe CompactionBench construction, the OpenClaude implementation, and the experimental and evaluation protocols. Supporting code and benchmark artifacts are available in our anonymous repository. These materials support reproduction of the benchmark experiments, although hosted-model behavior may vary over time. We will release production telemetry following appropriate redaction and privacy review.

REFERENCES

Anthropic. Automatic context compaction. https://platform.claude.com/cookbook/tool-use-automatic-context-compaction, 2026. Claude Cookbook. Accessed: 2026-09-22.

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

Datacurve. DeepSWE. https://github.com/datacurve-ai/deep-swe, 2026.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Daniel Wang, Derek Chen, Pranav Raja, Albert Liu, Marek Šuppa, Nevasini Sasikumar, Luyang Kong, Erik Quintanilla, Xiangyi Li, Ivan Bercovich, and Steven Dillmann. SWE-Marathon: Can agents autonomously complete ultra-long-horizon software work? *arXiv preprint arXiv:2606.07682*, 2026. URL https://arxiv.org/abs/2606.07682.

GitHub. Managing context in GitHub Copilot CLI. https://docs.github.com/en/copilot/concepts/agents/copilot-cli/context-management, 2026. GitHub Docs. Accessed: 2026-09-22.

Gitlawb. OpenClaude. https://github.com/Gitlawb/openclaude, 2026. Accessed: 2026-09-22.

Google. Gemini cli context compression, 2026. URL https://github.com/google-gemini/gemini-cli/blob/main/packages/core/src/context/chatCompressionService.ts. Accessed: 2026-09-22.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

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. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026. URL https://icml.cc/virtual/2026/poster/66270.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. SWE-EVO: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025. URL https://arxiv.org/abs/2512.18470.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*, 2026a. URL https://arxiv.org/abs/2607.05378.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, Lichao Sun, Haotao Mi, and Leowei Liang. Long-Horizon-Terminal-Bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026b. URL https://arxiv.org/abs/2607.08964.

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. URL https://arxiv.org/abs/2508.21433.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Tuan Nguyen, Qiran Hu, Banruo Liu, Khoa D. Doan, Kok-Seng Wong, and Fan Lai. REVA: Reusable evidence view aggregation for context-efficient rag serving. *arXiv preprint arXiv:2609.11209*, 2026. URL https://arxiv.org/abs/2609.11209. Accepted at IEEE ICDM 2026.

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996. doi: 10.1007/s002360050048.

OpenAI. Codex automatic context compaction, 2026. URL https://github.com/openai/codex/blob/main/codex-rs/core/src/compact.rs. Accessed: 2026-09-22.

OpenCode. Compaction, 2026. URL https://opencode.ai/v2/docs/compaction. Accessed: 2026-09-22.

Pi. Compaction and branch summarization, 2026. URL https://github.com/earendil-works/pi/blob/main/packages/coding-agent/docs/compaction.md. Accessed: 2026-09-22.

Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, Zhepeng Cen, Cheng Qian, Shelby Heinecke, Weiran Yao, Silvio Savarese, Caiming Xiong, and Huan Wang. LoCoBench: A benchmark for long-context large language models in complex software engineering. *arXiv preprint arXiv:2509.09614*, 2025a. URL https://arxiv.org/abs/2509.09614.

Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, Roshan Ram, Akshara Prabhakar, Tulika Awalgaonkar, Zixiang Chen, Zhepeng Cen, Cheng Qian, Shelby Heinecke, Weiran Yao, Silvio Savarese, Caiming Xiong, and Huan Wang. LoCoBench-Agent: An interactive benchmark for LLM agents in long-context software engineering. *arXiv preprint arXiv:2511.13998*, 2025b. URL https://arxiv.org/abs/2511.13998.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. LongCodeBench: Evaluating coding LLMs at 1m context windows. *arXiv preprint arXiv:2505.07897*, 2025. URL https://arxiv.org/abs/2505.07897.

Vlad Sobal, Shuo Yang, Yuting Zhang, Wei Xia, and Stefano Soatto. StaminaBench: Stress-testing coding agents over 100 interaction turns. *arXiv preprint arXiv:2606.19613*, 2026. URL https://arxiv.org/abs/2606.19613.

Terminal-Bench Team. Terminal-Bench 4.0. https://www.tbench.ai/news/terminal-bench-4-0, 2026. Accessed: 2026-09-22.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Pruner: Self-adaptive context pruning for coding agents. *arXiv preprint arXiv:2601.16746*, 2026. URL https://arxiv.org/abs/2601.16746.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3(FSE):1241–1263, 2026. doi: 10.1145/3797084. URL https://arxiv.org/abs/2509.23586.

Buqiang Xu, Zirui Xue, Dianmou Chen, Chenyang Fu, Chiyu Wu, Caiying Huang, Chen Jiang, Jizhan Fang, Xinle Deng, Yijun Chen, Yunzhi Yao, Xuehai Wang, Jin Shang, Gong Yu, and Ningyu Zhang. TokenPilot: Cache-efficient context management for llm agents. *arXiv preprint arXiv:2606.17016*, 2026. URL https://arxiv.org/abs/2606.17016.

Peter Zhong, Pashootan Vaezipoor, Fuyang Cui, Vaibhav Kumar, James Austin, Azin Asgarian, Toby Ho, Paul Inder, Imen Kedir, Zhen Li, Nicholas Ondo, Asna Shafiq, Ibrahim Sheikh, Edouard Sioufi, Setareh Soltanieh, Ben Wilde, Jacky Zhao, Ryan Carelli, Heather Miller, and Michele Catasta. ViBench: A benchmark on vibe coding. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 497–513, 2026. doi: 10.1145/3786335.3813162. URL https://github.com/ViBench/vibench-public.

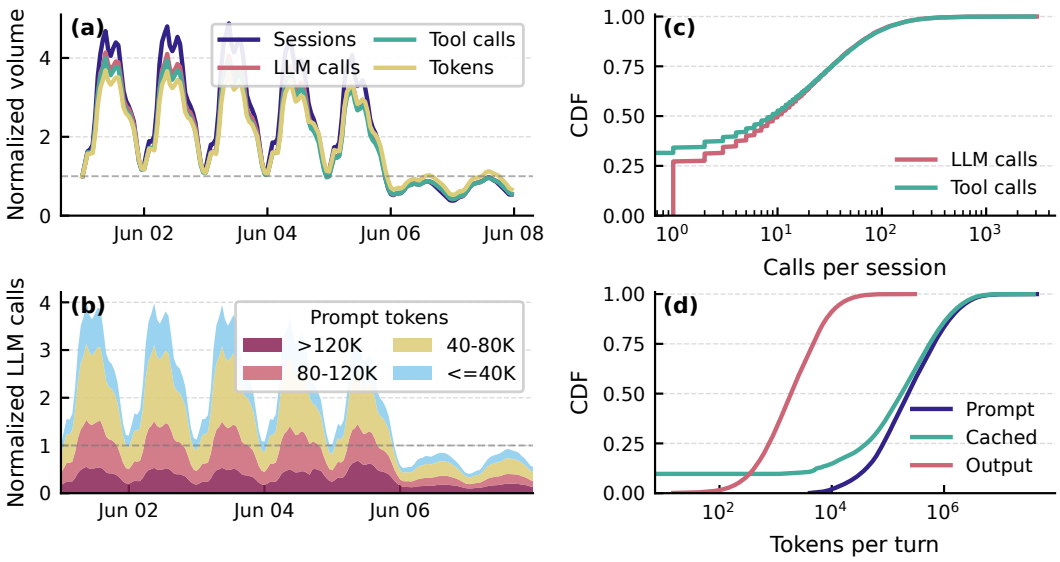


Figure 7: Production workload overview. (a) Hourly normalized volumes. (b) LLM calls by prompt length. (c) Calls per session. (d) Tokens per turn.

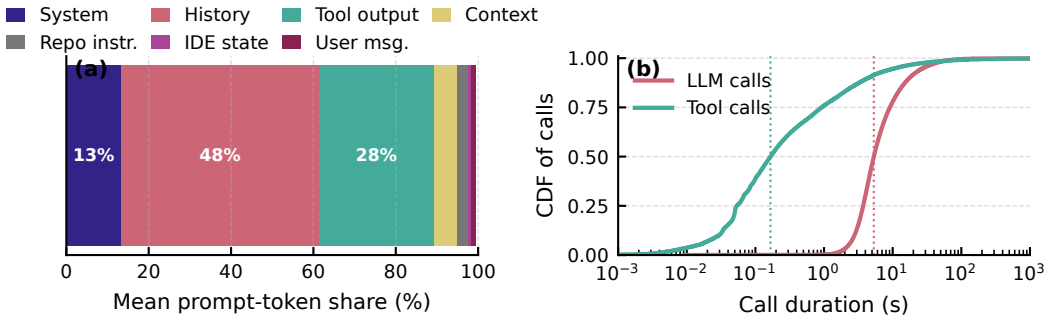


Figure 8: Prompt composition and call durations. (a) History and tool messages contribute 76% of prompt tokens. (b) LLM- and tool-call duration CDFs.

A Production Coding Agents

Figure 7 presents the temporal pattern of production agentic coding workloads and their general session behavior. Aggregate volumes follow the same week-long pattern, peaking during the day time while decreasing at night and during the weekend (panel a). Long and extremely long prompts persist throughout that cycle (panel b). Calls per session and token volume per turn are heavy-tailed (panels c–d), consistent with Table 1b: a median session has 15 LLM calls and lasts 4.2 minutes, but the P90 exceeds 100 calls and 177 minutes.

As shown in Figure 8a, conversation history supplies 48% of mean prompt tokens, and tool-call output adds 28%. Tool execution in coding agents is mostly short (0.17 s median), compared to LLM calls (5.3 s median), as shown in Figure 8b. However, 5% tool calls exceed 11 s and 1% exceed 79 s, which exposes seconds of idle time to leverage.

A Heterogeneous Model and Tool Mix. Production coding agents operate in a mixed setting rather than a single fixed configuration. The production traces span 27 models from multiple model families and more than 40 tools. Model usage is uneven: the three most-used models account for approximately 53% of invocations, while the remaining calls span many other models. Tool use likewise mixes file exploration, search, editing, command execution, builds, and tests. File retrieval accounts for 35.0% of tool invocations, command execution for 17.0%, and string replacement for 9.8%, alongside a long tail of less frequent tools. Aggregate token counts and execution patterns therefore reflect a mixture of models, tool capabilities, and user workflows, rather than one uniform agent workload.

Long-tailed Behavior. This heterogeneity extends beyond which models and tools are used to the amount of work performed in each session. As reported, tool execution has a median duration of only

Table 6: Per-session statistics for sessions with context compactions. "Skew" is mean/median. "↑ over all sessions" is the ratio between median of compaction sessions vs. all sessions.

Metric	Median	P75	P90	Mean	Skew	↑ over all sessions
User turns	8	15	27	12.3	1.5×	2.7×
LLM calls	99	172	277	134.8	1.4×	6.6×
Tool invocations	102	178	286	138.8	1.4×	7.8×
Duration (min)	139.1	323.4	538.3	341.4	2.5×	33.1×

166 ms but a mean of 16.7 s and a P99 of 79 s: frequent lightweight reads coexist with occasional expensive commands and builds. Tool-induced context growth also depends on execution outcomes; failed builds introduce roughly 7–8 times more tokens at the median than successful builds. Together with the heavy-tailed session lengths described above, these differences make production a complex mixture of short interactions and long-running, context-intensive workflows. Matching its complete workload distribution would require reproducing not only the model and tool mix, but also their frequencies, failure patterns, and interactions with user behavior.

How Compaction Changes Session Statistics. Table 6 characterizes the subset of sessions that trigger at least one compaction. These sessions are substantially longer and more active than the typical session: relative to the median across all sessions, compaction sessions have 2.7× more user turns, 6.6–7.8× more LLM calls and tool invocations, and 33.1× longer duration, indicating that compaction is driven primarily by extended, tool-heavy interactions rather than by turn count alone. All four metrics are also right-skewed (mean/median of 1.4–2.5×), with duration the most skewed, so a small number of very long-running sessions pull the mean well above the median.

Interpreting cache-hit changes.. Our cache analysis measures changes across individual compaction boundaries, rather than across entire sessions. The 139-minute median duration of sessions containing compaction therefore does not imply that a cached prefix remains idle for that duration: cache reuse depends on the timing and content of successive requests and the serving system's retention policy. The distribution also captures heterogeneous behavior: 17.4% of paired events show no decrease or an increase in cache-hit rate, which is compatible with the median per-event decrease of 57.8 percentage points. These measurements establish an observed association between compaction boundaries and reduced cache reuse, consistent with prefix rewriting; they do not estimate a no-compaction counterfactual or separately identify effects from cache expiration, eviction, and request routing. We use this production characterization to motivate prefix-preserving compaction, and evaluate the proposed mechanism separately in OpenClaude.

B COMPACTIONBENCH

Source and unit of evaluation. COMPACTIONBENCH instantiates the construction in Section 3.2 using release-evolution tasks from SWE-EVO (Le et al., 2025). Each task comprises a repository at a specified base revision, a target release, its changelog requirements, and an executable test environment. The main evaluation in Table 3 uses eight release transitions across four repositories, containing 315 changelog items (Table 7). A release transition is one evaluated task; its items are successive requests within that task.

Repository	Release transition	Items
Conan	2.0.2 → 2.0.3	22
	2.0.14 → 2.0.15	43
Dask	2022.9.2 → 2022.10.0	25
	2023.3.2 → 2023.4.0	39
	2023.6.0 → 2023.6.1	21
DVC	1.0.0a1 → 1.0.0a2	34
	2.8.1 → 2.8.2	30
Pydantic	v2.6.0b1 → v2.6.0	101
Total	**8 release transitions**	**315**

Table 7: CompactionBench task inventory. Each item introduces one requirement into a persistent coding session; a request may require multiple LLM and tool calls.

Task construction. We split each task's changelog into bullet-level requirements, retaining multi-line descriptions. Requirements with an identifiable pull-request number are ordered by ascending number; remaining requirements follow in their original order. This sequence approximates iterative development, in which engineers introduce, modify, or remove functionality across successive requests. The first request specifies the repository and release goal. Later requests introduce one new requirement and ask the agent to implement the new requirement without regressing existing functionality.

Container and harness setup. We substantially modify the generation environment and integrate OpenClaude with the Pier execution framework through a custom multi-turn agent adapter. Pier manages the task containers and execution lifecycle, while OpenClaude runs the model–tool interaction loop inside each container. Our adapter supplies successive changelog requirements, resumes the same OpenClaude session, and checkpoints the accumulated source changes after each request. The working tree and conversation state therefore persist throughout a release task, with context management performed by the compaction policy under evaluation.

We also observed agents cheating in the original SWE-EVO setup by reading Git history and tag references to retrieve target-release code instead of implementing the requested changes. We therefore rebuild the generation images with repository history and release references stripped, removing this local shortcut. Evaluation excludes agent-generated changes to tests and scores the resulting source patch.

Evaluation. We evaluate the final cumulative source patch using repository tests, we average the accuracy across tasks. Tasks has equal weight in final accuracy as some have a higher number of tests.

B.1 SIMILARITY TO PRODUCTION SESSIONS

As discussed in Section A, production combines many models, tools, and user workflows with long-tailed execution behavior. Reproducing its full joint distribution is inherently difficult for a controlled, reproducible benchmark, and we do not claim that COMPACTIONBENCH matches every production token, tool, or session-length distribution. Instead, our goal is to reproduce the characteristic behavior of production sessions that require compaction: continued user requests accumulate context, compaction makes room for further work, and execution continues with renewed context growth. This focuses the benchmark on the sessions relevant to evaluating compaction, rather than the many short production interactions that never need it.

COMPACTIONBENCH recreates this behavior through successive requirements in the same repository and conversation, allowing code, tool results, and prior decisions to accumulate through ordi-

nary agent execution rather than artificial context padding. The production and synchronous benchmark examples in Figure 3 also exhibit similar growth–compaction cycles: both show three context reductions over approximately 400 calls, with mean prompt growth between compaction boundaries of about 1.75K tokens per call. These examples, together with the repeated compactions measured in Table 3, support the benchmark's ability to reproduce production-like compaction behavior. Differences in individual model or tool output sizes do not negate this behavioral similarity; the relevant correspondence is sustained context accumulation, compaction, and subsequent regrowth.

C APEC IMPLEMENTATION

C.1 INTEGRATION WITH OPENCLAUDE

We implement APEC using a modified OpenClaude v0.15.0, independently of the benchmark runner and container environment. Our implementation on APEC consists of about 600 lines of code, excluding tests.

Compaction hooks. The native integration points are threshold checking and compaction dispatch in `autoCompact.ts`, history reconstruction in `compact.ts`, and the request/turn lifecycle in `query.ts`. Our adapter exposes these integration points through three strategy hooks: a pre-request threshold check, a compaction operation that installs replacement history, and a turn-end hook that settles pending background work before the process exits. The threshold hook can launch a summary without blocking the next foreground request; the compaction hook installs a completed result at a subsequent request boundary.

History replacement. The controller issue asynchronous summarization at turn boundaries, if the token number exceed the soft threshold. Then the summary status is checked later at each turn boundary, and the history is installed when the summary is ready. Given that we only install new snapshots at turn boundary, we don't need to worry about incomplete request or pending tool calls.

C.2 PORTING TO OTHER AGENT HARNESSES

Relationship to Copilot. The Copilot compaction implementation underlying our production traces is closed-source. We expect its high-level summarize-and-reconstruct workflow to resemble that of Codex and other coding harnesses.

The required hooks can be expressed by the following schematic interface:

```
interface CompactionHooks {
  // Before a model call: check thresholds, start/poll summary.
  onShouldAutoCompact(ctx: Context): boolean;

  // Install a ready result, or compact at the hard limit.
  onCompact(ctx: Context): Promise<CompactionResult>;
}
```

Most coding agent harness provide such hook interface: Codex provides analogous integration points in `session/turn.rs` for invoking compaction and `compact.rs` for summary generation and history replacement. OpenCode's `session/runner/llm.ts` checks context before a model call through `session/compaction.ts`, which generates and publishes completed checkpoints. Pi exposes the `session_before_compact` extension hook in `agent-session.ts`, with threshold checks and summary generation in `compaction/compaction.ts`. DeepSeek Harness (DSH) defines a replaceable `CompactionEngine` in `compaction/src/index.ts`; its `compaction-basic/src/index.ts` connects automatic compaction to the `agent/pre-step` hook and handles idle transitions through `agent/status`.

C.3 SUMMARIZATION CHOICE AND COMPARISON DESIGN

APEC studies the runtime organization of compaction: when summarization executes and how compacted history is incorporated into the reusable context prefix. We select baselines to examine these scheduling and history-reconstruction choices, using same-model summarization as the common summarization procedure.

Comparison design. Sync-L versus Async-L examines asynchronous execution with monolithic compaction; Sync-L versus PEC examines progressive reconstruction under blocking execution; and PEC versus APEC examines asynchronous execution with progressive reconstruction. These comparisons use the same low trigger ratio and summarization model, while varying scheduling and the organization of retained history. Together, these comparisons address the proposed runtime mechanisms rather than a ranking of all context-management systems.

Common summarization procedure. We use the acting model to summarize history, following a widely used coding-harness design. Retaining this procedure avoids introducing a change in

summarization-model capability or an auxiliary model into the scheduling and reconstruction comparisons. We evaluate its use through end-to-end task accuracy, cost, and completion time, together with cache reuse and compaction measurements.

Role of the pruning baselines. Prune-H and Prune-L augment synchronous compaction with conservative, rule-based tool-output cleanup: deduplicating file reads and removing formatting from tool outputs, at the high and low trigger ratios, respectively. They assess whether lightweight redundancy removal provides comparable benefits without introducing a learned compression policy.

Relationship to other context-management methods. TokenPilot combines cache-aware ingestion with lifecycle-based eviction (Xu et al., 2026); SWE-Pruner selects code observations conditioned on the agent's goal (Wang et al., 2026); and learned compression and parallel summarization change how compressed representations are produced (Kang et al., 2026; Cim et al., 2026). Replacing the common summarization procedure or adding these policies would introduce additional design choices into the comparisons above. Our baseline selection follows the mechanisms being evaluated.

D Evaluation Setup

D.1 Existing-benchmark characterization

Table 2 characterizes context growth during benchmark execution rather than comparing task-solving accuracy. We retain each workload's interaction structure: front-loaded prompts, autonomous tool use, or benchmark-defined continuation and phase transitions. For strategy, here we blocking automatic compaction (the default compaction from OpenCluade and other harness). The setup is same as Sync-H baseline. We use GPT 5.6 Sol and a compaction trigger of 90%.

For Figure 2, CompactionBench we use the result of Sync-H over the 5 repeats. And for production trace, we use population that triggers at least one compaction.

D.2 Main Evaluation

Model serving and API access. We use Azure-hosted model APIs for both the main evaluation and the ablation studies. GPT-5.6 Sol serves as the main model, while the model-family ablation uses GPT-5.4.

Metrics. We report test accuracy per task, averaged over repeats within each task and then equally across tasks. Token and cost measurements, we use price from official API on GPT models. Context growth counts new retained assistant and tool payload with `o200k_base`. Agent time is calculated by the API time plus harness time and tool execution time. For asnyc compaction, we calculate overlap by calculating the overlap oin time stamp.

We have made significantly efforts to control the noise from different API endpoint by using dedicated, single-tenant API enpdoint with same rate-limit, guardrail, version and region deployed. However we still notice some noise in terms of API measured time.

Repeated compaction by task. For each method, we average the compaction count over each of the eight tasks used in Table 3. Table 8 counts distinct tasks whose mean strictly exceeds each threshold. Each task contributes at most once per column, regardless of its repeat count.

Method	>1 compaction	>2 compactions	>3 compactions
Sync-H	3	1	0
Sync-L	8	6	4
Async-L	7	5	4
Prune-H	2	1	0
Prune-L	8	5	3
PEC	8	4	4
APEC	8	5	4

Table 8: Number of distinct tasks (out of eight) whose mean compaction count across strictly exceeds 1, 2, or 3.

Compaction coverage and future workloads. Compaction frequency depends jointly on the workload, model context window, trigger threshold, and agent trajectory; it is not a configuration-independent property of a task. In Table 8, three tasks average more than one compaction under Sync-H, compared with all eight under Sync-L. Nevertheless, demanding tasks compact repeatedly even under Sync-H, and individual Sync-L trials reach nine compactions. The benchmark supplies realistic, executable workloads for studying these cycles and their sensitivity to runtime policies, rather than guaranteeing a fixed event count under every configuration.

This coverage contrasts with Table 2: under the recorded configurations, every surveyed benchmark except LongSWE-Bench shows no compaction, while LongSWE-Bench triggers only an initial, front-loaded event rather than repeated growth–compaction cycles. Yet compaction is already consequential in real use: the 7.8% of production sessions that require it account for 44.2% of total token consumption. Recent work on optimized compression (Kang et al., 2026), cache-aware context management (Xu et al., 2026), parallel compaction (Cim et al., 2026), and compaction-aware training (Li et al., 2026a) further demonstrates growing attention to this problem. Despite this activity, the surveyed datasets do not provide comprehensive end-to-end coverage of repeated compaction in frontier coding agents.

To our knowledge, COMPACTIONBENCH is the first benchmark specifically designed to fill this evaluation gap. It is an initial step, not a fixed workload that will remain equally challenging indefinitely. As model capabilities and context windows grow, maintaining meaningful compaction coverage may require updating the benchmark with longer, heavier tasks. We therefore encourage broader long-horizon datasets spanning more repositories, evolving requirements, and sustained user–agent interactions.

Hard compaction and epoch rollover. The append-only invariant in APEC applies within an epoch, not to an unbounded session history. Once accumulated summaries reach the prefix cap, the runtime stops preparing incremental summaries; at the hard context threshold, it performs a full synchronous compaction and starts a new epoch. This transition is an explicit part of the design rather than a failure of the append-only invariant. No epoch rollover was observed in the reported main-evaluation APEC trials.

D.3 MODEL FAMILY

We use the same setup for GPT5.6 Sol except change the model to GPT5.4. Total cost in Table 4 adds foreground and applied automatic-summary usage at the reference price of \$5/\$0.50/\$25 per million uncached-input/cached-input/output tokens. Note that the exact price are subject to change, but the input/output difference, and 90% cache discount on cached read is generally true.

D.4 SUMMARY AND HISTORY BUDGETS

We vary the summary-output cap S_{cap} and raw-tail budget H; S denotes actual summary output. Context and trigger settings are fixed.

E COMPACTION COST MODEL

Scope and interpretation. This model provides an analytical estimate rather than an exact prediction of execution cost. It assumes fixed foreground work, constant context growth and summary sizes, filled retention budgets, reusable cached prefixes, call-aligned triggers, and a single append-only epoch. Real executions may violate these assumptions through variable tool outputs, cache eviction, retries, or epoch rollover, and also compaction implementation. The compaction may also change the agent's subsequent behavior. Nevertheless, the derivation makes the tradeoff between shorter prompts, cache refills, and summarization overhead explicit, allowing us to estimate potential savings under stated conditions.

E.1 CLOSED FORMS FOR THE MAIN-TEXT MODEL

This appendix derives the analytical cost expressions used in Section 4. Consider N foreground calls starting with a fixed prefix of b tokens. Each foreground call emits and appends a constant $g > 0$ tokens, and each summary emits and retains S tokens. A compaction retains H raw-history tokens in addition to the new summary, giving a reconstructed-history budget $B = S + H$ that excludes the fixed prefix and, for APEC, earlier immutable summaries. The summary size S is distinct from its configured output cap S_{cap}.

Synchronous compaction triggers at prompt length $T > b + S + H$. APEC snapshots $L_{\text{snap}} > S$ raw-history tokens and applies each summary after ℓ foreground calls. Let π_{pf}, π_{hit}, and π_{dec} denote prices per uncached input, cached input, and output token, respectively.

Common foreground cost. Before call $i \in \{1, \dots, N\}$, an uncompacted prompt contains $b + (i-1)g$ tokens. Summing over calls gives $Nb + gN(N-1)/2$ input tokens, of which $b + (N-1)g$ are uncached. The N calls generate Ng billed output tokens. These three quantities give $\mathcal{C}_0$ in the main text.

$$\mathcal{C}_0 = \pi_{\text{hit}} \left[Nb + \frac{gN(N-1)}{2} \right] + (\pi_{\text{pf}} - \pi_{\text{hit}})[b + g(N-1)] + \pi_{\text{dec}} Ng.$$

Synchronous compaction. The first rewrite occurs after $(T-b)/g$ foreground calls; subsequent rewrites occur every $(T-b-S-H)/g$ calls. Therefore the number before the last foreground call is

$$m_{\text{s}} = \max\left\{0, \left\lfloor \frac{g(N-1) - S - H}{T - b - S - H} \right\rfloor \right\}.$$

Each rewrite removes $T - b - S - H$ tokens. The number of subsequent calls benefiting from each removal decreases by $(T-b-S-H)/g$ per rewrite. Summing this arithmetic progression gives the cumulative input reduction

$$(T - b - S - H) m_{\text{s}} \left[N - \frac{T-b}{g} - \frac{(m_{\text{s}} - 1)(T - b - S - H)}{2g} \right].$$

After a rewrite, $S + H$ foreground input tokens are uncached rather than the usual g. The summary request reads $T - g$ cached and g uncached tokens and generates S output tokens. Their combined cost increment is

$$(\pi_{\text{pf}} - \pi_{\text{hit}})(S + H - g) + \pi_{\text{hit}}(T - g) + \pi_{\text{pf}} g + \pi_{\text{dec}} S = \pi_{\text{hit}} T + (\pi_{\text{pf}} - \pi_{\text{hit}})(S + H) + \pi_{\text{dec}} S.$$

Subtracting the cached-price input reduction from $\mathcal{C}_0$ and adding this increment per rewrite gives

$$\begin{aligned} \widehat{\mathcal{C}}_{\text{sync}} = \mathcal{C}_0 &- \pi_{\text{hit}}(T - b - S - H) m_{\text{s}} \left[N - \frac{T-b}{g} - \frac{(m_{\text{s}} - 1)(T - b - S - H)}{2g} \right] \\ &+ m_{\text{s}} [\pi_{\text{hit}} T + (\pi_{\text{pf}} - \pi_{\text{hit}})(S + H) + \pi_{\text{dec}} S]. \end{aligned} \tag{2}$$

Asynchronous compaction. The first snapshot starts after $(L_{\text{snap}} + H)/g$ foreground calls; later snapshots start every L_{snap}/g calls. A preparation lag of t calls leaves

$$m_{\text{a}}(t) = \max\left\{0, \left\lfloor \frac{g(N - 1 - t) - H}{L_{\text{snap}}} \right\rfloor \right\}$$

summaries available by the final call. Consequently $m_{\text{a}}(0)$ requests are started and $m_{\text{a}}(\ell)$ summaries are applied. Each application removes $L_{\text{snap}} - S$ tokens, beginning ℓ calls after its snapshot, for a cumulative input reduction of

$$(L_{\text{snap}} - S) m_{\text{a}}(\ell) \left[N - \ell - \frac{L_{\text{snap}} + H}{g} - \frac{(m_{\text{a}}(\ell) - 1) L_{\text{snap}}}{2g} \right].$$

An application makes $S + H + \ell g$ tokens uncached on the next foreground request, instead of the usual g. This adds $(\pi_{\mathrm{pf}} - \pi_{\mathrm{hit}})[S + H + (\ell - 1)g]$ per application. The kth summary request reads the cached fixed prefix, $k - 1$ previous summaries, and its L_{snap}-token snapshot, then generates S tokens. Summing these request costs gives

$$m_{\mathrm{a}}(0)\left[\pi_{\mathrm{hit}}\left(b + L_{\mathrm{snap}} + \frac{(m_{\mathrm{a}}(0) - 1)S}{2}\right) + \pi_{\mathrm{dec}}S\right].$$

Together these terms give

$$\begin{aligned}\widehat{\mathcal{C}}_{\mathrm{apec}} = \mathcal{C}_0 &- \pi_{\mathrm{hit}}(L_{\mathrm{snap}} - S)m_{\mathrm{a}}(\ell)\left[N - \ell - \frac{L_{\mathrm{snap}} + H}{g} - \frac{(m_{\mathrm{a}}(\ell) - 1)L_{\mathrm{snap}}}{2g}\right] \\ &+ (\pi_{\mathrm{pf}} - \pi_{\mathrm{hit}})m_{\mathrm{a}}(\ell)[S + H + (\ell - 1)g] \\ &+ m_{\mathrm{a}}(0)\left[\pi_{\mathrm{hit}}\left(b + L_{\mathrm{snap}} + \frac{(m_{\mathrm{a}}(0) - 1)S}{2}\right) + \pi_{\mathrm{dec}}S\right]. \qquad (3)\end{aligned}$$

Charging starts while crediting only applications also covers a final summary that finishes after the foreground workload ends. Subtracting Equation (3) from Equation (2) cancels $\mathcal{C}_0$. Substituting $T - b = D + B$ and collecting terms by π_{hit}, $\pi_{\mathrm{pf}} - \pi_{\mathrm{hit}}$, and π_{dec} yields Equation (1).