CADOC: Cache-Aware Dynamic Object Context for Long-Horizon Agents

Anonymous authors
Paper under double-blind review

Abstract

For a long-horizon agent, context is the bottleneck: the history is resent with every request, the window caps task length, and reasoning degrades as the history grows. Replacing structured objects with compact retrieval Cards shortens the prompt and keeps the exact originals retrievable, but editing the history can break prefix-cache reuse, and prior recoverable methods time their edits by forecasts of future reuse or by preset intervals. We propose CADOC (Cache-Aware Dynamic Object Context), an online algorithm that replaces structured objects with compact Cards while preserving exact, on-demand retrieval of their original contents. CADOC schedules replacements in batches by balancing accumulated waiting cost against shared cache-reconstruction cost. Its scheduling rule follows from an economic order quantity trade-off, recovers the optimal integer batch under stationary assumptions. Across evaluation, CADOC consistently achieves the lowest aggregate input cost among the compared configurations, which reduces input cost by approximately 40% on average while maintaining task performance close to full context. CADOC thus provides a cost-derived approach to compressible context management, demonstrating that efficient compression depends not only on shortening prompts but also on scheduling edits to preserve cache reuse.

1 Introduction

Context is the bottleneck for long-horizon agents. Every request resends the history, so retained tokens incur costs again, even on a cache hit. The context window limits task length, and models struggle to use relevant information in long histories (Kang et al., 2026; Sun et al., 2026). Managing context therefore affects both cost and performance.

Compression is one remedy, and much of what accumulates invites it: code, file contents, logs, and tool outputs matter when produced but seldom need to stay in view. Lossy methods, whether token pruning, learned agent compression, or subtrajectory folding (Jiang et al., 2023; Kang et al., 2026; Sun et al., 2026), discard details a later step may need; a structured object can instead be moved to external storage (Packer et al., 2023) and replaced by a compact retrieval Card that keeps the exact original retrievable (Figure 1b).

A recoverable method must still decide when to commit; the two closest systems account for the prefix cache. Self-GC commits the edits a planner LLM proposes once their expected saving over future requests outweighs the cache break, so it must estimate future reuse (Hao et al., 2026). TokenPilot runs an LLM estimator every fixed, empirically chosen number of turns and evicts segments it judges expired (Xu et al., 2026). Neither sets the timing from what is already observable. Editing an earlier object can invalidate the cached suffix after it, including the raw hot tail of recent blocks, so separate commits repeat the same reconstruction while a batch shares it; and waiting keeps payloads in every intervening request. On an example from Hermes Compression Evaluation (HCE) histories (Nous Research, 2026), we find that immediate replacement removes 55.93% of cumulative input tokens yet raises input cost by 3.67% (Figure 1a).

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

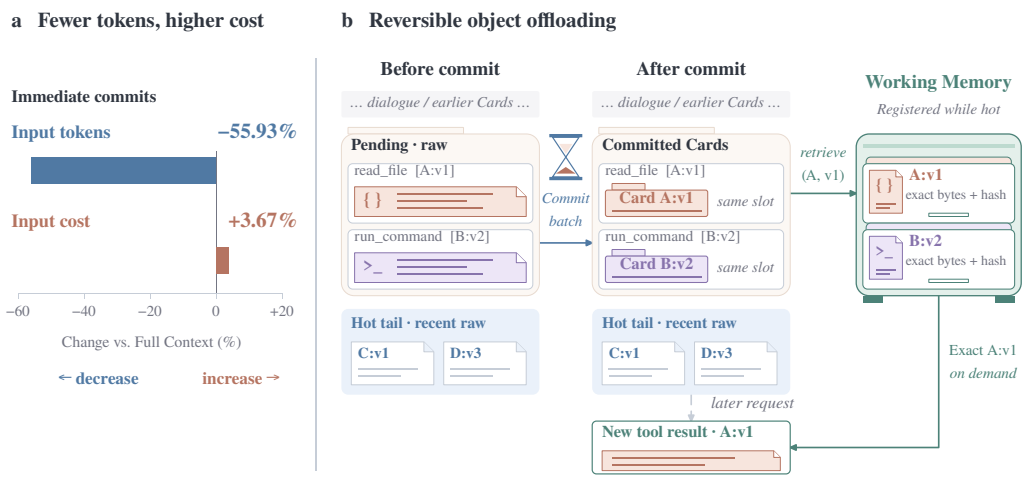


Figure 1: (a) Immediate cuts cumulative input tokens by 55.93% but raises input cost by 3.67% versus Full Context. (b) Pending objects become Cards in place; the hot tail stays visible, and Working Memory returns exact contents.

CADOC, Cache-Aware Dynamic Object Context, is a fully online, LLM-free scheduler that determines when to commit object replacements in batches. It requires no future information or workload forecasts, no additional LLM calls, and no per-setting batch-size tuning: every scheduling decision uses only current shortening and cache estimates. Rather than fixing a batching interval in advance, CADOC dynamically balances the cache-reconstruction cost that a batch can share against the cost accumulated by delaying compression, and commits when further waiting no longer pays.

Our experiments demonstrate two complementary advantages of CADOC. First, CADOC reduces agent input costs by approximately 40% while maintaining comparable task performance. We evaluate the complete system through continuing task chains on several benchmarks, retaining conversation history across tasks to capture the costs of long-horizon execution. Relative to a full-context baseline, CADOC reduces input costs substantially. Second, CADOC consistently achieves the lowest cost among the compared commit policies across diverse workloads and hyperparameter settings. We evaluate commit scheduling through fixed-history replays, comparing CADOC with immediate replacement, fixed-size batching, and token-threshold policies. CADOC attains the lowest cost at all settings. Whereas the best fixed batch size changes across settings, CADOC adapts its commit timing automatically, obtaining these results without future information, additional LLM calls for scheduling, or per-setting batch-size tuning.

2 Objects, Cards, and the Hot Tail

CADOC shortens the visible context by reversible object externalization, not lossy compression: Working Memory keeps the original bytes, and a Card stands in for the object in the prompt. Registration and replacement preserve the object's original position in the interaction history; retrieval returns its immutable version as a new tool-result occurrence.

Objects and immutable versions. Runtime metadata and parsing rules identify structured spans such as file contents, code, tool results, logs, tables, and generated artifacts. Reliably identified objects whose replacement reduces tokens are registered before replacement; ordinary dialogue stays visible. Each stored object has an immutable version, so a historical Card recovers the contents observed at that point even if the file is modified later.

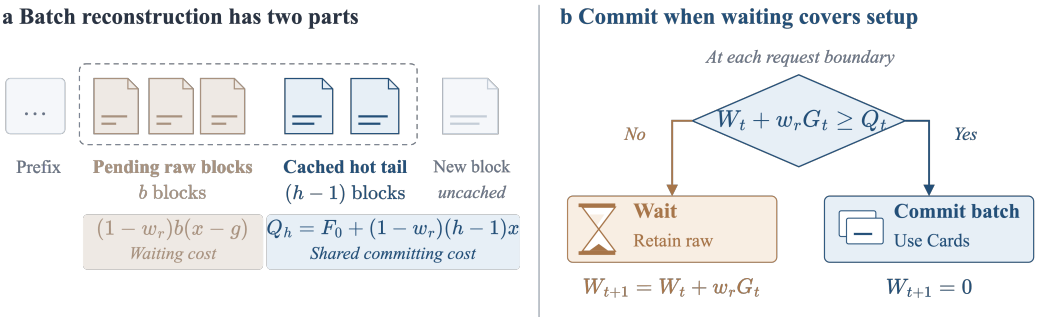


Figure 2: The cached interval and the batch-commit rule. Left: the b pending blocks shorten to $b(x-g)$ tokens, while the $h-1$ cached hot blocks stay raw. Right: waiting-loss updates and the commit decision.

Cards and exact retrieval. A Card records the object's identity, version, source, and structural metadata, with no LLM-generated synopsis. The Card below replaced a terminal search result in a recorded task; ellipses mark omitted metadata:

```
<OBJECT_CARD>
{"contains": {..., "top_level_keys":
   ["output", "exit_code", "error"]}, ...,
   "object_ref": "object://obj_0722dfa749e04b58a0094af4@v1",
   "origin": {..., "tool": "terminal"}, ...,
   "type": "structured_data", "version": 1}
</OBJECT_CARD>
```

The model calls retrieve_object, an ordinary function tool taking object_ref and reason; Section C shows the recorded call, recovery check, and subsequent action.

Hot tail and pending sequence. An inference block is a model response together with its tool results. CADOC keeps a recent raw suffix bounded by a maximum block count h and a token budget T_{hot}. Once an object leaves this protected suffix, it joins the pending sequence and stays raw until a batch commit replaces its prepared span with a Card.

3 Cache-Aware Batch Scheduling

Batching lets several object replacements share the cost of reconstructing the cached hot tail, but waiting to form a batch keeps removable payloads in every intervening input. We derive the batch size that balances these costs in a stationary stream, then restate its optimality condition in quantities available during execution; the result is the online rule CADOC uses. Figure 2 summarizes the affected interval and the decision.

3.1 Replacement Changes Both Prompt Length and Cache Reuse

Normalize the cost of an uncached input token to one, and let the cache-read weight $w_r \in [0,1]$ be the relative cost of a cached token. Total input cost is

$$\mathcal{C} = \sum_t (N_t + w_r R_t), \tag{1}$$

where N_t and R_t count the uncached and cached input tokens of request t. The main experiments use $w_r = 0.1$.

Consider replacements inside an otherwise reusable prefix. Let I be the number of cached tokens from the first edit to the cache frontier, and G the number of tokens removed. Keeping the interval costs $w_r I$ on the next request; replacing it requires fresh processing of $I - G$

tokens. The first-request cost difference is therefore

$$\begin{aligned}\Delta\mathcal{C}_{\text{first}} &= (I-G) - w_r I + F \\ &= \underbrace{(1-w_r)(I-G)+F}_{\text{reconstruction premium}} - \underbrace{w_r G}_{\text{shortening benefit}} .\end{aligned} \tag{2}$$

Here F is an optional auxiliary commit cost in the same units. The input-token accounting of Equation (1) has no auxiliary costs, so $F = 0$ there; Card tokens are already part of G and are not charged again through F.

3.2 Stationary Model and the EOQ Correspondence

Consider one new block per request cycle, of fixed length x and containing one replaceable object whose replacement shortens the block by $g \in (0, x)$ tokens. The token budget is inactive here, so the hot tail holds exactly $h \geq 1$ blocks, and eligible objects have completed their initial raw exposure. Each commit replaces all b pending blocks. We assume stable prefix caching, no retrieval, and an indefinitely continuing stream; in this block model the editable span starts at the beginning of each block.

Decisions are made after a block completes and before the next request. At that point the newest block has not yet appeared in any input, while the preceding $h-1$ hot blocks are cached, so a commit affects a cached interval of length

$$I_b = \underbrace{bx}_{\text{pending blocks}} + \underbrace{(h-1)x}_{\text{cached hot tail}} . \tag{3}$$

The newest block needs fresh processing under either decision, so its cost cancels from the comparison.

To compare batch sizes on a common basis, we use an idealized reference that replaces every object as soon as it is eligible and gets a cache hit on the new Card representation without reconstruction. The reference is an accounting device: relative to it, a real schedule pays for delayed shortening and for reconstruction, and all other input costs are common.

An object retained for one extra cached request costs $r = w_r g$. The b objects of a batch wait $b-1, b-2, \ldots, 0$ requests after becoming eligible, so their total waiting loss is

$$W_b = r\sum_{k=0}^{b-1} k = \frac{rb(b-1)}{2}. \tag{4}$$

At the commit, schedule and reference both use Cards, so the only extra cost is the reconstruction premium of Equation (2), namely $(1-w_r)(I_b - bg) + F(b)$. Write $F(b) = F_0 + bf$, with shared overhead $F_0 \geq 0$ and per-object overhead $f \geq 0$. The average excess cost per object is

$$\begin{aligned}\ell_h(b) &= \frac{W_b + (1-w_r)(I_b - bg) + F_0 + bf}{b} \\ &= \underbrace{f + (1-w_r)(x-g)}_{K} + \frac{\overbrace{F_0 + (1-w_r)(h-1)x}^{Q_h}}{b} + \frac{r(b-1)}{2}.\end{aligned} \tag{5}$$

Thus,

$$\ell_h(b) = K + \frac{Q_h}{b} + \frac{r(b-1)}{2}, \qquad r = w_r g. \tag{6}$$

The pending blocks contribute a constant K per object. Batching lowers the per-object share of the shared cost, Q_h/b, but raises the per-object waiting loss $r(b-1)/2$: the EOQ trade-off between setup and holding costs (Harris, 1913; Erlenkotter, 1990).

Proposition 1 (Stationary optimal batch). For $Q_h \geq 0$ and $r > 0$, the continuous minimizer over $b \geq 1$ is

$$b^*_{\text{cont}} = \max\left\{1, \sqrt{\frac{2Q_h}{r}}\right\}. \tag{7}$$

A positive integer b minimizes ℓ_h if and only if

$$\frac{rb(b-1)}{2} \le Q_h \le \frac{rb(b+1)}{2}. \tag{8}$$

The square-root expression follows from $\ell_h'(b) = -Q_h/b^2 + r/2$. For integer batches the relevant comparison is

$$\ell_h(b+1) - \ell_h(b) = \frac{r}{2} - \frac{Q_h}{b(b+1)}. \tag{9}$$

These differences are nondecreasing in b, so a batch is optimal exactly when enlarging it no longer lowers the cost and the previous enlargement did not raise it; these two conditions are Equation (8). Section A gives the full accounting, proofs, and boundary cases. Larger shared reconstruction costs favor larger batches; larger per-request savings favor earlier commits.

3.3 From the Stationary Optimum to an Online Rule

In the stationary model, a pending batch of size b offers shortening $G_b = bg$, and its waiting loss satisfies

$$W_b + w_r G_b = \frac{rb(b-1)}{2} + rb = \frac{rb(b+1)}{2} = W_{b+1}. \tag{10}$$

Hence Equation (8) reads $W_b \le Q_h \le W_b + w_r G_b$. Start from an empty pending sequence and take the first b at which $W_b + w_r G_b \ge Q_h$. If $b > 1$, the previous decision did not cross, so $W_b < Q_h$; if $b = 1$, then $W_1 = 0 \le Q_h$. The first crossing therefore selects an optimal integer batch, the smaller one when adjacent sizes tie.

For a changing history, CADOC evaluates the same crossing with current quantities. At boundary t, let $\mathcal{P}_t$ be the set of pending object occurrences. Their potential shortening is

$$G_t = L_t^{\text{keep}} - L_t^{\text{commit}}, \tag{11}$$

where the two lengths describe the next prompt with all pending contents retained or replaced; under additive token accounting this is the sum of raw-to-Card reductions. Let W_t be the waiting loss accumulated since the current pending contents became eligible; newly eligible contents enter with zero loss, and carrying the pending contents through one more cached request adds

$$\Delta W_t = w_r G_t. \tag{12}$$

Let H_t^{shared} count the already cached tokens after the newest pending block and before the reusable cache frontier. These tokens survive the replacements but must be reconstructed with them, so the estimated shared reconstruction cost is

$$Q_t = F_{0,t} + (1 - w_r) H_t^{\text{shared}}. \tag{13}$$

This reduces to Q_h in the stationary model. The recorded replay uses $F_{0,t} = 0$; Section A.3 details how this suffix is separated from the pending region.

For nonempty $\mathcal{P}_t$, the economic crossing is

$$\boxed{\text{commit all pending replacements if } W_t + w_r G_t \ge Q_t.} \tag{14}$$

Otherwise the controller waits and adds $w_r G_t$ to the ledger after each request that carries the pending contents raw; committed entries are cleared once a successful model response confirms the commit. Hard capacity limits can force an earlier commit. Since the earliest pending edit fixes where reconstruction begins, committing the later pending replacements in the same request shortens the interval at no additional cache break, which is why CADOC commits the whole pending sequence at once.

The evaluated runtime estimates G_t from rough keep and commit prompt lengths, Q_t from prompt and cache estimates, and W_t from a persistent per-delta waiting ledger; the recorded decisions in Section E use the same symbols for these estimates.

Under the stationary assumptions, the crossing recovers the optimum exactly; in changing histories it adapts commit timing to the observed shortening and cache estimates. For the associated waiting–setup objective with fixed eligibility, Section A.4 also proves a finite-horizon competitive bound.

4 Fixed-History Evaluation of Commit Scheduling

4.1 Protocol and Compared Policies

HCE replay (Nous Research, 2026) compares commit policies on identical source histories. Model outputs, object versions, and Card contents are frozen, so each configuration yields a prompt stream whose lengths and reusable prefixes determine its cost. We evaluated at $h \in \{2, 4, 8, 16\}$. All costs follow Equation (1) with $w_r = 0.1$; Section F examines other cache-read weights.

We compare Full Context, Immediate, fixed batches $b \in \{2, 4, 8, 16\}$, five token thresholds from 4,096 to 65,536, and CADOC. Full Context retains everything, Immediate commits at eligibility, and the remaining baselines wait for a block count, or a pending raw-token threshold. CADOC uses the object lifecycle of Section 2, rough prompt and cache estimates, and a persistent waiting ledger. Section B details the replay paths and metric definitions.

4.2 Aggregate Cost and Sensitivity to Batch Size

CADOC achieves the lowest aggregate input cost among all evaluated configurations at every tested hot-tail setting, without per-setting batch-size tuning (Figure 3). Relative to Full Context, it saves 30.65%, 20.61%, 15.14%, and 11.56% at $h = 2, 4, 8, 16$, respectively.

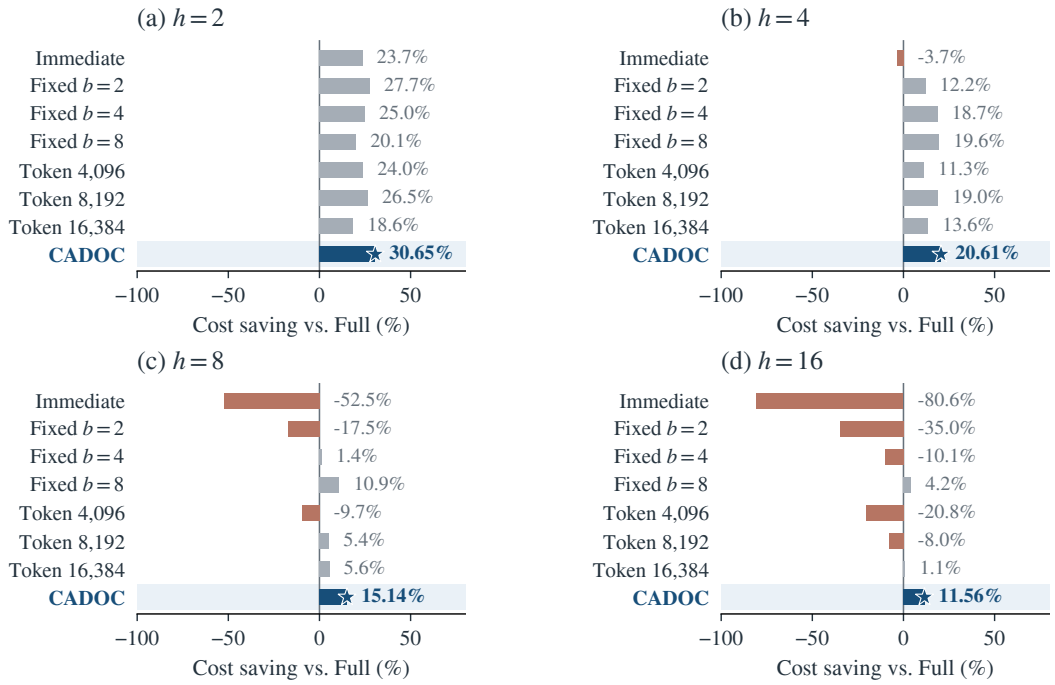


Figure 3: Aggregate input-cost savings relative to Full Context over HCE. Negative values indicate costs above Full Context.

By contrast, a fixed batch size that works well at one hot-tail setting can lose its benefit or even increase cost at another (Figure 4a). The best tested fixed batch shifts from $b = 2$ at $h = 2$ to $b = 8$ at $h = 4, 8, 16$. Keeping $b = 2$ increases cost relative to Full Context by approximately 17.5% at $h = 8$ and 35.01% at $h = 16$, while $b = 4$ increases cost by approximately 10.1% at $h = 16$. CADOC avoids these reversals without manual batch-size adjustment and remains cheaper than the best tested fixed-batch configuration selected separately for each setting.

CADOC's recorded batch sizes make this adaptation explicit (Figure 4b). As h increases, its mean batch size rises from 1.50 to 3.25, 4.75, and 5.00 blocks, while its total commit count

falls from 20 to 3. This trend agrees with the stationary prediction in Equation (7): a larger hot tail increases the shared cache-reconstruction cost, favoring larger batches over which to amortize that cost. CADOC automatically makes this adjustment through the same online rule, rather than requiring a manually chosen batch size for each setting. Aggregate continuation-quality results across the four hot-tail settings appear in Section D.

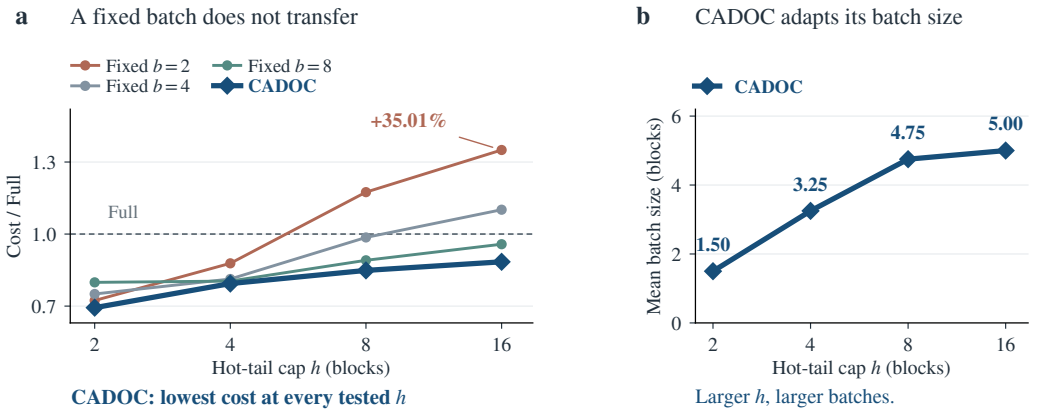


Figure 4: Commit costs and batch sizes across hot-tail settings. (a) Aggregate input costs of fixed-batch configurations and CADOC, normalized by Full Context. (b) CADOC's mean number of blocks per committed batch at $h = 2, 4, 8, 16$.

5 System-Level Evaluation on Benchmark Chains

To evaluate the complete system, we run CADOC and a full-context control arm, FULL, through the same fixed task lists with history retained across successive tasks; we call each such run a benchmark chain. Each arm develops its own model and tool trajectory, so total input cost reflects both the changed context representation and any change in model requests or retrieval. Costs cover every evaluated input request, including retrieved payloads and their later replacement by retrieval receipts, under Equation (1) with $w_r = 0.1$. Section B gives the configuration and execution scope.

Table 1: Complete benchmark-chain results. Input denotes cumulative input tokens, Cost uses $N + 0.1R$, and both reductions are relative to FULL.

Benchmark	FULL solved	CADOC solved	Input ↓	Cost ↓
Terminal-Bench 2.1 (Merrill et al., 2026)	38/89	36/89	49.40%	47.13%
LongMemEval-V2 (Wu et al., 2026a)	10/18	10/18	80.60%	42.40%
SWE-bench (Jimenez et al., 2024)	31/50	29/50	44.72%	34.12%

5.1 Task Performance and Long-Horizon Cost Savings

CADOC substantially reduces input cost while maintaining task performance close to FULL (Table 1). Task-completion rates decrease by at most four percentage points, whereas input cost falls by 34.12–47.13% and cumulative input tokens by 44.72–80.60%. Thus, CADOC achieves substantial savings with only small observed differences in task performance.

The cumulative-cost curves in Figure 5 further show that CADOC's input cost grows more slowly overall than FULL's. As execution continues and conversation histories lengthen, the gap between the two curves becomes increasingly pronounced. Replacing historical objects with compact Cards avoids repeatedly carrying their full contents into subsequent requests, allowing savings from earlier replacements to accumulate. This growing separation highlights CADOC's advantage in long-horizon execution, where historical content would otherwise be processed repeatedly over many requests.

These savings remain substantial even when the agent actively retrieves externalized objects. On LongMemEval-V2, CADOC preserves every task-level outcome while recording 40 retrieval events over 146 model requests and reducing input cost by 42.40%. The reported costs include requests carrying retrieved contents, so the savings account for the input overhead of restoring information when needed. The recorded case in Sections 2 and C further verifies byte-exact recovery and successful downstream use, showing that replacing objects with Cards preserves access to task-relevant content rather than discarding it.

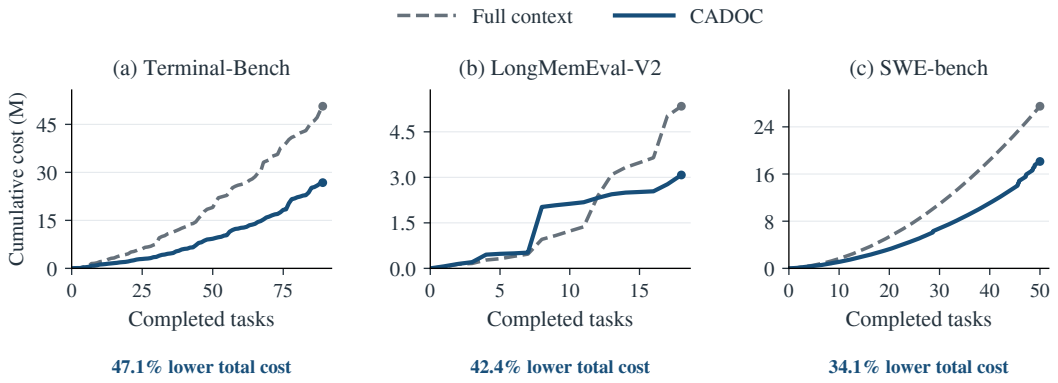


Figure 5: Cumulative input cost during the three benchmark chains. The horizontal axis counts completed tasks. Gray dashed curves denote FULL and blue curves denote CADOC.

5.2 Replay on Benchmark Histories

To compare compression policies on identical agent trajectories, we use the retained conversation histories exported from the FULL runs on Terminal-Bench, SWE-bench, and LongMemEval-V2. We then reconstruct the successive inference boundaries and replay each history in chronological order under different compression configurations, rather than compressing only the final context snapshot. During replay, each policy determines when to replace eligible objects; the resulting prompt lengths and reusable cache prefixes determine its input cost at each request. Context and cache state persist across task boundaries. This protocol compares the costs of different compression policies while holding the recorded agent behavior fixed.

CADOC achieves the lowest aggregate input cost among all evaluated configurations on all three benchmark histories (Figure 6). This consistent advantage contrasts with the workload-dependent effectiveness of other strategies. We further evaluate sensitivity to the cache-read weight w_r (Section F). Across all tested weights on all three benchmark histories, CADOC attains the lowest input cost among the compared policies. These results show that its cost advantage persists across both different workloads and different relative cache-read costs, rather than depending on a single pricing configuration. Section B.1 provides the replay protocol.

6 Related Work

Agent context management spans prompt compression and observation masking (Jiang et al., 2023; Lindenbauer et al., 2025), history consolidation, and external memory. ReSum and MEM1 maintain summaries or compact memory states (Wu et al., 2025; Zhou et al., 2026), ACON refines compression guidelines through failure analysis (Kang et al., 2026), and Context-Folding and AgentFold condense agent trajectories (Sun et al., 2026; Ye et al., 2026). ContextBudget explicitly learns when and how much to compress from the remaining context budget and incoming observation size (Wu et al., 2026b). Complementary systems such as MemGPT, Pichay, and LCM move information outside the active prompt while preserving access through memory tiers, demand paging, or summary hierarchies (Packer et al., 2023;

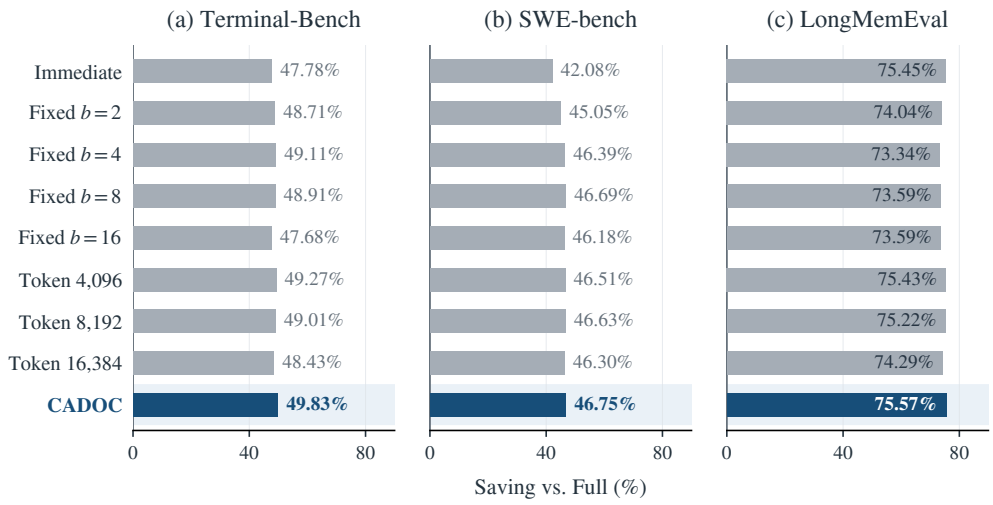


Figure 6: Input-cost savings from offline replay of the conversation histories retained by FULL on three benchmarks.

Mason, 2026; Ehrlich & Blackman, 2026). Beyond reducing prompt length, cache-aware approaches account for the cost implications of prefix reuse (Lumer et al., 2026). Self-GC uses a side-channel semantic planner for recoverable context edits and weighs expected future savings against cache disruption and planning overhead (Hao et al., 2026). TokenPilot combines deterministic compaction and artifact recovery with model-based lifecycle estimation at an empirically selected check interval (Xu et al., 2026). CADOC instead operates on deterministic structured-object replacements, keeping surrounding dialogue visible. Its central distinction is cost-derived commit scheduling: replacement eligibility is separated from commit timing, and batches are triggered by accumulated waiting cost, current token shortening, and shared cache-reconstruction cost. This enables online adaptation without future-reuse forecasts and LLM calls.

7 Conclusion

We proposed CADOC, an online algorithm for recoverable, cache-aware context management that addresses the cost of disrupting prefix caches during compression. CADOC replaces structured objects with retrievable Cards and balances accumulated waiting loss against shared reconstruction cost to determine commit timing, without future information, LLM calls for Card construction or scheduling, or per-setting batch-size tuning. Its rule recovers the optimal integer batch under stationary assumptions. Experiments show the lowest aggregate input cost among evaluated policies across benchmark-history replays. On three continuing benchmark chains, CADOC reduces input cost by 34.12–47.13% while maintaining task performance close to FULL. We thus provide a cost-derived approach to long-horizon context management, demonstrating that effective compression depends on both what is replaced and when replacements are committed.

AI Use Statement

Generative AI tools (ChatGPT with gpt-5.6-sol) assisted with drafting the manuscript and improving the writing.

Reproducibility Statement

The supplementary material contains complete derivations, experimental configurations, aggregate evaluation results, a recorded retrieval case, and runtime diagnostics. Configuration details appear in Section B, and proofs appear in Section A.

References

Daniel R. Dooly, Sally A. Goldman, and Stephen D. Scott. On-line analysis of the TCP acknowledgment delay problem. Journal of the ACM, 48(2):243–273, 2001. doi: 10.1145/375827.375843. URL https://dl.acm.org/doi/10.1145/375827.375843.

Clint Ehrlich and Theodore Blackman. LCM: Lossless context management. arXiv preprint arXiv:2605.04050, 2026. URL https://arxiv.org/abs/2605.04050.

Donald Erlenkotter. Ford Whitman Harris and the Economic Order Quantity model. Operations Research, 38(6):937–946, 1990. doi: 10.1287/opre.38.6.937. URL https://pubsonline.informs.org/doi/10.1287/opre.38.6.937.

Xubin Hao, Hongjin Meng, Xin Yin, Jiawei Zhu, and Chenpeng Cao. Self-GC: Self-governing context for long-horizon LLM agents. arXiv preprint arXiv:2607.00692, 2026. URL https://arxiv.org/abs/2607.00692.

Ford W. Harris. How many parts to make at once. Factory, The Magazine of Management, 10 (2):135–136, 152, 1913. URL https://pubsonline.informs.org/doi/10.1287/opre.38.6.947. Reprinted in Operations Research 38(6):947–950, 1990.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 13358–13376. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.825. URL https://aclanthology.org/2023.emnlp-main.825/.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. 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 Forty-third International Conference on Machine Learning, 2026. URL https://arxiv.org/abs/2510.00615.

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. In Fourth Deep Learning for Code Workshop (DL4Code) at NeurIPS 2025, 2025. URL https://arxiv.org/abs/2508.21433.

Elias Lumer, Faheem Nizar, Akshaya Jangiti, Kevin Frank, Anmol Gulati, Mandar Phadate, and Vamse Kumar Subbiah. Don't break the cache: An evaluation of prompt caching for long-horizon agentic tasks. arXiv preprint arXiv:2601.06007, 2026. URL https://arxiv.org/abs/2601.06007.

Tony Mason. The missing memory hierarchy: Demand paging for LLM context windows. arXiv preprint arXiv:2603.09023, 2026. URL https://arxiv.org/abs/2603.09023.

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. arXiv preprint arXiv:2601.11868, 2026. URL https://arxiv.org/abs/2601.11868.

Nous Research. Hermes Compression Evaluation. Software repository, 2026. URL https://github.com/NousResearch/hermes-compression-eval. Accessed September 9, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. arXiv preprint arXiv:2310.08560, 2023. URL https://arxiv.org/abs/2310.08560.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon agent via context folding. In Forty-third International Conference on Machine Learning, 2026. URL https://openreview.net/forum?id=lNRgWoGfYg.

Di Wu, Zixiang Ji, Asmi Kawatkar, Bryan Kwan, Jia-Chen Gu, Nanyun Peng, and Kai-Wei Chang. LongMemEval-V2: Evaluating long-term agent memory toward experienced colleagues. arXiv preprint arXiv:2605.12493, 2026a. URL https://arxiv.org/abs/2605.12493.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. ReSum: Unlocking long-horizon search intelligence via context summarization. arXiv preprint arXiv:2509.13313, 2025. URL https://arxiv.org/abs/2509.13313.

Yong Wu, Yanzhao Zheng, Tianze Xu, Zhentao Zhang, Yuanqiang Yu, Jihuai Zhu, Chao Ma, Binbin Lin, Baohua Dong, Hangcheng Zhu, Ruohui Huang, and Gang Yu. ContextBudget: Budget-aware context management for long-horizon search agents. In Conference on Language Modeling (COLM), 2026b. URL https://arxiv.org/abs/2604.01664.

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. In Findings of the Association for Computational Linguistics: EMNLP 2026, 2026. URL https://arxiv.org/abs/2606.17016.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Jingren Zhou, Siheng Chen, and Yong Jiang. AgentFold: Long-horizon web agents with proactive context folding. In The Fourteenth International Conference on Learning Representations, 2026. URL https://openreview.net/forum?id=IuZoTgsUws.

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Pu Liang. MEM1: Learning to synergize memory and reasoning for efficient long-horizon agents. In The Fourteenth International Conference on Learning Representations, 2026. URL https://openreview.net/forum?id=XY8AaxDSLb.

A Cost Accounting and Scheduling Proofs

We derive the waiting–reconstruction decomposition, prove the stationary optimum and its first-crossing rule, and then state a finite-horizon bound for the online waiting–setup objective.

A.1 A Common Reference for Comparing Schedules

Let $L_t = N_t + R_t$ be prompt length. Input cost satisfies

$$N_t + w_r R_t = w_r L_t + (1 - w_r) N_t. \tag{15}$$

The first term charges every input token at the cached rate; the second adds the uncached-token premium.

Fix the stationary block stream and its eligibility times from Section 3.2. Define an ideal accounting reference with prompt length L_t^0 and uncached count N_t^0: it replaces every object at its first eligible request, receives cache hits for those replacements, and processes the same newly appended material. This reference is independent of batch size.

After the actual boundary decision, let U_t count removable tokens still present in eligible, uncommitted objects, and let V_t count additional uncached tokens caused by a commit. Then $L_t - L_t^0 = U_t$ and, under stable prefix reuse, $N_t - N_t^0 = V_t$. Subtracting the reference cost in Equation (15) gives

$$(N_t + w_r R_t) - (N_t^0 + w_r R_t^0) = w_r U_t + (1 - w_r) V_t. \tag{16}$$

Thus delayed shortening and reconstruction contribute separate terms.

Before a batch of b objects is committed, the wait requests carry $1, 2, \ldots, b-1$ pending objects. With $r = w_r g$, their total excess is

$$\sum_{j=1}^{b-1} w_r j g = \frac{rb(b-1)}{2} = W_b. \tag{17}$$

At the commit request, all pending objects become Cards, so $U_t = 0$. The cached interval $I_b = (b + h - 1)x$ shortens by bg, leaving $V_t = I_b - bg$ tokens to reconstruct. Including auxiliary overhead $F_0 + bf$, the cycle excess is

$$\begin{aligned} E_b &= W_b + (1 - w_r)(I_b - bg) + F_0 + bf \\ &= b \underbrace{\big[f + (1 - w_r)(x - g)\big]}_{K} + \underbrace{F_0 + (1 - w_r)(h - 1)x}_{Q_h} + \frac{rb(b-1)}{2} \\ &= bK + Q_h + W_b. \end{aligned} \tag{18}$$

Each cycle contains b requests and processes b eligible objects, giving excess $E_b/b = \ell_h(b)$ per request and per object. Initial and final incomplete cycles vanish from this average as the stream grows with fixed b.

The first-request comparison and this common-reference accounting agree because

$$(1 - w_r)I - G = (1 - w_r)(I - G) - w_r G : \tag{19}$$

the reference already receives the $w_r G$ shortening benefit on the commit request, leaving $(1 - w_r)(I - G)$ as excess reconstruction.

A.2 Stationary Optimum and First Crossing

Proof of Proposition 1. Since K is independent of b, minimize $Q_h/b + r(b-1)/2$. For $Q_h > 0$ and $r > 0$,

$$\ell_h'(b) = -\frac{Q_h}{b^2} + \frac{r}{2}, \qquad \ell_h''(b) = \frac{2Q_h}{b^3} > 0. \tag{20}$$

The unconstrained minimizer is $\sqrt{2Q_h/r}$; imposing $b \geq 1$ gives Equation (7). If $Q_h = 0 < r$, the objective increases with b, giving $b = 1$.

For integer $b \geq 1$, define

$$d_b = \ell_h(b+1) - \ell_h(b) = \frac{r}{2} - \frac{Q_h}{b(b+1)}. \tag{21}$$

These differences are nondecreasing. For $b \geq 2$, optimality is therefore equivalent to $d_{b-1} \leq 0$ and $d_b \geq 0$, or

$$\frac{rb(b-1)}{2} \leq Q_h \leq \frac{rb(b+1)}{2}. \tag{22}$$

For $b = 1$, only $d_1 \geq 0$ is required, giving $Q_h \leq r$; the lower bound follows from $Q_h \geq 0$. □

Choosing the smaller batch in a tie gives

$$b^\dagger = \max\left\{1, \left\lceil \frac{\sqrt{1 + 8Q_h/r} - 1}{2} \right\rceil \right\}. \tag{23}$$

This is the smallest positive integer with $rb(b+1)/2 \geq Q_h$. Equality gives adjacent optima b and $b+1$; otherwise the optimum is unique. Under a fixed integer cap $B_{\max}$, the smallest optimal feasible size is $\min\{b^\dagger, B_{\max}\}$.

First crossing. Starting from an empty pending sequence, one object becomes eligible per request. When the next batch first contains b objects, their waiting ages are $b-1, b-2, \ldots, 0$, so

$$G_b = bg, \qquad W_b = \frac{rb(b-1)}{2}, \qquad W_b + w_r G_b = \frac{rb(b+1)}{2} = W_{b+1}. \tag{24}$$

For $r > 0$, the first crossing $W_b + w_r G_b \geq Q_h$ consequently selects $b^\dagger$. It also directly establishes both optimality inequalities: the current crossing gives the upper bound on Q_h, while for $b > 1$ the preceding noncrossing gives

$$W_b = W_{b-1} + w_r G_{b-1} < Q_h. \tag{25}$$

For $b = 1$, the lower bound is $W_1 = 0 \leq Q_h$. Equality at the current crossing selects the smaller adjacent optimum. A confirmed full commit resets the ledger, and the argument repeats in the next cycle. This recovery uses equal object gains, one eligible block per request, constant Q_h and w_r, and stable prefix caching. A fixed batch cap truncates the selected size to $B_{\max}$.

Zero-cost cases. If $r = 0 < Q_h$, $K + Q_h/b$ strictly decreases: the largest feasible batch is optimal under a fixed cap, and there is no finite optimum without a cap. If $Q_h = 0 < r$, immediate commitment is optimal; if both vanish, all batch sizes have equal modeled excess. With $h = 1$, the cache timing gives $Q_h = F_0$, since no cached hot-tail suffix remains to reconstruct.

A.3 Online Quantities and Ledger Updates

Pending region and shared suffix. For a candidate commit inside a reusable prefix, split the affected interval at the end of the newest pending block. Let A_t cover the interval from the first edit to that boundary, including unchanged content between replacements. The remaining cached suffix has length H_t^{shared}. Using $Q_t = F_{0,t} + (1 - w_r)H_t^{\text{shared}}$, the reconstruction premium is

$$F(\mathcal{P}_t) + (1 - w_r)(A_t - G_t + H_t^{\text{shared}}) = \underbrace{F(\mathcal{P}_t) - F_{0,t} + (1 - w_r)(A_t - G_t)}_{\text{pending-region cost}} + Q_t. \tag{26}$$

In the stationary model, these terms are bK and Q_h: the pending region has constant cost per object, while batching amortizes Q_h. This gives the shared-cost threshold in Equation (14). With heterogeneous blocks, internal gaps, or nonlinear overhead, the pending-region cost can vary per object; the runtime applies the shared-cost crossing to its current estimates.

The split uses the end of the newest pending block, so trailing dialogue within that block belongs to A_t. Actual edit positions also determine the interval: if the first edit starts s tokens into its block, the homogeneous interval is $I_b = (b + h - 1)x - s$. The stationary derivation uses $s = 0$; runtime estimates use the actual edit positions and reusable cache frontier.

Waiting ledger. A request carrying pending contents raw adds $w_r G_t$ to their ledger, and newly eligible contents enter with zero accumulated loss. For fixed additive gains this gives

$$W_{t+1} = W_t + w_r G_t, \qquad W_t = w_r \sum_i g_i a_i(t), \tag{27}$$

where $a_i(t)$ counts eligible requests carrying occurrence i raw. A successful model response confirms a full commit and clears its entries. A retrieved tool-result occurrence starts its own exposure and waiting ledger, independently of the historical Card, and can later become a retrieval receipt. The runtime aggregates persistent per-delta entries and estimates G_t from whole-prompt lengths; its recorded ledger uses these estimates rather than canonical per-object token sums.

Cache state. Reconstruction includes only tokens that would otherwise be reused, excluding natural cache misses and the newest uncached block. The increment $w_r G_t$ is the cached retention cost; for pending payloads that miss the cache, it serves as a cached-rate estimate. Irregular arrivals, retrieval, and changing Q_t are handled through the current quantities. The following online result treats exact waiting costs and schedule-independent shared setup costs.

A.4 A Competitive Bound for the Waiting–Setup Objective

To isolate the online batching trade-off, fix requests $t = 1, \dots, T$ and each object's eligibility time $a_i \in \{1, \dots, T\}$. An eligible object i incurs a fixed cost $r_i \geq 0$ on every request that retains it raw. A commit replaces all pending objects and incurs shared setup cost Q_t. Eligibility, object gains, and Q_t are independent of the schedule; objects remain pending until committed, and either action is feasible at every boundary. Let $\mathcal{U}_t^\pi$ be the objects left pending after schedule π's decision at t. The waiting–setup objective is

$$J(\pi) = \sum_{t=1}^{T} \sum_{i \in \mathcal{U}_t^\pi} r_i + \sum_{t:\, \pi \text{ commits at } t} Q_t. \tag{28}$$

Objects may remain raw when the history ends. This is the setup-versus-delay objective of online acknowledgment batching (Dooly et al., 2001). Here it isolates waiting and shared reconstruction from the full input-cost decomposition. With additive token gains and cached retention, $r_i = w_r g_i$.

Proposition 2 (Online waiting–setup bound). Suppose $0 < Q_{\min} \leq Q_t \leq Q_{\max}$. Let W_t be the waiting cost already incurred by the current pending batch and $\rho_t = \sum_{i \in \mathcal{P}_t} r_i$ its cost if retained on request t. The rule that commits exactly when $\mathcal{P}_t \neq \varnothing$ and $W_t + \rho_t \geq Q_t$ satisfies

$$J(\mathrm{ALG}) \leq \frac{2Q_{\max}}{Q_{\min}} J(\mathrm{OPT}), \tag{29}$$

where OPT minimizes Equation (28) with full knowledge of the history. In particular, the rule is 2-competitive for constant positive Q_t.

Proof. Let $\tau_1 < \cdots < \tau_m$ be the algorithm's commit times and set $\tau_0 = 0$. A completed phase is the request interval $(\tau_{j-1}, \tau_j]$; its objects are those becoming eligible in that interval. Since every commit clears the pending set, the algorithm's phase cost is $W_{\tau_j} + Q_{\tau_j}$. If the phase has more than one request, the preceding noncrossing gives

$$W_{\tau_j} = W_{\tau_j - 1} + \rho_{\tau_j - 1} < Q_{\tau_j - 1} \leq Q_{\max}. \tag{30}$$

For a one-request phase, $W_{\tau_j} = 0$. Thus every completed phase costs at most $2Q_{\max}$.

If OPT commits during that phase, assign one such setup cost to the phase; it is at least $Q_{\min}$. Otherwise, OPT retains all the phase's objects on every request from their eligibility through τ_j. Assign their waiting cost in this interval, which is

$$W_{\tau_j} + \rho_{\tau_j} \geq Q_{\tau_j} \geq Q_{\min}. \tag{31}$$

The term ρ_{τ_j} is the cost of the current request, so this argument also applies when $\tau_j = T$. Each completed phase therefore costs ALG at most $2Q_{\max}/Q_{\min}$ times its assigned OPT cost.

In a final uncompleted phase, ALG's total cost is zero if no objects arrive; otherwise the last noncrossing gives $W_T + \rho_T < Q_T \leq Q_{\max}$. If OPT commits in this phase, assign one setup cost, at least $Q_{\min}$. If it does not, assign the waiting cost of the phase's objects, equal to ALG's cost. This phase satisfies the same bound. All assigned costs are distinct: setups belong to disjoint phases, and waiting charges belong to distinct phase objects and request intervals. Their sum is at most $J(\mathrm{OPT})$, proving Equation (29). □

If pending-region reconstruction contributes the same nonnegative total B to every feasible schedule, the bound also holds for $B + J$: for $\alpha = 2Q_{\max}/Q_{\min} \geq 1$, $B + J(\mathrm{ALG}) \leq \alpha[B + J(\mathrm{OPT})]$.

B Experimental Configuration and Evaluation Protocols

Benchmark chains. The continuing task lists contain 89 Terminal-Bench 2.1 tasks, the first 18 evaluated LongMemEval-V2 tasks, and 50 SWE-bench Verified tasks. All runs request the gpt-5.6-terra route. Each benchmark uses one continuing chain per arm, with history retained across tasks. Task identities and positions align where task-level exports are available; model and tool trajectories develop separately. The evaluated scope includes processed requests on the completed task lists. Terminal-Bench includes a resumed execution, with cache continuity reconstructed at the continuation boundary. All benchmarks use Equation (1); prompt accounting uses o200k_base and the versioned serializer logical-prompt-json-v1.

HCE replay and aggregation. The three frozen HCE histories cover configuration, debugging, and feature implementation. They contain 25, 21, and 37 source-request boundaries, respectively, and 44 frozen Card occurrences in total. Object identities, immutable versions, original bytes, and prepared Cards are fixed across configurations at $h = 2, 4, 8, 16$. Prompts use canonical_json_sorted_utf8:v1, with tiktoken:o200k_base:0.11.0 tokenization and adjacent-request prefix reuse under canonical_prefix_reference. Reported costs sum $N + 0.1R$ across all three histories. Aggregate saving is $1 - \sum_j C_{p,j} / \sum_j C_{\mathrm{Full},j}$ for policy p and histories j.

Hot tail and production configuration. All strategies use both the block cap h and the 12,800-token hot-tail cap. Production CADOC additionally confirms an occurrence's initial raw exposure after a successful model response. Its token estimate covers the suffix starting at the earliest protected occurrence, whereas baseline replay logs the corresponding recent-block token sum. The exported hot_tokens and hot_block_count therefore describe each replay path's partition. Effective configurations and governing source files accompany the code.

CADOC uses rough prompt/cache estimates, a persistent per-delta waiting ledger, and zero auxiliary commit overhead: $Q_t = (1 - w_r)H_t^{\mathrm{shared}}$. Pending guards fire above $2h$ runtime inference groups or 25,600 raw tokens; baseline capacity guards use 16 pending blocks or 65,536 raw tokens. All 35 observed CADOC commits follow the economic crossing and are confirmed after a successful response. Fixed $b = 16$, Token 32,768, and Token 65,536 make no commits on these finite HCE histories and therefore equal Full Context; the main figure shows the eight active configurations.

Continuation evaluation. Six policies are evaluated on 31 end-of-history probes, each repeated with seeds 20260907, 20260908, and 20260909. Pass counts pool the 93 evaluations.

Session-macro scores average repeats within probes, probes within each history, and the three histories equally. Answering and judging request gpt-5.6-terra. Section D reports the aggregate policy comparison.

B.1 Fixed-History Replay of Benchmark Traces

The benchmark replay uses $h = 8$ and $w_r = 0.1$ on 1,801 recorded FULL request boundaries: 1,013 spanning Terminal-Bench tasks, 691 from SWE-bench, and 97 spanning LongMemEval tasks. Source outputs and deterministic Cards are fixed, all configurations use a shared inference candidate pool, and CADOC runs its production scheduler and lifecycle. Context and cache state continue across tasks; Terminal-Bench continuation follows the retained source-task order. Each stream ends at its last observed input. The replay performs no model calls.

C Object Implementation and Recorded Retrieval

The Card shown in Section 2 comes from LongMemEval-V2 task 0f970f01, which asks which column lies between Price and Delivery time in a ServiceNow Catalog Items list. A terminal search returns trajectory locations as a structured tool result. CADOC preserves its original bytes and replaces its historical occurrence with a deterministic Card containing a versioned object reference and structural metadata.

Retrieval and subsequent use. Request 11 contains the original result; request 12 contains its Card after an economic-crossing commit. The model then issues the recorded call:

```
{"object_ref":
   "object://obj_0722dfa749e04b58a0094af4@v1",
   "reason": "Need the exact trajectory ID and state index returned by the
evidence search for the Catalog Items column-header question."}
```

Request 13 contains the call and its full-object result. The original payload and the returned retrieved_object.content both contain 9,925 UTF-8 bytes and have identical recomputed SHA-256 hashes. The retrieved content identifies trajectory 3c588c61, state 33; the next action in request 14 uses that location:

```
python3 enterprise/show_trajectory.py 3c588c61 –start 33 –end 39
```

The task completes correctly. Runtime estimates are 2,473 tokens for the object and 174 for its Card.

Retrieval returns the immutable referenced version as a new tool-result occurrence. The historical Card remains in place, while the returned occurrence receives its initial raw exposure and then follows the same hot-tail and commit lifecycle, including replacement by a retrieval receipt. The evidence package contains the complete Card, original and returned payloads, request snapshots, grading record, and registration/retrieval source code.

D Aggregate HCE Continuation Quality

Table 2 reports the six-policy comparison at each hot-tail setting, aggregated over the same 93 probe evaluations using the protocol in Section B. Full Context uses the same unchanged control across all four settings.

Table 2: Continuation passes and session-macro scores for six policies across four hot-tail settings.

	$h = 2$		$h = 4$		$h = 8$		$h = 16$	
Policy	Passes	Macro	Passes	Macro	Passes	Macro	Passes	Macro
Full	57/93	4.318	57/93	4.318	57/93	4.318	57/93	4.318
Immediate	64/93	4.326	59/93	4.274	64/93	4.285	63/93	4.404
Fixed $b = 4$	65/93	4.318	64/93	4.346	62/93	4.305	61/93	4.354
Fixed $b = 8$	63/93	4.336	59/93	4.263	64/93	4.353	69/93	4.538
Token 16,384	65/93	4.421	60/93	4.247	67/93	4.492	67/93	4.447
CADOC	61/93	4.297	61/93	4.204	65/93	4.418	62/93	4.330

E Recorded Scheduling Decisions

Table 3 reconstructs consecutive production decisions at $h = 4$ from the runtime's W, G, and Q, with G given by its rough keep/commit prompt difference. The inequality changes from $4,787.1 < 5,688.9$ to $6,274.9 \geq 6,124.5$, triggering a commit. That batch replaces five objects containing 15,006 raw tokens with 764 Card tokens, a 94.91% reduction.

Table 3: Two decisions to show the process; boundaries are zero-based request indices.

Boundary	W	G	Q	$W + 0.1G$	Decision
14	3,424.9	13,622	5,688.9	4,787.1	Wait
15	4,787.1	14,878	6,124.5	6,274.9	Commit

Across all 184 nonempty-pending boundaries, recomputing $W + 0.1G - Q$ matches every recorded decision. Holding preceding history fixed, 172 of these decisions remain unchanged throughout independent $\pm 5\%$ perturbations of the recorded W, G, Q.

F Sensitivity to Cache-Read Weight

Protocol. We use the benchmark histories from Section B.1 and test $w_r \in \{0.05, 0.1, 0.25, 0.5, 1\}$ with input cost $C = N + w_r R$. CADOC executes its production planner separately at each weight. The nine baselines have weight-independent triggers, so their fixed token trajectories are repriced at each weight. Within each benchmark, request boundaries, inference candidates, Card bytes, and context budgets are fixed. Each history starts with an empty cache and ends at its last recorded input.

Results. CADOC attains the lowest input cost among the ten compared policies in all fifteen benchmark–weight combinations (Figure 7 and Table 4). It ties with Immediate at $w_r = 1$ on all three histories and at $w_r = 0.5$ on LongMemEval, and is uniquely best in the remaining combinations.

Immediate commitment without a cache discount. At $w_r = 1$, cached and uncached input tokens have equal cost. The evaluated scheduler uses zero auxiliary commit overhead, $F_{0,t} = 0$, so Equation (13) gives

$$Q_t = F_{0,t} + (1 - w_r)H_t^{\text{shared}} = 0. \tag{32}$$

For a nonempty eligible batch, $W_t \geq 0$ and $G_t \geq 0$, so the commit condition in Equation (14) reduces to $W_t + G_t \geq 0$ and is satisfied at every feasible boundary. CADOC therefore commits as soon as replacement is permitted, matching Immediate's commit timing and input cost. Immediate is the lowest-cost competing compressor at this endpoint on all three histories; CADOC's relative advantage is consequently zero, while its savings relative to Full remain positive.

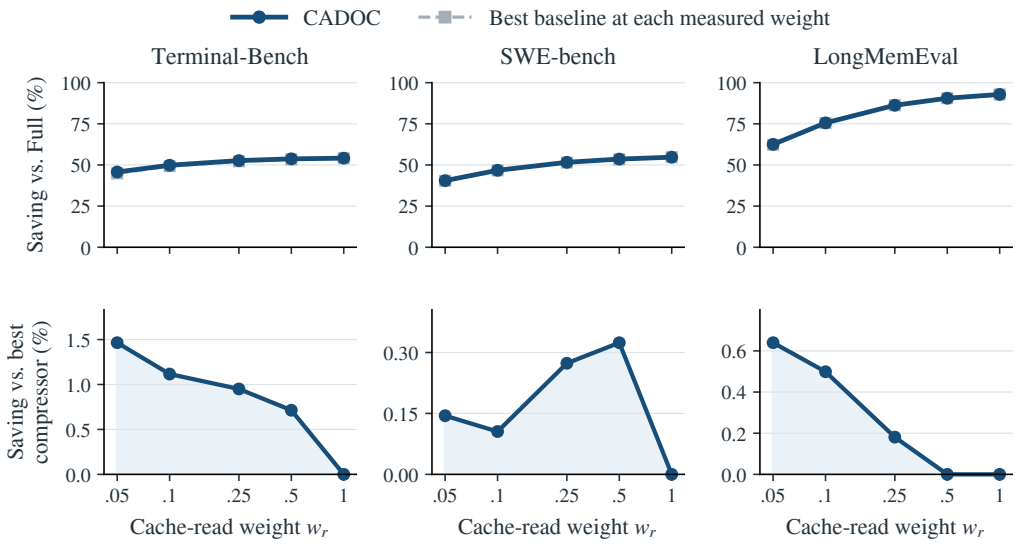


Figure 7: Cache-read weight sensitivity. Top: savings relative to Full for CADOC and the best of nine baselines, including Full, at each weight. Bottom: savings relative to the best of eight other compression policies, with separate vertical scales starting at zero. Markers are measured points; lines guide the eye.

Table 4: CADOC input-cost savings relative to Full at five cache-read weights. Pooled savings use the ratio of summed costs across the three benchmark histories.

w_r	Terminal-Bench	SWE-bench	LongMemEval	Pooled
0.05	45.73%	40.48%	62.52%	45.06%
0.1	49.83%	46.75%	75.57%	50.39%
0.25	52.66%	51.61%	86.28%	54.26%
0.5	53.71%	53.57%	90.55%	55.75%
1	54.12%	54.72%	92.85%	56.48%