PERSISTENT CONTEXT GRAPHS FOR EFFICIENT MEMORY COMPACTION IN LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As LLM capabilities advance, agents are tackling increasingly complex tasks over longer horizons. Their growing interaction histories make memory compaction essential for staying within context windows and reducing prefill cost. Existing methods summarize the history or compress its KV cache, often adding model computation to preserve information for future requests. A new user request can change which history matters, but reassessing that history with the model requires re-encoding it if the KV cache has expired. Past attention provides signals of historical importance and dependencies between messages, while relevance to the current task must be assessed using the new user request. We introduce RECAP, a memory compaction method that stores attention-derived importance scores and dependency links in a lightweight, persistent context graph. For each new request, RECAP combines stored importance with relevance cues from the request and follows dependency links to select messages and their supporting context, without additional model calls for selection. Compared with Codex's default summarization-based compaction, RECAP reduces estimated latency for compaction and cold restoration by approximately 95% on both Qwen3-Coder and gpt-oss. It also roughly halves the historical context per call on SWE-Together at comparable task quality and improves accuracy on the code tasks of Lost-in-Conversation over full history by 19.8 and 41.2 points.

1 INTRODUCTION

Advances in LLM capabilities are enabling agents to solve increasingly complex tasks over longer horizons (*e.g.*, repository-level coding and autonomous research). Such workflows can run for tens of minutes to hours, accumulating user instructions, intermediate decisions, and tool outputs as execution proceeds. The resulting history can eventually exceed the model's finite context window. It also raises inference cost: measurements of agentic workloads report inputs tens of times longer than their generated outputs, making repeated context encoding a substantial part of execution (Lu et al., 2026). Modern agent harnesses and recent research therefore adopt *memory compaction*, which consolidates session memory into a shorter context (Kang et al., 2025). By limiting the history carried into subsequent calls, compaction allows the agent to continue beyond a single context window while reducing repeated prefilling.

Existing approaches compact agent memory either by summarizing the history into a shorter prompt or by retaining a subset or compressed form of the model's KV cache (Kang et al., 2025; Li et al., 2024; Zweiger et al., 2026). Doing so efficiently across an interactive session involves three linked challenges. ❶ **What to retain depends on future use.** A new user request can change the task's direction or revisit an earlier decision, so compaction performed before it arrives must preserve information for needs not yet known. Query-aware KV eviction, for instance, degrades when later queries differ from the one that guided eviction (Li et al., 2025; Kim et al., 2026). ❷ **Anticipating future use requires additional computation.** Deciding which history a later request will use requires queries that resemble that request, which do not yet exist when compaction runs. Attention Matching (Zweiger et al., 2026), for example, synthesizes such reference queries by re-prefilling the context or simulating interactions and fits the compact cache to them, adding model computation to every compaction. ❸ **Cache expiration adds reconstruction cost.** In about $4{,}300$ Claude Code and Codex sessions, user think time accounts for 92% of session time, and requests after hour-long gaps almost always miss the serving cache (Zhu et al., 2026). Compacted KV states must then be

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.

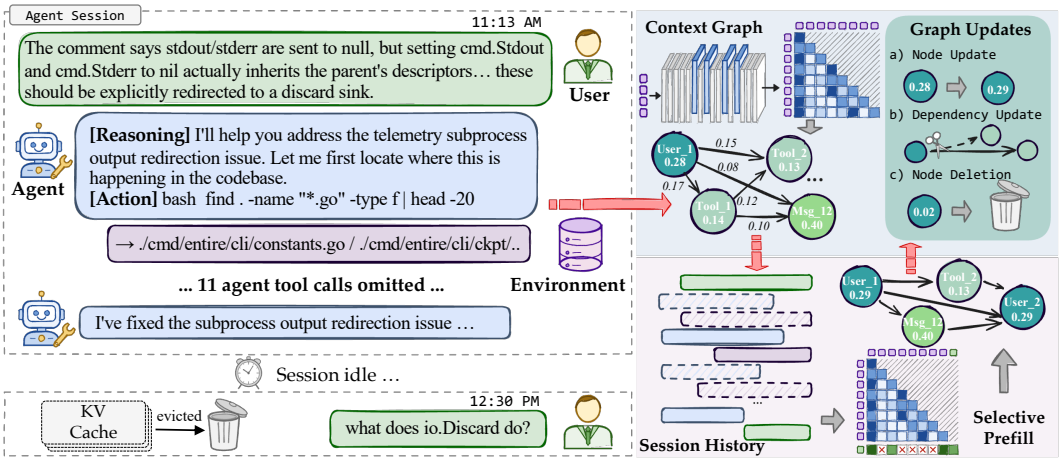

Figure 1: RECAP preserves attention-derived importance and dependency links in a persistent context graph. For each new request, it combines stored importance with the request's relevance cues to select historical blocks and their supporting context, retaining protected instructions and recent modifications. The paused session shows how the graph guides compaction after KV-cache eviction.

stored and reloaded (Gao et al., 2025), and a compactor that waits for the request must re-encode the history to obtain fresh attention.

Each forward pass of the agent already computes attention over its history, yet this signal is discarded once the request completes. We study how coding agents attend to their histories across user turns and find that attention concentrates on a small part of the history, that past attention predicts which history the next turn reuses, and that the arriving request redirects attention (§3.2). In light of these findings, we propose RECAP, a memory compaction method that keeps this attention rather than discarding it and organizes it in a lightweight, *persistent context graph* (Figure 1): nodes are history blocks annotated with accumulated importance, and edges link each block to the earlier blocks it depends on. When a new request arrives, RECAP selects from the full history by combining stored importance with the request's identifier overlap and following dependency edges, so selection adapts to the actual request and can restore previously omitted blocks (❶). The graph is built from attention that inference computes anyway, and selection uses only graph traversal and string matching, so compaction adds no model call (❷). Because the graph persists independently of the KV cache, RECAP can select after the cache expires without re-encoding the full history (❸).

Our contributions are as follows:

- We show that attention from ordinary agent inference concentrates on a small part of the history, predicts which history the next turn reuses, and is redirected by the arriving request (§3.2).
- We propose RECAP, which persists this attention in a context graph of block importance and dependencies and queries it for each new request to select context without an additional model call, independently of the KV cache (§4).
- On SWE-Together (Wu et al., 2026b) and the code tasks of Lost-in-Conversation (Laban et al., 2026) with two model families, RECAP reduces the estimated latency of compaction and cold restoration by approximately 95% relative to Codex's default summarization, roughly halves the historical context on SWE-Together at comparable quality, and improves Lost-in-Conversation accuracy over full-history conditioning by 19.8 and 41.2 points (§5).

2 RELATED WORK

Context management for LLM agents. Agent harnesses summarize long histories with the backbone, and recent work optimizes such compressors or trains agents to fold their own context (Kang et al., 2025; Sun et al., 2025; Zhou et al., 2026); both add model computation and discard detail irreversibly. Cheaper alternatives mask old observations (Lindenbauer et al., 2025) or prune text with small models (Pan et al., 2024; Jiang et al., 2024). Two recent methods select history by depen-

dencies or attention at higher cost. ContextWeaver calls the backbone at every step to identify the earlier steps it depends on and to summarize these dependencies, and it replaces the observations of all other steps with placeholders (Wu et al., 2026a). AttnCompress reruns a separate 4B-parameter model over the trajectory and scores blocks by the attention of the agent's next generation step, recomputing all scores at each refresh (Zeng et al., 2026). Both target autonomous single-issue runs, in which no new user request redirects the task. RECAP records importance and dependencies from the agent's own attention during inference and accumulates them across turns. When a user request arrives, it selects from the full archive without a model call, so omitted blocks can return.

KV-cache compression and reuse. KV methods evict cached entries by attention statistics (Zhang et al., 2023; Li et al., 2024; Feng et al., 2026) or fit compact states (Zweiger et al., 2026), and query-aware eviction degrades when later queries differ (Li et al., 2025; Kim et al., 2026). In agents, compacting each turn immediately loses accuracy, while waiting for later queries requires keeping the uncompressed cache meanwhile (Liu et al., 2026); learned cross-turn scoring still evicts entries permanently (Li et al., 2026). Recallable methods and serving systems avoid such loss by keeping full caches or session state in GPU or host memory (Tang et al., 2024; Xiao et al., 2024; Gao et al., 2024; 2025). Yet user think time dominates coding-agent sessions, and caches often expire across these gaps (Zhu et al., 2026). RECAP waits for the actual request without holding KV state: its text archive and graph survive cache expiration, and it coexists with prefix caching.

3 MEMORY COMPACTION: SETTING AND OBSERVATIONS

3.1 QUERY-CONDITIONED MEMORY COMPACTION

An agent session alternates between user requests and agent execution. In turn t, the user sends a request q_t, and the agent produces assistant messages, tool calls, and tool results before returning control; H_t denotes the accumulated history. When q_{t+1} arrives, memory compaction constructs a smaller working context from H_t for the new request, limiting the history carried into subsequent calls and thus both context-window pressure and encoding cost. If the previous KV state has expired or been evicted, the engine rebuilds it by prefilling this compact context.

We study compaction by selecting historical message blocks. Let V_t contain the selectable blocks in H_t, excluding system instructions (§4.1), and let $C \subseteq V_t$ be the selected set, with cost $c(C) = \sum_{v \in C} c(v)$ in characters of message content and serialized tool calls. A nominal retention fraction $\beta \in (0, 1]$ sets the packing budget $B_t = \lfloor \beta\, c(V_t) \rfloor$. System instructions and the current turn are rendered separately. Our method prioritizes protected blocks over this nominal budget and allows bounded dependency expansion (§4.3); we measure the resulting token count, latency, and quality.

We seek a compactor that *(i)* selects context from stored information without an additional model call when a request arrives, *(ii)* keeps tool calls paired with their results, *(iii)* keeps the selected historical prefix fixed across tool steps within a user turn for prefix-cache reuse, and *(iv)* adapts to the arriving request, including requests that revisit earlier work. These requirements separate two sources of evidence: historical usage becomes available during execution, whereas relevance to the next request can be assessed only after that request arrives. A reusable representation of past usage can connect these two stages, even when the KV state is no longer available.

3.2 WHAT ATTENTION REVEALS ABOUT USEFUL CONTEXT

We examine whether attention provides a historical importance signal, whether that signal predicts later reuse, and how the arriving request changes access to the same history, using a multi-turn example, a cross-turn reuse study, and a controlled query study (Figure 2; protocols in Appendix B).

Observation 1: attention concentrates on a small part of the history. In one coding session (Figure 2, left), attention is unevenly distributed over historical blocks at every checkpoint: ranking blocks by attention per token within a 10% historical-token budget retains 65% of the attention mass at the final checkpoint. The reuse-study transitions show the same concentration (Appendix B). This concentration gives a compactor a ranking signal for allocating a limited context budget.

Observation 2: past attention predicts subsequent reuse. We evaluate selection signals over 64 turn transitions from eight coding-agent sessions, approximating reuse by the IDF-weighted cov-

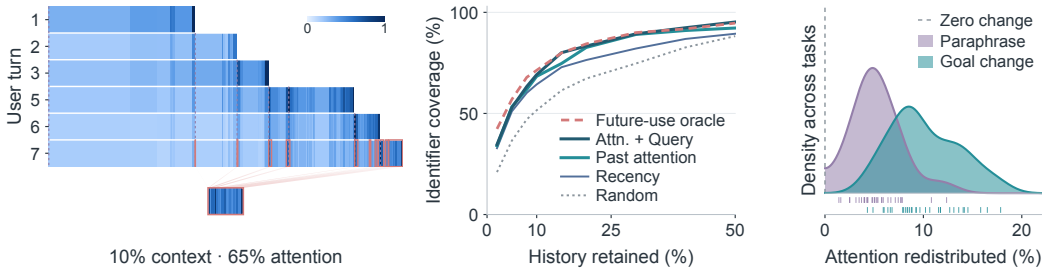

Figure 2: **Attention signals for memory compaction. Left:** Attention over history blocks per turn (width: tokens; dashed: user turns; blue: attention density, cube-root scale). Coral: blocks selected under a 10% token budget, concatenated below. **Middle:** Next-turn identifier coverage versus retained-history budget; *Attn. + Query* adds query-identifier overlap. **Right:** Attention redistribution (TV distance) over 32 fixed histories for paraphrases versus changed targets; ticks mark tasks.

erage of identifiers that appear during the next turn's execution. At a 10% historical-text budget, ranking by attention from the preceding turn reaches a median coverage of about 0.68, close to the 0.71 of a future-use oracle that ranks blocks by those identifiers (Figure 2, middle). Past attention thus supplies an importance signal that remains useful beyond the turn in which it was measured. Adding overlap with the incoming query's identifiers yields about 0.69. We next hold the history fixed to isolate the request's effect on attention.

Observation 3: the arriving request redirects attention. For each of 32 tasks, we hold one historical context fixed and vary only the request, using two phrasings for each of two target files, and measure attention change over historical blocks by total-variation distance. Changing the target redistributes 10.1% of attention on average, compared with 5.3% for paraphrasing (Figure 2, right), and produces greater redistribution in 28 of the 32 tasks. This pattern also holds for requests that explicitly ask the agent to recall earlier work (Appendix B.2). Historical importance therefore supports selection across turns, while the arriving request adds a cue for relevance to the current task.

These observations motivate preserving historical importance between turns and combining it with the current request when selecting context. RECAP does so with a persistent context graph: message nodes retain attention-derived importance, and attention links provide dependency cues for supporting context, independently of the KV cache.

4 RECAP: MEMORY COMPACTION WITH A PERSISTENT CONTEXT GRAPH

RECAP maintains a persistent context graph that connects past attention to future compaction decisions: node scores summarize historical importance, and directed edges record attention-based dependencies between messages. It *updates* the graph using attention from agent execution and *queries* it when a new user request arrives, combining stored importance with relevance to that request and including supporting context through graph traversal. Figure 3 illustrates the workflow; Algorithm 1 in Appendix D.2 gives the complete procedure.

4.1 MODELING SESSION HISTORY AS AN EVOLVING CONTEXT GRAPH

Each node represents an *atomic block* of history that can be carried into the working context: a user message, an assistant message that issues tool calls together with their results, or a plain assistant message. Grouping tool calls with their results preserves a valid message sequence when blocks are selected. Nodes receive stable identifiers in chronological order as the history grows.

We define the context graph as a directed graph $G_t = (V_t, E_t)$ that evolves with each turn, where each node $v \in V_t$ is an atomic block and carries an importance score $u_t(v) \in [0, 1]$ as a node attribute. Each edge $e = (a, b) \in E_t$ is an ordered pair from an earlier supporting block a to a later dependent block b and is associated with a weight $w_t(a, b) > 0$ that indicates the strength of the dependency: selecting a diagnosis, for example, can bring in the failure trace it relies on. The graph stores block references, scalar scores, and dependency links alongside the full history.

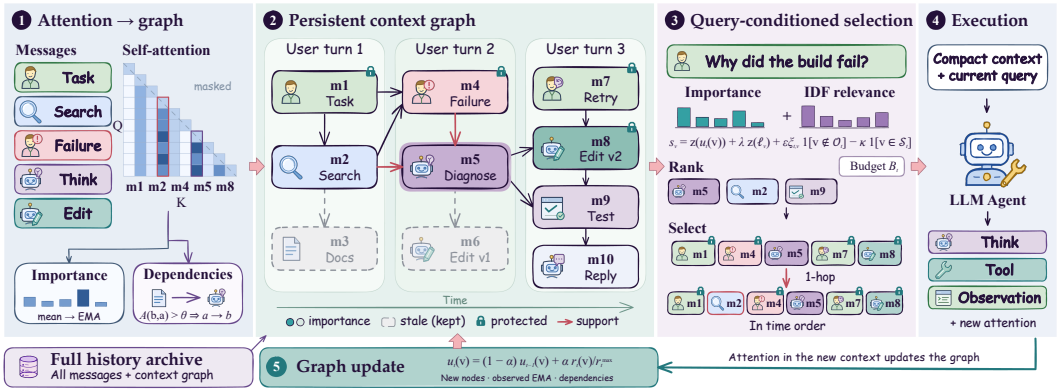

Figure 3: **ReCAP workflow. 1–2:** Sampled attention supplies node importance and support-to-dependent edges. **3:** Selection combines stored importance with the query's identifier overlap, keeps protected records, and adds one-hop predecessors (diagnosis m_5 brings in search record m_2), without a model call. **4–5:** Execution produces new messages and attention that update the graph.

These records persist across turns independently of the KV cache, so later requests can select any accumulated block, even one previously omitted.

4.2 Persisting Importance and Dependencies from Attention

At each turn boundary, ReCAP consolidates attention over the working context to node scores and dependency links. Let $R_t \subseteq V_t$ be the blocks observed in this update, including those created during the turn. Attention from the completed turn measures each block's use, and attention between blocks identifies supporting relationships; both use sampled attention rows (Appendix D.1).

Accumulating historical importance. We sample token positions W_t from the completed turn and average their attention over heads and selected layers; after clipping extreme values, $\rho_t(j)$ denotes the attention received by historical token j. For each block $v \in R_t$, we average ρ_t over its tokens $\mathrm{span}_t(v)$ and update its importance with an exponential moving average:

$$\bar{\rho}_t(v) = \frac{1}{|\mathrm{span}_t(v)|} \sum_{j \in \mathrm{span}_t(v)} \rho_t(j), \qquad u_t(v) = (1-\alpha) u_{t-1}(v) + \alpha \frac{\bar{\rho}_t(v)}{\bar{\rho}_t^{\max}}, \tag{1}$$

where $\bar{\rho}_t^{\max} = \max_{v \in R_t} \bar{\rho}_t(v)$ and new nodes start at zero. Per-token averaging accounts for block length, while the moving average combines recent usage with previous observations. Nodes outside R_t retain their stored scores, keeping this evidence for later requests that revisit omitted history.

Recording dependencies. To identify the context supporting a block b, we sample positions within b and measure the mean attention mass they assign to each earlier block a at a selected layer. When this mass exceeds a threshold θ, we add the edge (a, b), pointing from support to dependent (the reverse of the attention direction). Edges accumulate across turns, and each weight $w_t(a, b)$ keeps the largest mass observed so far. During compaction, they let a selected block retrieve its supporting predecessors, including relationships formed within a single turn.

The graph also tracks observation status and potential supersession. The set $\mathcal{O}_t = \mathcal{O}_{t-1} \cup R_t$ records which nodes have received attention measurements, and a stale set $\mathcal{S}_t$ marks earlier blocks whose identifiers substantially overlap a later block that receives stronger attention (Appendix D.2). These marks lower the priority of potentially superseded content while preserving its node and text.

4.3 Querying the Graph for Memory Compaction

When q_{t+1} arrives, ReCAP queries G_t to construct a compact context for it. Historical importance supplies the prior usage signal established in §3.2, and the request supplies the relevance cue. Selection combines these signals to rank nodes, packs them with protected records, and follows dependency edges to add supporting context, using only stored graph state and string matching.

Query-conditioned node scores. We compute a relevance cue from identifier overlap between the request and each block, weighted by inverse document frequency:

$$\ell_v = \sum_{w \in I(q_{t+1}) \cap I(v)} \log \frac{|V_t|}{\mathrm{df}_t(w)}, \tag{2}$$

where $I(\cdot)$ extracts identifiers from message text and $\mathrm{df}_t(w)$ counts historical blocks containing w. The weighting emphasizes explicit references to rare file names, functions, and error symbols. We combine this cue with stored importance and the node annotations:

$$s_v = \mathrm{z}(u_t(v)) + \lambda\, \mathrm{z}(\ell_v) + \epsilon\, \xi_{t,v}\, \mathbf{1}[v \notin \mathcal{O}_t] - \kappa\, \mathbf{1}[v \in \mathcal{S}_t]. \tag{3}$$

Here z standardizes each channel over V_t and λ controls the query contribution; never-observed nodes receive a small exploration bonus ($\xi_{t,v} \in [0.5, 1]$, drawn with a fixed session-and-turn seed), and stale nodes receive a penalty κ. The first two terms combine historical importance with current relevance, while the annotations adjust the priority of unmeasured and potentially superseded blocks.

Packing historical blocks. Selection begins with a protected set P_t: all user-message nodes and the latest edit or write node per modified file, preserving task instructions and the agent's latest changes (Appendix B.3). Starting from $C = P_t$, we visit the remaining nodes in descending score order, breaking ties chronologically, and add each block that fits within the nominal budget B_t (§3.1). P_t is retained even when its cost exceeds B_t. Let C_0 denote this initial selection.

Including supporting context. For each selected block $b \in C_0$, we traverse incoming edges and add each unselected, non-stale predecessor that fits within an expanded budget of $(1 + \delta)B_t$; expansion is one hop from the fixed seed set C_0. This couples selection across related events: a supporting block can enter through its dependency link even when its own score was insufficient for the initial packing. Appendix D.2 gives the traversal order and cost bound.

Constructing the working context. RECAP renders the selected blocks chronologically between system instructions and the current turn. The selection stays fixed as tool steps extend the turn, allowing prefix-cache reuse, and execution feeds attention to the next update, completing the cycle.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks and models. We evaluate RECAP on two interactive coding benchmarks: 106 tasks from SWE-Together (Wu et al., 2026b), which involves repository-level tasks with tool use and user feedback, and 100 code instances from Lost-in-Conversation (LiC) (Laban et al., 2026), which reveal requirements across user turns, with five conversations per instance. Our backbones are Qwen3-Coder-30B-A3B-Instruct (Yang et al., 2025) and gpt-oss-20b (OpenAI et al., 2025), served with SGLang (Zheng et al., 2024). SWE-Together runs in OpenCode, and GPT-5.4 simulates all users.

Baselines. We compare against full-history restoration and six compaction baselines in three groups. *Heuristic* baselines select history without model computation: Recency retains the most recent content, and Random samples content and restores its order. *Prompt-compression* baselines, LLMLingua-2 (Pan et al., 2024) and LongLLMLingua (Jiang et al., 2024), compress historical text with a small model, the latter conditioned on the request. *Memory-compaction* baselines use the backbone itself: Codex-style summarization (OpenAI, 2025) generates a handoff summary while preserving historical user instructions, and KV eviction (Liu et al., 2026) keeps, per attention head, the cached entries most attended by the new user message and discards the rest permanently. All compactors operate on history preceding the current user turn, with system instructions and current-turn messages retained separately; KV eviction also ranks cached system instructions.

End-to-end protocol and budgets. Compaction affects subsequent responses, tool use, and user interactions, so methods develop different histories, and matching tokens at every turn would require adjusting budgets as trajectories evolve. We therefore fix each method's hyperparameters throughout execution: a nominal retention of $\beta = 0.1$ for history selection, a compression rate of 0.1 for Lingua (0.2 for LiC LLMLingua-2), and 10% of the historical cache entries per head for KV eviction. RECAP counts protected records toward its budget and retains them when they exceed it, and

Table 1: Main results on SWE-Together (reward) and Lost-in-Conversation (LiC, accuracy), scaled by 100; ±: standard deviation across runs. Δ is RECAP's score minus the row's (orange: RECAP higher; teal: lower). *Ctx.*: mean tokens per call, excluding fixed system prompts. Compaction adds, per invocation, *+Pre.* prompt and *+Dec.* completion tokens on the backbone (means) and *+Lat.* seconds (one H100; RECAP on CPU); *+Mem.*: memory it keeps in MB (GPU for compressors and KV cache, CPU for the context graph). **Bold**/underline: best/second-best scores; Appendix C gives the accounting.

Method		Qwen3-Coder-30B-A3B							gpt-oss-20b						
		Score↑	Δ	Ctx.↓	+Pre.↓	+Dec.↓	+Lat.↓	+Mem.↓	Score↑	Δ	Ctx.↓	+Pre.↓	+Dec.↓	+Lat.↓	+Mem.↓
SWE-Together															
NO COMPR.	Full	24.4±1.87	↑4.1	28,504	0	0	0	0	8.2±0.15	↑0.8	8,508	0	0	0	0
HEURISTIC	Recency	26.6±0.40	↑1.9	10,154	0	0	0	0	8.4±0.08	↑0.6	1,264	0	0	0	0
	Random	23.8±0.99	↑4.7	12,712	0	0	0	0	7.4±0.35	↑1.6	1,011	0	0	0	0
PROMPT COMPR.	LLMLingua-2	23.9±1.17	↑4.6	11,062	0	0	7.39	2,236	7.8±0.22	↑1.2	3,340	0	0	0.43	2,236
	LongLLMLingua	26.6±0.76	↑1.9	7,429	0	0	9.48	5,559	7.9±0.04	↑1.1	2,416	0	0	1.58	5,559
MEMORY COMPACT.	Summarize	**28.7**±1.26	↓0.2	5,239	71,530	397	7.67	0	8.1±0.45	↑0.9	4,543	7,664	678	2.43	0
	KV eviction	24.1±0.54	↑4.4	4,706	0	0	0.41	1,292	8.5±0.30	↑0.5	1,638	0	0	0.29	255
OURS	RECAP	28.5±0.40	—	15,344	0	0	0.02	0.02	**9.0**±0.01	—	4,043	0	0	<0.01	<0.01
Lost-in-Conversation															
NO COMPR.	Full	65.6±1.82	↑19.8	1,121	0	0	0	0	26.4±3.44	↑41.2	1,023	0	0	0	0
HEURISTIC	Recency	59.0±2.83	↑26.4	248	0	0	0	0	22.6±2.41	↑45.0	174	0	0	0	0
	Random	75.2±3.11	↑10.2	170	0	0	0	0	33.2±2.86	↑34.4	158	0	0	0	0
PROMPT COMPR.	LLMLingua-2	48.8±3.96	↑36.6	253	0	0	1.20	2,236	19.2±5.07	↑48.4	17	0	0	0.94	2,236
	LongLLMLingua	65.6±4.56	↑19.8	163	0	0	1.26	5,559	21.0±2.83	↑46.6	17	0	0	1.09	5,559
MEMORY COMPACT.	Summarize	75.4±2.70	↑10.0	372	1,500	285	1.45	0	36.8±5.12	↑30.8	602	1,971	591	2.38	0
	KV eviction	70.0±3.58	↑15.4	146	0	0	0.38	62	33.4±2.87	↑34.2	256	0	0	0.31	30
OURS	RECAP	**85.4**±1.95	—	94	0	0	<0.01	<0.01	**67.6**±3.58	—	73	0	0	<0.01	<0.01

summarization sets its own summary length. We compare task quality together with realized context usage and compaction overhead (Appendix C).

Metrics. We report reward on SWE-Together and accuracy on LiC, scaled by 100, with standard deviations across repeated runs. Context usage is the mean number of tokens per call, excluding fixed system prompts, with the same accounting scope for every method. We also report the additional backbone input and output tokens used for compression, compression latency, and the memory each method keeps for compaction (Appendices C and D.1).

5.2 MAIN RESULTS

Table 1 reports task quality, context usage, and the additional cost of compaction. On SWE-Together, RECAP at β=0.1 roughly halves the historical context of full restoration while scoring 4.1 and 0.8 points higher with Qwen3-Coder and gpt-oss, respectively. On Lost-in-Conversation, it improves accuracy over full restoration by 19.8 and 41.2 points.

On LiC, RECAP exceeds Codex-style summarization by 10.0 and 30.8 accuracy points with mean contexts of 94 and 73 tokens, compared with 372 and 602 for summarization and 1,121 and 1,023 for full restoration. Summarization also processes 1,500 and 1,971 input tokens and generates 285 and 591 output tokens per invocation, whereas RECAP selects context through graph traversal and lexical matching without an additional model call.

RECAP also keeps little state for compaction: its graph of scalar scores and dependency links occupies about 19 KB of CPU memory per SWE-Together session with Qwen3-Coder and under 3 KB elsewhere. The Lingua baselines keep 2.2 and 5.6 GB of compressor weights on the GPU, and KV eviction holds 30 MB to 1.3 GB of KV cache per session until the next user message arrives, because it needs that message to rank the cached entries.

Because RECAP selects from the full archive at each request, omitted blocks can return later: on SWE-Together with Qwen3-Coder, 70% of turn transitions revive at least one omitted block, whereas recency and KV eviction cannot recover dropped history (Appendix C.2).

5.3 INFERENCE EFFICIENCY

Memory compaction reduces context-encoding time, but its own computation also adds latency. We evaluate this trade-off on SWE-Together with one H100 80GB GPU, measuring cold time to first token (TTFT) for each model and method over 30 uncached synthetic requests at its mean retained context length. Adding mean TTFT to the serial H100 compaction estimates in Table 1 estimates the latency at a compaction boundary; RECAP uses a prepared context graph (Appendix C.1).

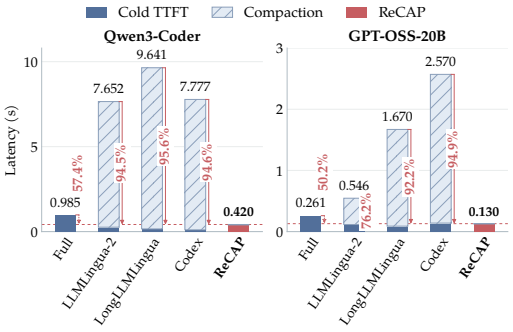

Figure 4: Estimated cold-restoration latency on SWE-Together. Bars combine TTFT and compaction overhead; arrows show reductions with RECAP. The two panels use separate scales.

RECAP reduces cold TTFT from 0.985 to 0.395 s for Qwen3-Coder and from 0.261 to 0.128 s for gpt-oss, reductions of 59.9% and 50.8% over Full, or speedups of 2.49× and 2.03×. Compaction overhead changes the comparison among methods. The Lingua baselines achieve lower TTFT than RECAP, but their compression dominates the combined cost, and summarization adds a generation step. Including compaction, their estimated latencies range from 7.65 to 9.64 s for Qwen3-Coder and from 0.55 to 2.57 s for gpt-oss, and RECAP reduces these totals by 94.5–95.6% and 76.2–94.9% (Figure 4). Reusing importance and dependencies from the persistent graph lets RECAP shorten the working context without an additional model call.

5.4 Ablation Studies

We ablate RECAP on the LiC code tasks with `Qwen3-Coder-30B-A3B` under the protocol of Table 1. Removing the attention graph discards importance scores, dependency expansion, and stale marks, leaving a selector that keeps the protected records and fills the remaining budget by identifier overlap with the request. The segmentation variants split plain assistant messages into paragraphs or lines instead of one block, keeping fenced code intact. All three variants score below RECAP (Table 2): by 8.8 points without the attention graph, and by 2.2 and 2.4 points with paragraph and line blocks. They also differ in context size: without the attention graph, the working context stays at 95 tokens on average (94 for RECAP), whereas finer blocks let fragments of earlier answers fit within the budget, raising it to 152 and 168 tokens.

Table 2: Component ablations on Lost-in-Conversation with `Qwen3-Coder-30B-A3B` (β=0.1). Δ is RECAP's accuracy minus the variant's. *Ctx.* is the mean context size in tokens, excluding the fixed system prompt, as in Table 1.

Variant	Acc.↑	Δ	Ctx.↓
RECAP	**85.4**±1.95	—	94
w/o attention graph	76.6±1.95	↑8.8	95
paragraph blocks	83.2±1.92	↑2.2	152
line blocks	83.0±2.45	↑2.4	168

5.5 Compatibility with Prefix Caching

Memory compaction targets requests that arrive after the serving cache has expired. Many user turns, however, arrive while the session's KV cache is still resident, and prefix caching already avoids re-encoding the history for them. RECAP complements prefix caching rather than replacing it: it rewrites the working context only when the cache is gone (Figure 5a). At an idle boundary, RECAP selects a compact context S_n, which the engine prefills. At a warm boundary, it skips selection: the next request is appended to the resident context, so the model sees S_n, the complete turn n, and the new request, and the engine reuses their cached prefix. Within a turn, the selected history likewise stays fixed. Graph updates stay off the critical path: attention statistics are captured during inference, and the graph is updated on the CPU in the background, overlapping the next turn's generation at warm boundaries and the idle period at idle boundaries.

We evaluate this policy on SWE-Together with gpt-oss-20b served by vLLM (Kwon et al., 2023) with automatic prefix caching. At each user boundary, a seeded draw marks the session warm with probability p, keeping its cache, or idle, discarding it and running RECAP; we run $p \in$

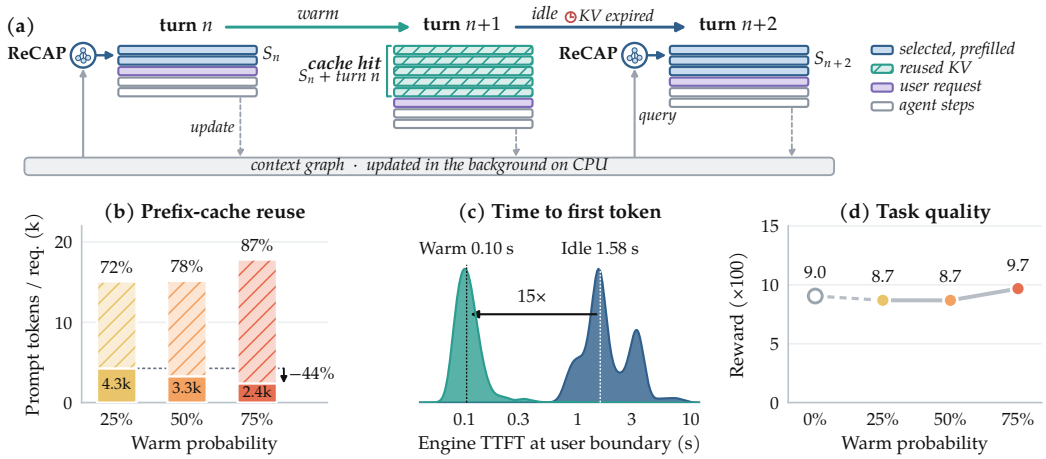

Figure 5: **ReCAP with prefix caching** (gpt-oss-20b, SWE-Together). **(a)** Idle boundaries prefill the compact context S_n; warm ones reuse the cache; the graph updates in the background. **(b)** Prompt tokens per request, cached (hatched) and uncached (solid), with hit rates. **(c)** Engine TTFT at warm and idle boundaries. **(d)** Task reward versus warm probability; hollow point: all idle (Table 1).

$\{25\%, 50\%, 75\%\}$ once each. Warm boundaries reuse over 99.9% of their prompt tokens from the cache. As p rises from 25% to 75%, the cache-hit rate grows from 72% to 87% and uncached prompt tokens per request fall by 44% (Figure 5b). The engine returns the first token in a median of 0.10 s at warm boundaries, compared with 1.58 s when an idle boundary prefills the compacted context (Figure 5c). Task reward stays between 8.7 and 9.7 across the three settings, with no significant difference from the 25% setting (Figure 5d), and is 9.0 when every boundary is idle (Table 1). Consecutive warm turns carry their appended history, so the average prompt grows from 15.1k to 17.8k tokens, but the cache serves this growth. Prefix caching thus serves turns whose cache survives, and ReCAP serves turns whose cache has expired, without undoing each other.

5.6 Attention Estimation for Black-Box Agents

ReCAP reads attention from the agent's own model, which a closed API does not expose. Because the graph stores block-level statistics, it can come from a different model that replays the served context with its own tokenizer and maps its attention back to the same blocks. We use Qwen3-0.6B, about 2–3% of the agent's parameters, as the proxy for a Qwen3-Coder-30B agent from the same family and a gpt-oss-20b agent from a different family, with all other settings as in Table 1.

Table 3: ReCAP with attention from the agent or a Qwen3-0.6B proxy (β=0.1). Scores are scaled by 100; ±: standard deviation across runs.

	LiC		SWE
Attention	gpt-oss	Qwen	Qwen
agent's own	67.6±3.58	85.4±1.95	28.5±0.40
Qwen3-0.6B proxy	68.2±3.49	85.2±2.17	29.0±0.64
none	63.4±2.61	76.6±1.95	25.4±0.22

The proxy matches the agent's own attention in all three settings (Table 3): the differences are $+0.6$ and -0.2 points on LiC with gpt-oss and Qwen, and $+0.5$ points on SWE-Together with Qwen, none statistically significant. With Qwen on LiC, removing the attention graph lowers accuracy, and the proxy retains this gain, scoring 8.6 points above the no-attention variant. A small open model can therefore supply attention for a black-box agent, including one from a different model family.

6 Conclusion

ReCAP keeps attention that inference already computes in a persistent context graph queried at each request. Without extra model calls, it cuts compaction and cold-restoration latency by about 95% versus summarization while raising LiC accuracy over full history by 19.8 and 41.2 points.

AI USE STATEMENT

We used generative AI tools primarily for assisting method implementation and polishing the paper to increase the clarity. We did not use generative AI tools to generate datasets, formulate theoretical claims, or write proofs. The authors reviewed all AI-assisted work: the writing is precise, every cited paper was manually added and checked by the authors, and the code is aligned with our proposed method. We take responsibility for the final content of this work.

ETHICS STATEMENT

This work studies memory compaction for LLM coding agents on public benchmarks. User turns are simulated by an LLM, and the study involves no human subjects or personal data. Compaction necessarily omits part of the history from a model call; RECAP keeps user instructions and the latest edits in every working context and retains the full history, so omitted content remains available to later requests and to auditing.

REPRODUCIBILITY STATEMENT

Section 4 and Algorithm 1 specify the graph update and context selection, and Appendix D gives the remaining implementation rules and all hyperparameters (Table 5). Appendix C describes the benchmarks, baselines, budgets, and cost accounting, including the summarization prompts in Appendix C.5. Appendix B details the observation studies. We will release the code and evaluation scripts.

REFERENCES

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S Kevin Zhou. Ada-kv: Optimizing kv cache eviction by adaptive budget allocation for efficient llm inference. *Advances in Neural Information Processing Systems*, 38:113152–113188, 2026.

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

Shiwei Gao, Youmin Chen, and Jiwu Shu. Fast state restoration in llm serving with hcache. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 128–143, 2025.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, 2024.

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

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W Lee, Sangdoo Yun, and Hyun Oh Song. Kvzip: Query-agnostic kv cache compression with context reconstruction. *Advances in Neural Information Processing Systems*, 38:167563–167591, 2026.

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

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation. In *International Conference on Learning Representations*, volume 2026, pp. 54738–54778, 2026.

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

Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, et al. Scbench: A kv cache-centric analysis of long-context methods. In *International Conference on Learning Representations*, volume 2025, pp. 66063–66093, 2025.

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

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

Yujian Liu, Jiabao Ji, Li An, Rohit Jain, Gungor Polatkan, Siyu Zhu, and Shiyu Chang. Practical online kv cache compaction for llm agents: An empirical study. *arXiv preprint arXiv:2608.00902*, 2026.

Haiquan Lu, Zigeng Chen, Gongfan Fang, Xinyin Ma, and Xinchao Wang. Mix-quant: Quantized prefilling, precise decoding for agentic llms. *arXiv preprint arXiv:2605.20315*, 2026.

OpenAI. Codex CLI: A lightweight coding agent that runs in your terminal. `https://github.com/openai/codex`, 2025. Version X.Y.Z. Accessed: 2026-09-25.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025. URL `https://arxiv.org/abs/2508.10925`.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, et al. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 963–981, 2024.

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

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Yating Wu, Yuhao Zhang, Sayan Ghosh, Sourya Basu, Anoop Deoras, Jun Huan, and Gaurav Gupta. Contextweaver: Selective and dependency-structured memory construction for llm agents. *arXiv preprint arXiv:2604.23069*, 2026a.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, et al. Swe-together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026b.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. *Advances in neural information processing systems*, 37:119638–119661, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhengran Zeng, Yixin Li, Rui Xie, Wei Ye, and Shikun Zhang. Attncompress: Dynamic attention-guided trajectory compression for software engineering agents. *arXiv preprint arXiv:2609.08318*, 2026.

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

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

Zijian Zhou, Ao Qu, Zhaoxuan Wu, Sunghwan Kim, Alok Prakash, Daniela Rus, Bryan Kian Hsiang Low, and Paul Liang. Mem1: Learning to synergize memory and reasoning for efficient long-horizon agents. In *International Conference on Learning Representations*, volume 2026, pp. 58413–58438, 2026.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Tracelab: Characterizing coding agent workloads for llm serving. *arXiv preprint arXiv:2606.30560*, 2026.

Adam Zweiger, Xinghong Fu, Han Guo, and Yoon Kim. Fast kv compaction via attention matching. *arXiv preprint arXiv:2602.16284*, 2026.

A FAILURE VIGNETTES UNDER SCORE-ONLY SELECTION

Two representative failures when selection uses scores alone, without protected records. *(i) Task statement evicted at cold start:* at the first post-eviction turn of a session whose instruction was to analyze a GitHub issue (analysis only, no implementation), the selector kept the agent's own 215-character preamble and dropped the 9,116-character task block. The agent then asserted it had read the issue, invented a different problem (a bug in a nonexistent `reset` command), spent some twenty steps searching for files that do not exist, and shipped an unrelated edit. *(ii) Own-edit record evicted:* the selector kept stale analysis prose describing the pre-edit code as current while dropping the tool-call record of the agent's own just-completed fix. On re-reading the file, the agent found code it did not remember writing, concluded that the issue had "already been fixed" and that no further changes were needed, and repeated that conclusion until the session ended. In both cases the lost information was recoverable from the workspace through ordinary tools—but the agent had no signal that anything needed recovering.

B ADDITIONAL OBSERVATIONS AND MEASUREMENT DETAILS

The studies in Figure 2 use Qwen3-Coder-30B-A3B-Instruct to examine three aspects of compaction: attention concentration within a session, identifier reuse across turns, and attention redistribution under controlled requests. We describe their cohorts and read-outs separately below. We obtain attention by running the model on recorded contexts through the interface in Appendix D.1.

B.1 ATTENTION CONCENTRATION AND CROSS-TURN REUSE

Multi-turn attention example. The left panel of Figure 2 shows one coding session at six measured user-turn checkpoints. Each incoming request supplies attention queries over its preceding history. We aggregate received attention over the tokens of each historical block and normalize its mass over the history. For selection, blocks are ranked by attention mass per token and greedily packed under a 10% historical-token budget. At the final checkpoint, the selected blocks contain $2{,}824$ of $28{,}309$ tokens and retain 65.01% of historical attention. Heatmap columns and the concatenated context use the same token-length scale. Color denotes per-token attention divided by the maximum within each row, displayed on a cube-root scale.

Cross-turn study. The middle panel of Figure 2 uses 64 transitions from eight coding-agent sessions. Past-attention scores aggregate attention sampled from the preceding completed turn across heads at layers $\{16, 24, 32, 40\}$ of the 48-layer model. Candidate blocks precede that completed turn. Block scores summarize received attention per token. Figure 6 also reports aggregate concentration and the full comparison of selection signals.

Identifier-coverage proxy. Target identifiers are extracted from assistant and tool-result text, together with tool-call names and arguments, recorded before the following user message. The incoming user request itself is excluded from this evidence. The extractor retains code-like identifiers of at least four characters and dotted paths under a short stop list. An identifier occurring in df of the n candidate blocks receives weight $\log(n/\mathrm{df})$. Only target identifiers appearing in the candidate history contribute to the denominator. Coverage is the fraction of their total weight present in the selected block set. Selectors rank blocks and pack them greedily under a fraction of the historical-text length; the evaluation code counts characters. The future-use oracle ranks blocks by overlap with the future target identifiers and uses the same packing rule. The combined selector adds a query-identifier score to past attention; the standalone query-identifier and embedding comparisons appear in Figure 6.

B.2 CONTROLLED QUERY STUDY

Fixed histories and requests. We select one checkpoint from each of 32 distinct coding tasks, with 16 histories from RECAP traces and 16 from recency-selection traces. Histories range from 8k to 32k rendered tokens. Each history contains explicit references to two target files, chosen before attention measurement. Historical block contents and identities remain fixed across request conditions.

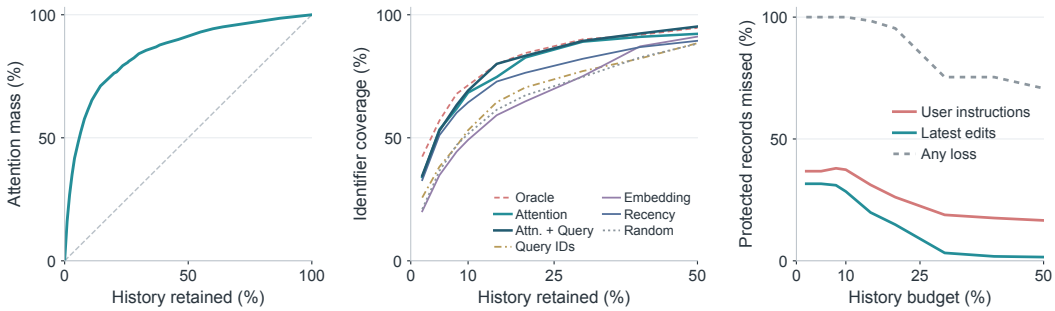

Figure 6: **Additional historical-context observations. Left:** Aggregate attention concentration under a retained-history budget; the diagonal denotes uniform attention per unit of history. **Middle:** The full comparison of selection signals, including query identifiers and query embeddings. **Right:** User-instruction and latest-edit records omitted by attention-ranked selection, together with the fraction of transitions losing any protected record. The selector receives enough budget to fit the complete protected set.

For each target, we construct two short requests that ask the agent to continue or return to that file. This yields four requests per history. We repeat the comparison with longer, framed requests that explicitly ask the agent to recall earlier implementation facts, constraints, and tool observations. The two template families therefore provide eight measurements per history, or 256 in total. Only the incoming request changes between measurements.

Attention redistribution. We normalize attention mass over historical blocks after excluding the initial task instruction. For two resulting distributions p and q, redistribution is their total-variation distance,

$$D_{\mathrm{TV}}(p, q) = \frac{1}{2} \sum_{v} |p(v) - q(v)|. \tag{4}$$

Within each template family, a history contributes one paraphrase value, averaging the two same-target comparisons, and one goal-change value, averaging the four cross-target comparisons. Statistical analysis treats histories as paired observations.

Results and visualization. For short requests, mean redistribution is 5.33% under paraphrasing and 10.13% under a target change. The paired difference is 4.81 percentage points (task-bootstrap 95% CI $[3.41, 6.25]$) and is positive in $28/32$ histories. Framed requests give 3.82% and 8.46%, respectively, with a paired difference of 4.64 points ($[3.64, 5.69]$), positive in $30/32$ histories. Confidence intervals use $10{,}000$ task-bootstrap resamples.

B.3 Preserving instructions and recent modifications

User instructions specify the task, and the latest edit or write record for each modified file records the agent's most recent changes. We protect these blocks as the session's skeleton. In the cross-turn corpus, this set occupies a median 10.3% of historical text. To measure whether attention ranking retains it, we give the selector budget $\max(\beta\, c(V_t), c(P_t))$, so the complete skeleton fits. At $\beta = 0.1$, score-only selection still omits approximately 37% of user messages and 28% of latest-edit records, and every transition loses at least one protected block (Figure 6, right). These measurements motivate explicit protection during packing. Appendix A gives two corresponding failure examples.

C Evaluation protocol and cost accounting

Tasks and budgets. Table 1 reports SWE-Together results on 106 tasks and Lost-in-Conversation results on 100 instances, with five conversations scheduled per instance. For LiC, the kth conversation of every task forms repeat k. We report the mean and sample standard deviation ($n - 1$ denominator) of the five repeat accuracies. Each repeat has 100 tasks, with missing or unscored conversations counted as unsuccessful. SWE-Together standard deviations summarize two replicates.

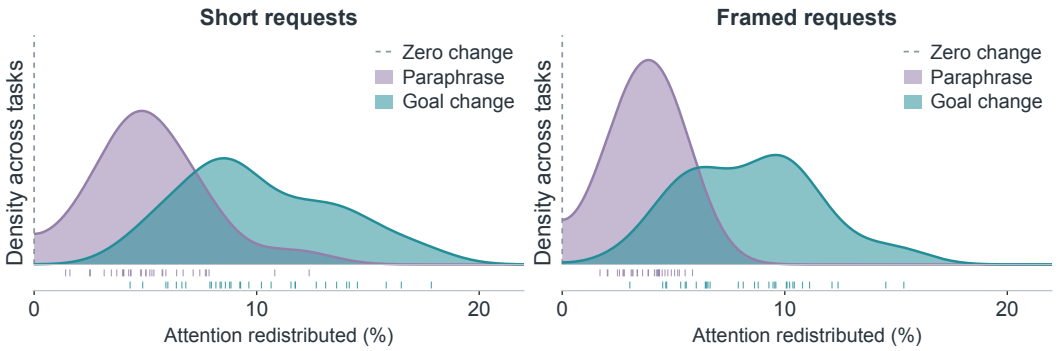

Figure 7: **Query effects across request templates.** Paraphrases of the same target and requests about different targets are compared on the same 32 histories. Longer framed requests explicitly ask the agent to recall earlier facts and constraints. Both template families show greater attention redistribution under a target change. Curves share a smoothing bandwidth; bottom ticks identify the paired task-level observations, and the dashed line marks zero redistribution.

For history selection, the budget is computed from each method's own accumulated history. Lingua receives its compression rate through the compressor's own interface. LiC Recency and Random use line-level splitting of assistant text, while RECAP uses message-level blocks. RECAP's protected blocks consume the nominal allowance before other blocks are added, and dependency expansion allows up to 10% additional budget, subject to the bound in Eq. 6. These rules and the independently generated histories determine realized usage. The Full row summarizes the full-restoration policy's own trajectories; per-request compression ratios use each compactor's own history before and after compression.

Token and latency accounting. For SWE-Together, *Ctx.* reports the mean retained historical content per request. Every method excludes the original system instructions, tool definitions, and current turn from this quantity. For LiC, *Ctx.* subtracts the fixed rendered system prefix from each complete input: 73 tokens for Qwen3-Coder and 135 for gpt-oss, the same for all methods. The remaining count includes retained history, the current user turn, message framing, and any omission notice. LiC means cover all completed assistant calls, including the first.

Compressor prompt and completion counts use serving-engine usage; a reused summary incurs no new compression call. LLMLingua uses a separate small model, whose computation appears in compression latency. Compression latencies of the baselines are measured on one H100, one request at a time (Appendix C.1). For LiC, latency is the mean over 30 sampled compression invocations after 10 warm-up invocations, timed until the full compressed output is returned. RECAP's selection runs on the CPU without a model call; its *+Lat.* is the mean selection time on one CPU core (Appendix C.4).

KV eviction. At the start of each user turn after the first, it scores every resident historical entry by its attention from the new user message and keeps $\lceil 0.1N \rceil$ entries per head, where N counts all historical tokens, including those evicted earlier. The current input is kept in full, and evicted entries are never recomputed. After eviction, the engine recomputes the final input token against the compacted cache; we omit this single token from *+Pre.* Its *+Lat.* is the mean over 30 compaction events sampled from the quality runs and replayed on one H100 with caches of the recorded sizes, one event at a time after 10 warm-up events. Each measurement covers scoring, selection, the copy into compacted tensors, and the one-token recomputation. Scoring, selection, and copying take 5–32 ms on average, and the one-token recomputation takes the rest. For *Ctx.*, we count the historical entries retained per head, adding the current input on LiC. These entries may include cached system instructions, so the value is an upper bound on retained history under the other methods' accounting.

Memory accounting. *+Mem.* counts memory that a method keeps for compaction, beyond the backbone weights and the KV cache of the working context. For the Lingua baselines, it is the compressor weights: 2.24 GB for LLMLingua-2 (559M parameters in FP32) and 5.56 GB for LongLLMLingua (Phi-2, 2.78B parameters in FP16), excluding activations. For KV eviction, it is the mean KV cache a session holds when the next user message arrives, before eviction. For RE-

CAP, it is the context graph at the end of each session, averaged over sessions, stored with 12 bytes per node (identifier, score, and flags) and 12 bytes per edge (two identifiers and a weight). The full history itself is kept by the agent harness for every method and is not counted. We count at most 6.5 edges per block, the rate measured on gpt-oss SWE-Together sessions. Because an edge requires more than θ=0.02 of the dependent block's attention mass, each block has at most 49 incoming edges; even this worst case stays below 0.8 MB per session. Full restoration, Recency, Random, and summarization keep no additional state.

C.1 Cold-TTFT measurement protocol

We measure TTFT on one NVIDIA H100 80GB HBM3 GPU with SGLang 0.5.17, PyTorch 2.11.0 with CUDA 13.0, Transformers 5.12.1, and FA3 attention. Qwen3-Coder uses native BF16 weights and gpt-oss uses native MXFP4 weights. Each condition contains 30 timed synthetic requests after warm-up. No request reuses cached tokens.

Synthetic input lengths match the SWE-Together mean historical-context sizes in Table 1. In the order Full, LLMLingua-2, LongLLMLingua, Summarize (Codex), and RECAP, these lengths are 28,504, 11,062, 7,429, 5,239, and 15,344 tokens for Qwen3-Coder, and 8,508, 3,340, 2,416, 4,543, and 4,043 tokens for gpt-oss. These requests measure cold TTFT at historical-context lengths, excluding the separately supplied system prompt, tool definitions, and current turn.

SWE-Together compaction costs are measured on the same GPU, one request at a time: the Lingua compressor shares the GPU with the serving engine, and summarization is a complete additional model call, including prefill and generation, on an idle engine with its cache cleared.

For each condition, Figure 4 adds the mean measured TTFT to the compaction latency in the main table, estimating a request that performs compaction before cold restoration. The reductions shown by the arrows are $100(1 - L_{\text{RECAP}}/L_{\text{baseline}})$, where L is this combined estimate.

C.2 Compaction pattern analysis

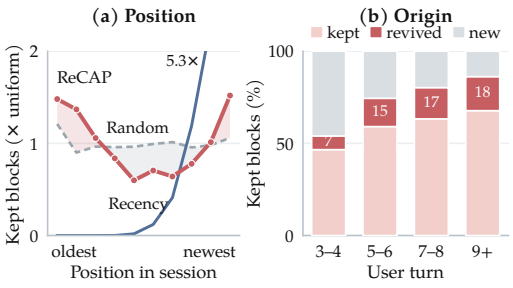

Figure 8: History retained by RECAP on SWE-Together with Qwen3-Coder. **(a)** Retained blocks in each tenth of the session, relative to a uniform spread; counts include protected records. **(b)** Origin of the blocks retained at each user turn: kept from the previous working context, revived after being omitted from it, or created since.

We examine which history RECAP carries into the working context, using the compaction records of the Qwen3-Coder SWE-Together runs in Table 1. Figure 8a divides each session into ten equal spans by block position and compares each span's share of retained blocks with a uniform spread. Recency concentrates on the latest span, at 5.3× its uniform share, and retains almost nothing from the oldest 40% of the session, while Random stays close to uniform. RECAP draws from both ends: the oldest and newest spans each receive about 1.5× their uniform share, and the middle falls to 0.6×. The oldest 30% of the session supplies 39% of the blocks RECAP retains, compared with 31% for random selection and none for recency.

Because RECAP selects from the full archive at each request, blocks omitted from one working context can return in a later one (Figure 8b). About 60% of retained blocks carry over from the previous working context, and the share of revived blocks grows from 7% in turns 3–4 to 18% from turn 9 onward; 70% of turn transitions revive at least one omitted block. Recency never revives a block once its window has passed it, and KV compaction discards evicted entries permanently, so a later request cannot recover them without re-encoding the full history (Liu et al., 2026; Zweiger et al., 2026).

C.3 EXAMPLE OF A REVIVED BLOCK

The example below comes from a SWE-Together session with Qwen3-Coder. At the first user turn, the agent checked the authentication-event routes and handlers with `grep`. The resulting block is not protected and had no recorded importance, so it stayed outside the working context for the next seven user turns. At turn 8, the user asks the agent to verify these handlers and names six of the functions that appear in the block. These rare identifiers give the block the highest query-overlap score among the 98 history blocks, and RECAP selects it back into the working context under the same 10% budget. Because selection runs over the full history at each request, content omitted for several turns can return once a request refers to it.

History block from user turn 0 (omitted at turns 1–7, selected at turn 8)

Let me verify that all the necessary endpoints have been added by running a quick check:

```
$ cd /workspace/hyperswitch && grep -n "get_.*auth_event"
crates/router/src/analytics.rs
318:  .route(web::post().to(get_org_auth_event_metrics)),
409:  .route(web::post().to(get_profile_auth_event_metrics)),
[...]
1121:  pub async fn get_org_auth_event_metrics(
1166:  pub async fn get_profile_auth_event_metrics(
[...]
3087:  pub async fn get_profile_auth_event_sankey(
```

User request at turn 8

wait, this still looks incomplete/wrong — your final diff only shows route additions, but earlier you said you added new handler fns and core auth changes too. pls verify the new handlers actually exist and are wired, [...] plus a quick grep for `get_org_auth_event_metrics|get_profile_auth_event_metrics|get_org_auth_event_sankey|get_profile_auth_event_sankey|get_org_auth_events_filters|get_profile_auth_events_filters`, and fix/commit if anything's missing.

C.4 SELECTION COST

We time RECAP's context selection, from block segmentation through scoring, packing, and dependency expansion, on one CPU core (AMD EPYC 7453) for requests of the sizes in Table 1. Table 4 reports the per-request time. Selection takes about a millisecond on LiC and on gpt-oss SWE-Together sessions and a median of 20 ms on the longer Qwen3-Coder SWE-Together sessions, well below the compression latency of every baseline. Almost all of this time extracts identifiers from the history text, so it grows linearly with history length; because blocks do not change once written, their identifier sets can be computed once and reused. Updating the graph from attention statistics, including the moving average, edge merging, and the supersession test, takes a median of 3.7 ms per turn with Qwen3-Coder and 0.35 ms with gpt-oss on SWE-Together.

Table 4: Per-request selection time of RECAP on one CPU core.

Benchmark	Backbone	Blocks (median)	Median (ms)	P90 (ms)	Max (ms)
SWE-Together	Qwen3-Coder	132	20.2	45.2	112.6
SWE-Together	gpt-oss	18	1.4	4.4	20.3
LiC	Qwen3-Coder	6	0.30	0.80	1.69
LiC	gpt-oss	6	0.26	0.65	1.51

C.5 SUMMARIZATION BASELINE PROMPTS

The Codex-style summarization baseline uses the compaction prompts of the Codex CLI (OpenAI, 2025) verbatim. The instruction below is appended to the history as a final user message, and the returned summary follows the prefix in the compacted context.

Compaction instruction (final user message)

You are performing a CONTEXT CHECKPOINT COMPACTION. Create a handoff summary for another LLM that will resume the task.

Include:

- Current progress and key decisions made
- Important context, constraints, or user preferences
- What remains to be done (clear next steps)
- Any critical data, examples, or references needed to continue

Be concise, structured, and focused on helping the next LLM seamlessly continue the work.

Summary prefix (precedes the summary in the compacted context)

Another language model started to solve this problem and produced a summary of its thinking process. You also have access to the state of the tools that were used by that language model. Use this to build on the work that has already been done and avoid duplicating work. Here is the summary produced by the other language model, use the information in this summary to assist with your own analysis:

D Implementation details and hyperparameters

D.1 Attention access and serving realization

Attention interface. Native integration obtains sampled attention rows from retained query vectors, or their source hidden states, and resident keys. Importance extraction samples current-turn positions across a layer band; dependency extraction samples positions within each block at one selected layer. The sampled rows are reduced to block statistics at the turn boundary. Subsequent context selection reads these stored statistics and the graph's adjacency information.

Request construction and context limits. The proxy rewrites the request's message array with the selected blocks. When history is omitted, a system notice identifies the omission and states that the workspace remains available for retrieving further details. Selection uses the same history boundary, graph version, request, and random seed throughout a user turn. The engine enforces a 360k-character request cap by dropping the oldest selected blocks when the full request exceeds the limit. This guardrail takes precedence over protected-record retention and prefix stability.

D.2 Graph construction and selection

Graph state and initialization. The persistent state contains the graph $G_t = (V_t, E_t)$ with node scores u_t and edge weights w_t, and the node annotations $\mathcal{O}_t$ and $\mathcal{S}_t$. Node scores start at zero; edges and annotation sets start empty. A graph update registers newly observed blocks and revises scores only for R_t. Edges, stale marks, and observation membership accumulate across updates. The state requires $O(|V_t| + |E_t|)$ scalar metadata in addition to the history text. The protected set P_t is recomputed from message roles and edit/write records for each new turn.

Dependency and supersession rules. For sampled positions $W_t(b)$ within block b, the strength of an edge from a to b is

$$d_t(a, b) = \frac{1}{|W_t(b)|} \sum_{i \in W_t(b)} \sum_{j \in \mathrm{span}_t(a)} \bar{a}_t^{\mathrm{dep}}(i \to j), \tag{5}$$

where $\bar{a}_t^{\mathrm{dep}}$ averages heads at the dependency layer. We add an edge when a precedes b and $d_t(a, b) > \theta$, retaining its maximum measured strength across updates. Expansion uses the retained edge's presence. Supersession compares observed pairs in chronological order. An earlier block a is marked stale when its identifier-set Jaccard similarity with a later block b exceeds 0.5 and $\bar{\rho}_t(b) > 2\bar{\rho}_t(a) + 0.05\bar{\rho}_t^{\max}$. The identifiers for this test are extracted from truncated block text, as specified below. Stale marks persist; selection applies their score penalty and skips stale predecessors during expansion.

Packing order and budget. Algorithm 1 summarizes the two graph operations. Ranked packing breaks score ties by chronological order. Expansion visits the fixed seed set C_0 in chronological or-

Algorithm 1 Updating and querying the persistent context graph.

procedure UPDATEGRAPH(R_t, sampled attention, G, $\mathcal{O}$, $\mathcal{S}$)
  register newly observed blocks in V with zero initial importance
  compute $\bar{\rho}_t(v)$ and update $u(v)$ for $v \in R_t$ using Eq. 1
  $\mathcal{O} \leftarrow \mathcal{O} \cup R_t$
  add edges (a, b) with $a < b$ and $d_t(a, b) > \theta$ using Eq. 5
  set each edge weight $w(a, b)$ to its maximum observed $d_t(a, b)$
  mark $a \in \mathcal{S}$ when a later b passes the supersession test
procedure SELECTCONTEXT(H_t, q_{t+1}, G_t, $\mathcal{O}_t$, $\mathcal{S}_t$)
  register any remaining blocks in H_t; derive P_t and $B_t = \lfloor \beta c(V_t) \rfloor$
  score nodes using Eq. 3; $C \leftarrow P_t$
  for $v \in V_t \setminus P_t$ in descending score order:
    if $c(C) + c(v) \le B_t$: $C \leftarrow C \cup \{v\}$
  $C_0 \leftarrow C$ ▷ fixed seeds for one-hop expansion
  for $b \in C_0$ in chronological order, each predecessor a in stored order:
    if $a \notin C \cup \mathcal{S}_t$ and $c(C) + c(a) \le (1 + \delta)B_t$: $C \leftarrow C \cup \{a\}$
  return selected blocks in chronological order

der and each seed's edges in their stored order. Newly added predecessors are not used as expansion seeds. Before the serving cap is applied, the selected set satisfies

$$P_t \subseteq C, \qquad c(C) \le \max\{c(P_t), (1 + \delta)B_t\}. \tag{6}$$

If protected records already exceed $(1+\delta)B_t$, the packing and expansion tests add no further blocks. System instructions, the omission notice, and the current turn are outside this historical-block budget.

D.3 HYPERPARAMETERS AND PREPROCESSING

Table 5 lists the graph-update and compaction settings. Character cost counts message content and serialized tool-call objects. Identifier extraction for Eq. 2 uses message content and matches code-like tokens of length ≥ 4, including dotted file paths, under a short stop list; document frequencies are computed within the session. Supersession compares identifier sets from up to 4,000 characters of block content and at most 400 matches. Received attention is clipped at its 99.9th percentile before block averaging. A zero maximum in importance normalization contributes a zero observation; standardization returns zero for a constant channel. Before any attention is observed, query overlap and the exploration bonus determine the unprotected ranking.

Table 5: RECAP hyperparameters. Selection and update thresholds are shared across tasks and replicates; attention layers depend on the model architecture.

Symbol	Meaning	Value
β	nominal history-character retention fraction	0.10
λ	weight of the lexical channel in Eq. 3	0.5
α	importance EMA rate (§4.2)	0.6
ϵ	exploration bonus scale for unobserved blocks	0.1
κ	stale penalty (in standardized-score units)	2.0
δ	allowance for one-hop dependency expansion	0.1
L	Qwen feedback layers (48 layers, zero-indexed)	$\{16, 24, 32, 40\}$
L	gpt-oss feedback layers (24 layers, zero-indexed)	$\{11, 15, 19, 23\}$
$\lvert W_t \rvert$	sampled current-turn positions per feedback pass	≤ 640
—	sampled positions per block for edge extraction	≤ 96
θ	attention-mass threshold for a dependency edge	0.02
—	supersede identifier-Jaccard threshold	0.5
—	supersede attention dominance	$\bar{\rho}_t(b) > 2\,\bar{\rho}_t(a) + 0.05\,\bar{\rho}_t^{\max}$
—	received-attention tail clip	99.9th percentile