PROGRESSIVE HIERARCHICAL CONTEXT COMPACTION FOR LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon LLM agents accumulate increasingly large interaction histories, making context management essential for both inference efficiency and task performance. An important challenge in context compaction is how to maintain and progressively update historical information at different levels of abstraction as an agent executes. We introduce a progressive hierarchical context-compaction framework that maintains an agent's working context as an ordered sequence of blocks at three abstraction levels: trajectory, subtask, and state. At each compaction checkpoint, an LLM compactor examines the current working context and chooses an abstraction level. The selected level determines how much of the recent history is rewritten, producing an updated working context. Repeated compaction progressively produces a hierarchical representation in which different portions of interaction history are maintained at different abstraction levels. The framework further supports a plug-and-play compactor that can be trained independently of the task-performing agent. Experiments on BrowseComp-Plus and NL2Repo-Bench demonstrate substantial improvements in task performance and the context processed per task-model call over existing compaction methods. Training a smaller compactor using decisions generated by a stronger teacher yields further performance improvements.

1 INTRODUCTION

Large language model (LLM) agents are being deployed on long-horizon tasks that require interaction with tools and environments. As an agent progresses through such tasks, its working context continuously accumulates, resulting in extremely long context. Processing long contexts not only substantially raises inference cost, but retaining noisy context can also dilute the model's attention and degrade task performance (Liu et al., 2023). Effectively managing an agent's working context is thus important for both the capability and efficiency of LLM agents.

Cognitive science provides a useful perspective on how past experience can be organized. Prior work on event cognition suggests that humans do not represent ongoing experience at a single level of abstraction; rather, the same activity can be understood at multiple hierarchical levels, from fine-grained actions to higher-level goal-directed events (Zacks & Tversky, 2001; Kurby & Zacks, 2008). A similar spectrum of representations appears in context compaction for LLM agents, where working context may be maintained as raw context, workflow and subtask summaries (Sun et al., 2025; Wang et al., 2024), or task states and long-term memory (Zhou et al., 2025; Wu et al., 2026). These approaches represent historical information at different levels of detail, but typically organize compacted history around a particular representational form. This highlights an important challenge in context compaction: how to maintain and progressively update historical information at different levels of abstraction as an agent executes.

To address this challenge, we introduce HICONTEXT, a progressive **HI**erarchical **CONTEXT** compaction framework. We maintain the agent's working context as an ordered sequence of history blocks, including previously compressed blocks and newly accumulated interactions. At each compaction checkpoint, an LLM compactor examines the current working context and chooses one of three abstraction levels: trajectory, subtask, or state. The chosen level determines, through a fixed cutoff rule, how much of the most recent history is rewritten. The selected contiguous ending portion is replaced by one new block at that level, while earlier blocks remain unchanged (Figure 1,

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.

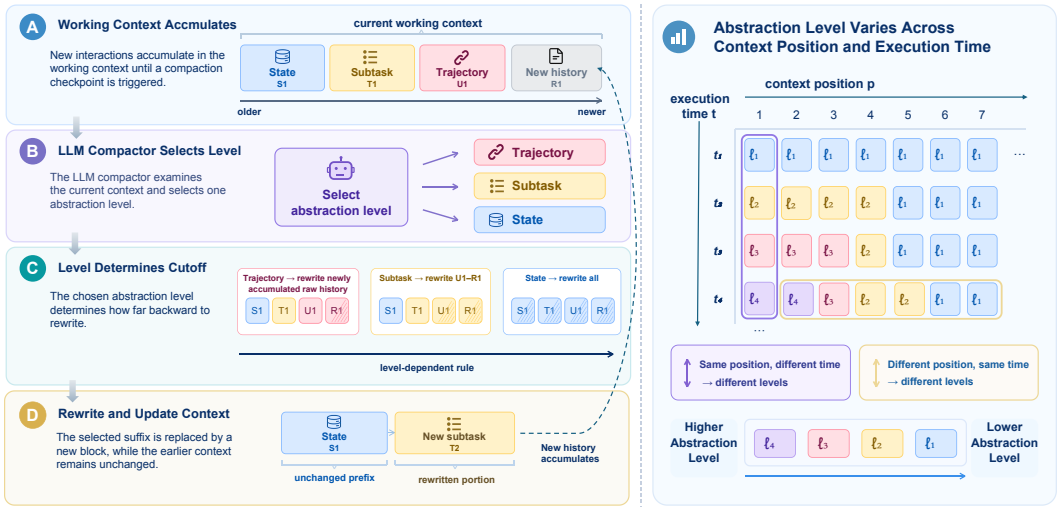


Figure 1: Overview of progressive hierarchical context compaction. **Left**: (A) The current working context is an ordered sequence of blocks, some of which may already be compacted. (B) At each checkpoint, an LLM compactor examines this context and selects a target abstraction level. (C) A deterministic, level-dependent rule sets the cutoff for the contiguous ending portion to be rewritten. (D) Rewriting replaces that portion with one block at the chosen level and leaves the earlier context unchanged. **Right**: Repeated application yields a working context with multiple abstraction levels and progressively re-abstracts them to higher levels as execution proceeds.

left). Repeating this operation progressively updates the working context and produces a hierarchical structure as execution proceeds: different portions of interaction history can coexist at different levels of abstraction, while previously compacted history can be progressively re-abstracted at higher levels as execution proceeds. (Figure 1, right).

Our key contribution is a progressive context-compaction mechanism that combines LLM-based abstraction-level selection with deterministic cutoff rules, enabling an agent to maintain and update its interaction history at multiple levels of abstraction. By organizing fine-grained trajectory information, intermediate subtask summaries, and higher-level task states within the same abstraction hierarchy, HiContext provides a unified view of context representations that are otherwise often treated separately. The LLM compactor implementing this mechanism is external to the task-performing agent, making HiContext plug-and-play and allowing the compactor to be trained independently of the task agent.

We evaluate HiContext on long-horizon web-browsing and repository-level coding tasks. Across both domains, HiContext improves task performance while substantially reducing the context processed per task-model call. Figure 2 further compares the performance–cost trade-off when accounting for both the task-performing agent and the compactor. Fixed-level ablations in the same figure show that trajectory-, subtask-, and state-level compaction all underperform the full hierarchical mechanism, indicating that the gains cannot be attributed to any single abstraction level. We further analyze the abstraction-level selection behavior over the course of agent execution. Finally, supervised fine-tuning on samples generated by a stronger teacher compactor raises the BrowseComp-Plus score from 65.3 to 71.9, while bringing the student's selection patterns closer to those of the teacher.

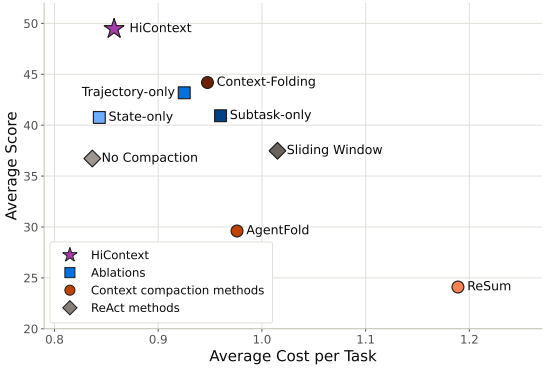


Figure 2: Performance–cost averaged over BrowseComp+ and NL2Repo. Cost includes the task-performing model and the compactor, calculated using their respective API prices.

Our contributions are threefold:

1. We introduce HICONTEXT, a progressive hierarchical context compaction mechanism that allows different portions of an agent's interaction history to be maintained and repeatedly rewritten at different abstraction levels.

2. We decouple the compactor from the task agent and train HICONTEXT-9B on outcome-filtered teacher demonstrations.

3. We show improved performance and efficiency on long-horizon search and coding tasks, and analyze how history moves through the hierarchy.

2 RELATED WORK

2.1 CONTEXT COMPACTION FOR AGENTS

Long-horizon agents increasingly require explicit mechanisms to control their growing working context, as ReAct-style agents (Yao et al., 2023) append every step to it. MemGPT (Packer et al., 2024) and CoALA (Sumers et al., 2024) separate a bounded working memory from external storage. Early context-compaction approaches primarily rely on periodic summarization or trajectory folding, such as ReSum (Wu et al., 2026) and Context-Folding (Sun et al., 2025). MEM1 (Zhou et al., 2025) maintains a compact internal state, ACON (Kang et al., 2026) optimizes and distills compression guidelines, and HiAgent (Zhang et al., 2026) summarizes history around subgoals. AgentFold (Ye et al., 2025) allows agents to proactively choose the scope of context consolidation, while LongSeeker (Lu et al., 2026) exposes different context maintenance operations as different tools for long-horizon search agents. PRO-LONG (Fox et al., 2026) further combines autonomous context editing with programmatic tools. These approaches substantially expand when, where, and how agents can manage their context. However, their primary control variables concern the timing, scope, or content of memory operations, and compacted history is typically kept in a single form. HICONTEXT instead focuses on the **abstraction levels** of the retained history, maintaining an ordered hierarchy of forms that can be re-abstracted over time. Prompt-compression methods such as LLMLingua (Jiang et al., 2023), gist tokens (Mu et al., 2024), AutoCompressors (Chevalier et al., 2023), and ICAE (Ge et al., 2024) compress inputs at the token or latent level, whereas HICONTEXT compresses semantic units of agent execution into natural language.

2.2 HIERARCHICAL SEMANTIC MEMORY

Our formulation is motivated by cognitive-science accounts of hierarchical event representation. Human observers segment continuous activity simultaneously at multiple temporal scales, with fine-grained actions grouped into larger goal-directed events (Kurby & Zacks, 2008). Such segmentation is influenced not only by low-level perceptual structure but also by higher-level goals and conceptual information (Zacks & Tversky, 2001), and event representations are updated at perceived event boundaries (Zacks et al., 2007).

Hierarchical summaries have also been used to process long documents, as in RAPTOR (Sarthi et al., 2024), MemWalker (Chen et al., 2023), and ReadAgent (Lee et al., 2024), and Generative Agents (Park et al., 2023) synthesize higher-level reflections from observations. Hierarchical organization has also been explored in LLM-agent memory systems. H-MEM (Sun & Zeng, 2025) organizes long-term memory into multiple layers according to semantic abstraction and performs coarse-to-fine retrieval through hierarchical indices. HiGMem (Cao et al., 2026) similarly uses high-level event summaries to guide retrieval of lower-level dialogue turns (see also HiMem (Zhang et al., 2026)), while PlugMem (Yang et al., 2026) transforms episodic interaction histories into a hierarchical knowledge-centric memory graph for efficient retrieval. These methods organize static inputs or stored memory to improve retrieval. HICONTEXT instead organizes the agent's **active working context**, which the agent conditions on at every step and which is rewritten as execution proceeds.

3 METHODOLOGY

3.1 OVERVIEW

We consider an agent that interacts with an environment through a sequence of model generations, tool calls, and observations. As execution proceeds, these interactions continuously accumulate in the agent's working context. Inspired by hierarchical representations of experience in cognitive science (Zacks & Tversky, 2001), we posit that the appropriate abstraction level of interaction history is not fixed. At a given point in execution, different portions of the working context may be represented at different abstraction levels; as execution progresses, the same historical information may be progressively re-abstracted at higher levels. Conceptually, we view the appropriate abstraction level as depending on both context position p and execution time t: $\ell^*(p, t) \in \mathcal{L}$, where $\mathcal{L}$ is the set of abstraction levels $\mathcal{L} = \{\ell_{\text{raw}}, \ell_1, \ldots, \ell_M\}$. ℓ_{raw} denotes the original interaction history without abstraction, and higher levels correspond to increasingly abstract semantic representations. We do not specify ℓ^* explicitly. Instead, the abstraction level is selected from the current working context at each compaction checkpoint, and we analyze the resulting organization across context position and execution time in Section 4.4.1.

At the k-th compaction checkpoint, the working context is represented as an ordered sequence of context blocks

$$C_k = [c_{k,1}, c_{k,2}, \ldots, c_{k,m_k}],$$

where each block $c_{k,j} = (x_{k,j}, \ell_{k,j})$ contains textual content $x_{k,j}$ together with its current abstraction level $\ell_{k,j} \in \mathcal{L}$. Newly accumulated interaction history enters the working context at the raw level.

HICONTEXT maintains this hierarchy through a dedicated context compactor that operates alongside the task agent. The task agent follows its original prompt, tools, action space, and interaction loop, while the compactor is invoked after every L newly accumulated raw-context tokens. At each checkpoint, the compactor observes the current working context and selects a target abstraction level while generating a representation at that level: $(x_{\text{new}}, \ell_{\text{new}}) = f_\theta(C_k)$. The working context is then updated as

$$C_{k+1} = \mathcal{U}(C_k, x_{\text{new}}, \ell_{\text{new}}),$$

where $\mathcal{U}$ is a deterministic, level-dependent rule that determines the working context to rewrite.

Because previously compacted blocks remain eligible for subsequent rewriting, repeated application of this operation produces a working context in which different portions of history coexist at different abstraction levels and the same historical information can be progressively re-abstracted as execution proceeds. In the following section, we instantiate the abstraction hierarchy and its corresponding cutoff rules used in our implementation.

3.2 INSTANTIATING THE ABSTRACTION HIERARCHY

HICONTEXT is not tied to a particular set or number of abstraction levels. More generally, it provides a unified view in which context representations are organized within a common, ordered abstraction hierarchy. Different applications may therefore instantiate this hierarchy with different abstraction levels according to their context-management needs. In this work, we instantiate the hierarchy with three representative levels: trajectory, subtask, and state, spanning fine-grained execution, coherent units of work, and global task progress. Each abstraction level is further associated with a deterministic cutoff rule that specifies the suffix of the working context to be rewritten when that level is selected (Figure 3).

3.2.1 ABSTRACTION LEVELS

We instantiate the hierarchy as $\ell_{\text{raw}} \prec \ell_{\text{traj}} \prec \ell_{\text{subtask}} \prec \ell_{\text{state}}$.

Trajectory level. At the finest abstraction level, the representation preserves the step-by-step structure of agent execution. Given a history segment $(a_i, o_i, \ldots, a_t, o_t)$, trajectory-level compaction produces $(\bar{a}_i, \bar{o}_i, \ldots, \bar{a}_t, \bar{o}_t)$, where each action and observation is represented concisely. This level captures how the agent proceeded while removing low-level redundancy.

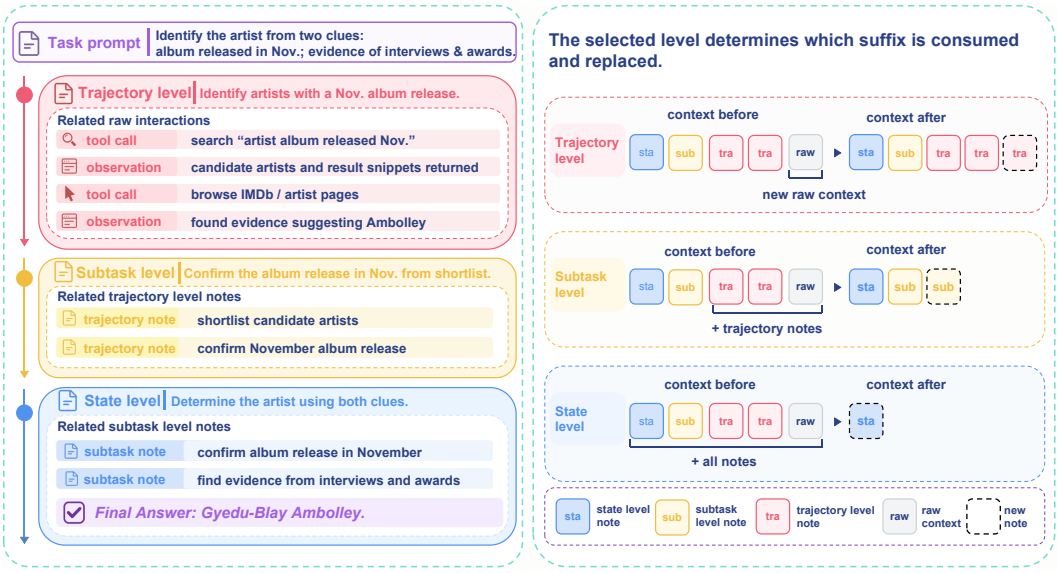


Figure 3: Instantiation of the abstraction hierarchy in HiContext. **Left:** Interaction history represented at trajectory, subtask, and state levels of abstraction. **Right:** The deterministic, level-dependent cutoff rules that determine the suffix rewritten at each level.

Subtask level. At the intermediate abstraction level, a sequence of interactions is represented as a coherent unit of goal-directed work. We abstract a history segment into an action–observation pair $(a_{i:t}, o_{i:t})$, where $a_{i:t}$ describes the objective being pursued and $o_{i:t}$ summarizes the resulting outcome. Internal execution details are omitted while the completed unit of work is retained.

State level. At the highest abstraction level in our implementation, the interaction history is represented in terms of the current task state s_t. s_t summarizes accumulated information relevant to subsequent decisions, including overall task progress, established results, and unresolved objectives.

The three levels provide increasingly abstract representations of the same underlying interaction history:

$$\underbrace{(a_i, o_i, \ldots, a_t, o_t)}_{\text{raw}} \longrightarrow \underbrace{(\bar{a}_i, \bar{o}_i, \ldots, \bar{a}_t, \bar{o}_t)}_{\text{trajectory}} \longrightarrow \underbrace{(a_{i:t}, o_{i:t})}_{\text{subtask}} \longrightarrow \underbrace{s_t}_{\text{state}}.$$

We instantiate the hierarchy as $\ell_{\text{raw}} \prec \ell_{\text{traj}} \prec \ell_{\text{subtask}} \prec \ell_{\text{state}}$.

Level selection. At each compaction checkpoint, the LLM compactor examines the current working context and selects one of the three abstraction levels above. The abstraction level is selected according to the semantic scope of the progress achieved at the current checkpoint. Intuitively, different levels correspond to increasingly larger units of completed work: from local execution steps, to a coherent subtask, to task-level progress. The compactor examines the current working context and determines which level best characterizes the newly established progress. The selection criteria are domain-specific and specified in the compactor prompt (Appendix A.3).

3.2.2 Level-Dependent Cutoff Rules

Once an abstraction level is selected, a deterministic, level-dependent cutoff rule determines the portion of the working context to be rewritten. Let $r_k(\ell)$ denote the first context block included in this portion when the selected abstraction level is ℓ. For the three levels used in our implementation, we define

$$r_k(\ell) = \begin{cases} \min\{j : \ell_{k,j} = \ell_{\text{raw}}\}, & \ell = \ell_{\text{traj}}, \\ \min\{j : \ell_{k,j} \preceq \ell_{\text{traj}}\}, & \ell = \ell_{\text{subtask}}, \\ 1, & \ell = \ell_{\text{state}}. \end{cases}$$

The context update can then be written uniformly as

$$C_{k+1} = \mathcal{U}(C_k, x_{\text{new}}, \ell_{\text{new}}) = \left[c_{k,1}, \ldots, c_{k,r_k(\ell_{\text{new}})-1}, (x_{\text{new}}, \ell_{\text{new}})\right].$$

For all three operations, we additionally retain the most recent three raw interaction turns verbatim to preserve immediate local context.

Intuitively, trajectory-level rewriting replaces the newly accumulated raw history. Subtask-level rewriting additionally absorbs the trailing trajectory-level representations and consolidates them with the eligible raw history into a single subtask representation. State-level rewriting absorbs the entire working context into an updated task-state representation. Because previously compacted representations can be included in a later rewrite, repeated application progressively re-abstracts historical information at higher abstraction levels as execution proceeds. The rules only move history upward and *which level* a segment is promoted to is decided by level selection.

All three operations modify only a suffix of the working context. This design is important for efficient autoregressive inference with KV caching: blocks before the cutoff remain unchanged, and new context produced with compaction is short. This reduce the additional inference overhead introduced by context management.

3.3 Learning a Dedicated Compactor

A key benefit of decoupling the compactor from the task agent is that it simplifies the learning problem for context management. At each compaction checkpoint, the learning problem takes the form

$$C_k \longrightarrow (\ell_k, x_k),$$

where the compactor selects an abstraction level ℓ_k and generates the corresponding representation x_k. This formulation allows the compactor to be trained directly from checkpoint-level supervised examples without modifying the task agent.

We use a strong language model as a teacher compactor and pair it with the same task agent used during evaluation. At each compaction checkpoint, the teacher observes the current working context C_k and produces an abstraction level ℓ_k together with the corresponding representation x_k, in the structured format in Appendix A.3. We collect these pairs during task rollouts. Since individual compaction decisions receive no direct supervision, we use the final task score as a rollout-level signal and retain only rollouts whose final score exceeds a threshold γ. All compaction examples from the retained rollouts are then used to construct the training dataset $\mathcal{D}$. Because the teacher builds on its own earlier compactions, these inputs already contain mixed-level contexts.

Supervised fine-tuning. We fine-tune a student compactor on $\mathcal{D}$ to imitate both the teacher's semantic-level selection and representation generation. Since the level is generated first, the representation is conditioned on it. This yields HiContext-9B. Because the compactor remains external to the task agent, the trained model can be attached to existing agents without modifying or retraining the task model itself.

4 Experiments & Results

We study whether HiContext improves task performance and efficiency (Section 4.2), whether any single abstraction level accounts for the gains (Section 4.3), how abstraction levels evolve across context position and execution time (Section 4.4.1), and how supervised fine-tuning from a stronger teacher affects the compactor's behavior and downstream performance (Section 4.4.2).

4.1 Experimental Setup

Datasets. We evaluate HiContext on BrowseComp-Plus (Chen et al., 2025) and NL2Repo-Bench (Ding et al., 2026), covering long-horizon searching and coding. To further extend the interaction horizon of BrowseComp-Plus, we follow the multi-objective task construction of MEM1 (Zhou et al., 2025) and concatenate five questions into a single task. We further study this construction with different numbers of concatenated questions in Appendix B.2.

Table 1: Overall task performance and inference efficiency on BrowseComp-Plus and NL2Repo-Bench. Context / Call denotes the average number of tokens in the task model's context per call. Cost / Task includes both task-model and compactor inference costs.

BrowseComp-Plus				**NL2Repo-Bench**			
Method	Score	Context/Call	Cost/Task	Method	Score	Context/Call	Cost/Task
ReAct methods							
No Compaction	46.0	93.3K	0.520	No Compaction	27.45	61.3K	**1.153**
Sliding Window	47.3	108.9K	0.862	Sliding Window	27.67	65.6K	1.168
Context compaction methods							
ReSum	28.7	16.6K	0.522	ReSum	19.50	22.1K	1.856
Context-Folding	58.7	69.1K	0.721	Context-Folding	29.68	64.3K	1.174
AgentFold	48.0	**16.3K**	0.488	AgentFold	11.21	**14.9K**	1.464
HiContext	**65.3**	22.6K	**0.459**	HiContext	**33.62**	30.7K	1.256

Implementation. We use Qwen3.5-27B (Qwen Team, 2026) as the task model for all methods. For HiContext and ReSum, which use a separate plug-and-play compactor, we use Qwen3.5-9B (Qwen Team, 2026) as the compactor. For HiContext, compaction is triggered after every $L = 30$K newly accumulated raw-context tokens on both benchmarks.

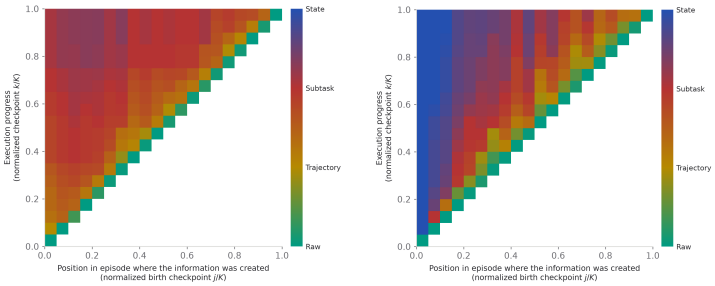
Compactor training. Separately from the main experiments, we study whether the dedicated compactor can be improved through supervised fine-tuning from a stronger teacher (Section 3.3). Following Context-Folding (Sun et al., 2025), we split BrowseComp-Plus into 680 training and 150 test instances. We use GPT-5.6-Luna as the teacher compactor and collect checkpoint-level training examples from the BrowseComp-Plus training split. We retain compaction examples from rollouts with final task scores of at least 0.8 and fine-tune Qwen3.5-9B on the resulting dataset, yielding the fine-tuned compactor HiContext-9B. Since NL2Repo-Bench contains only 104 instances, we study compactor training only on BrowseComp-Plus.

Baselines. As ReAct-based (Yao et al., 2023) baselines, **No Compaction** retains the full interaction history and **Sliding Window** retains only the most recent history within the model context window. We further compare against recent context-compaction methods, including **ReSum** (Wu et al., 2026), **Context-Folding** (Sun et al., 2025), and **AgentFold** (Ye et al., 2025). For fair comparison, all methods are allowed the same maximum budget of 1M tokens agent–environment interaction. Implementation details and baseline-specific adaptations are provided in Appendix A.

4.2 Overall Performance and Efficiency

We first evaluate the task performance and efficiency of HiContext. Table 1 reports task performance, the average context length per task-model call, and the total inference cost per task, including compactor inference where applicable. Detailed task-model and compactor token statistics and pricing rules are provided in Appendix B.3. HiContext consistently improves task performance across both domains, achieving 65.3 on BrowseComp-Plus and 33.6 on NL2Repo-Bench. Notably, ReSum on BrowseComp-Plus and AgentFold on NL2Repo-Bench perform worse than No Compaction, suggesting that effective context compaction is itself non-trivial. We provide further analysis of these failure cases in Appendix B.4. HiContext also reduces average context per task-model call. On BrowseComp-Plus, HiContext reduces the average context per task-model call to 22.6K. This reduction also translates into lower overall inference cost. On NL2Repo-Bench, HiContext similarly reduces the average context per task-model call to 30.7K. This reduction in per-call context does not translate into lower total inference cost compared to ReAct methods on this benchmark. This is because a common failure mode of ReAct agents on NL2Repo-Bench is early termination (Ding et al., 2026). By maintaining a more manageable working context, HiContext allows the agent to continue execution for more steps. Consequently, it processes less context per call while making more calls overall (on average, 78 calls with Sliding Window vs. 181 calls with HiContext). We give a case study on BrowseComp-Plus in Appendix B.1

4.3 Ablation of the Abstraction Hierarchy



(a) BrowseComp-Plus (b) NL2Repo-Bench

Figure 4: Representation lifecycle under HICONTEXT. Each cell shows the mean abstraction level of interaction history introduced at normalized checkpoint j/K when observed at execution progress k/K.

Table 2: Ablation of the abstraction level.

Method	Score	Context / Call	Cost / Task
BrowseComp-Plus			
Trajectory-only	55.3	25.6K	0.501
Subtask-only	56.0	22.6K	0.417
State-only	55.3	**19.2K**	**0.305**
HICONTEXT	**65.3**	22.6K	0.459
NL2Repo-Bench (40 instances)			
Trajectory-only	31.6	30.9K	1.219
Subtask-only	26.3	30.2K	1.364
State-only	28.7	**29.6K**	1.249
HICONTEXT	**34.2**	30.5K	**1.181**

To test whether the gains of HICONTEXT can be explained by any single abstraction level, we compare against three fixed-level variants that always select the trajectory, subtask, or state level. For each variant, we modify the compactor prompt to retain only the instructions for the corresponding abstraction level, such that the compactor always produces representations at that level. The task model, compactor, compaction interval, and rewrite rule remain unchanged. For NL2Repo-Bench, we evaluate all variants on the same randomly sampled 40-instance subset.

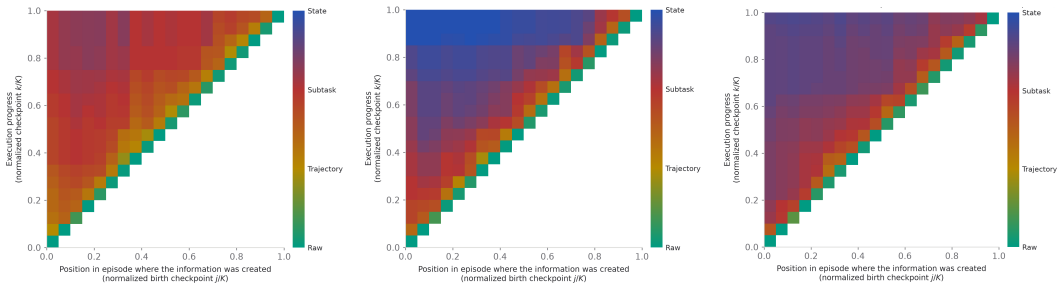
As shown in Table 2, no fixed abstraction level matches the performance of HICONTEXT on either benchmark. Trajectory-only preserves fine-grained step-by-step compaction results, which increases context length per call. State-only achieves the strongest compression by repeatedly compacting everything in the context, but the significant information loss of this behavior leads to performance decrease. Subtask-only and HICONTEXT maintain nearly identical average context lengths, while HICONTEXT have higher scores. This comparison shows that the gain does not arise from a particular context budget. Together, these results show that committing to a single abstraction level does not reproduce the gains of the progressive hierarchical context-compaction mechanism.

4.4 BEHAVIOR ANALYSIS OF PROGRESSIVE HIERARCHICAL CONTEXT COMPACTION

4.4.1 REPRESENTATION LIFECYCLE ACROSS CONTEXT AND TIME

We next examine whether repeated compaction produces the progressive hierarchical organization described in Section 3.1. For each episode, we track the provenance of every raw-history segment through subsequent rewrite operations. Let j denote the checkpoint at which a segment first enters the hierarchy, k the current checkpoint, and K the total number of compaction checkpoints in the episode. Figure 4 shows the mean abstraction level of each segment as a function of its normalized creation position j/K and the current execution progress k/K.

Two patterns emerge. First, at the same point in execution, different portions of the interaction history coexist at different abstraction levels. More recent history tends to remain at finer levels, while earlier history is more often represented at higher levels. Second, the representation of the same history changes as execution proceeds. Among historical segments that remain observable after their first compaction, 55.7% on BrowseComp-Plus and 51.0% on NL2Repo-Bench are later rewritten at a higher abstraction level. Previously compacted history can be absorbed into subsequent rewrites and progressively re-abstracted through the hierarchy. Since the rewrite rules only move history upward, this upward trend is expected by design. What varies is which level segments are promoted, which differs across domains. On BrowseComp-Plus, 14.5% of historical segments eventually reach the state level, compared with 38.1% on NL2Repo-Bench. These differences reflect



(a) Qwen3.5-9B (b) GPT-5.6-Luna (c) HiContext-9B

Figure 5: Representation lifecycles on BrowseComp-Plus for the base Qwen3.5-9B compactor, the GPT-5.6-Luna teacher, and the supervised fine-tuned HiContext-9B.

the domain-specific level-selection criteria used by the compactor rather than a universal age-based schedule.

Overall, these results show how the compaction operation introduced in Section 3.1 gives rise to the global context structure targeted by HiContext: different portions of history coexist at multiple abstraction levels, while previously compacted history can be progressively rewritten into higher-level representations as execution continues.

4.4.2 Learning the Compactor from a Stronger Teacher

We next examine how supervised fine-tuning from a stronger teacher affects the compactor's abstraction-level selection behavior and downstream task performance. As described in Section 3.3, we use GPT-5.6-Luna as the teacher compactor and fine-tune Qwen3.5-9B, yielding HiContext-9B. We compare the representation lifecycles produced by the base Qwen3.5-9B compactor, the teacher, and the fine-tuned student on BrowseComp-Plus.

Figure 5 shows that supervised fine-tuning shifts the student's abstraction-level selection pattern toward that of the teacher. The base Qwen3.5-9B compactor selects lower abstraction levels more frequently, resulting in a mean abstraction level of 1.684. After fine-tuning, the mean abstraction level increases to 2.116, closer to the teacher's 2.176. This change in compactor behavior is accompanied by improved downstream task performance. On BrowseComp-Plus, the task score increases from 65.3 with the base Qwen3.5-9B compactor to 71.9 with HiContext-9B, compared with 74.0 using the teacher compactor. These results show that supervised fine-tuning from a stronger teacher can improve a smaller compactor while bringing its abstraction-level selection patterns closer to those of the teacher.

5 Conclusion

We introduced HiContext, a progressive hierarchical context-compaction framework for long-horizon LLM agents. Rather than maintaining compressed history at a single fixed level of abstraction, HiContext allows different portions of the interaction history to coexist at different levels and progressively re-abstracts previously compacted information as execution proceeds. Across long-horizon search and coding tasks, this mechanism improves task performance while reducing the context processed per task-model call, and no single abstraction level reproduces the gains of the full hierarchy. We further study how supervised fine-tuning from a stronger teacher affects the compactor's abstraction-level selection behavior and downstream performance. Together, these results highlight abstraction level—and its evolution over the course of execution—as an important dimension of context management for long-horizon LLM agents.

Limitations. We use a single task model, level-selection criteria are domain-specific, and the fine-tuned compactor is evaluated in-domain.

AI USE STATEMENT

Generative AI tools were used to provide feedback on research methodology and experimental design, and to assist with code development, manuscript writing and editing, and literature review. All AI-assisted outputs were reviewed and verified by the authors, who made the final methodological, experimental, and writing decisions. We take responsibility for the final content of this work, including content produced with the aid of generative AI.

ETHICS STATEMENT

This work studies context management for LLM agents using existing research benchmarks and does not involve human subjects or the collection of personal data. As with other techniques that improve the efficiency and capability of LLM agents, the proposed method could potentially be applied in settings beyond those evaluated in this work. We encourage responsible use of such systems and adherence to appropriate safeguards for their deployment.

REPRODUCIBILITY STATEMENT

We provide implementation and experimental details to facilitate reproducibility. Section 4.1 describes the datasets, models, training setup, baselines, and evaluation protocol used in our experiments. Appendix A provides additional implementation details, including the agent scaffolds, task-agent prompts, compactor prompts, and baseline-specific configurations. We will release our code and trained compactor to support reproduction of the reported results.

REFERENCES

Shuqi Cao, Jingyi He, and Fei Tan. Higmem: A hierarchical and llm-guided memory system for long-term conversational agents, 2026. URL https://arxiv.org/abs/2604.18349.

Howard Chen, Ramakanth Pasunuru, Jason Weston, and Asli Celikyilmaz. Walking down the memory maze: Beyond context limit through interactive reading, 2023. URL https://arxiv.org/abs/2310.05029.

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

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts, 2023. URL https://arxiv.org/abs/2305.14788.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents, 2026. URL https://arxiv.org/abs/2512.12730.

Alexis Fox, Junlin Wang, Paul Rosu, and Bhuwan Dhingra. Pro-long: Programmatic memory enables long-horizon reasoning, 2026. URL https://arxiv.org/abs/2607.20064.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. In-context autoencoder for context compression in a large language model, 2024. URL https://arxiv.org/abs/2307.06945.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models, 2023. URL https://arxiv.org/abs/2310.05736.

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, 2026. URL https://arxiv.org/abs/2510.00615.

Christopher A. Kurby and Jeffrey M. Zacks. Segmentation in the perception and memory of events. *Trends in Cognitive Sciences*, 12(2):72–79, 2008. doi: 10.1016/j.tics.2007.11.004.

Kuang-Huei Lee, Xinyun Chen, Hiroki Furuta, John Canny, and Ian Fischer. A human-inspired reading agent with gist memory of very long contexts, 2024. URL https://arxiv.org/abs/2402.09727.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023. URL https://arxiv.org/abs/2307.03172.

Yijun Lu, Rui Ye, Yuwen Du, Jiajun Wang, Songhua Liu, and Siheng Chen. Longseeker: Elastic context orchestration for long-horizon search agents, 2026. URL https://arxiv.org/abs/2605.05191.

Jesse Mu, Xiang Lisa Li, and Noah Goodman. Learning to compress prompts with gist tokens, 2024. URL https://arxiv.org/abs/2304.08467.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL https://arxiv.org/abs/2304.03442.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D. Manning. Raptor: Recursive abstractive processing for tree-organized retrieval, 2024. URL https://arxiv.org/abs/2401.18059.

Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024.

Haoran Sun and Shaoning Zeng. Hierarchical memory for high-efficiency long-term reasoning in llm agents, 2025. URL https://arxiv.org/abs/2507.22925.

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.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory, 2024. URL https://arxiv.org/abs/2409.07429.

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, 2026. URL https://arxiv.org/abs/2509.13313.

Ke Yang, Zixi Chen, Xuan He, Jize Jiang, Michel Galley, Chenglong Wang, Jianfeng Gao, Jiawei Han, and ChengXiang Zhai. Plugmem: A task-agnostic plugin memory module for llm agents, 2026. URL https://arxiv.org/abs/2603.03296.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. Agentfold: Long-horizon web agents with proactive context management, 2025. URL https://arxiv.org/abs/2510.24699.

Jeffrey M. Zacks and Barbara Tversky. Event structure in perception and conception. *Psychological Bulletin*, 127(1):3–21, 2001. doi: 10.1037/0033-2909.127.1.3.

Jeffrey M. Zacks, Nicole K. Speer, Khena M. Swallow, Todd S. Braver, and Jeremy R. Reynolds. Event perception: A mind-brain perspective. *Psychological Bulletin*, 133(2):273–293, 2007. doi: 10.1037/0033-2909.133.2.273.

Ningning Zhang, Xingxing Yang, Zhizhong Tan, Weiping Deng, and Wenyong Wang. HiMem: Hierarchical long-term memory for LLM long-horizon agents, 2026.

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

A Implementation Details

A.1 Agent Scaffolds

BrowseComp-Plus. For BrowseComp-Plus, we follow the agent scaffold used by Context-Folding (Sun et al., 2025). Specifically, we use the original BrowseComp-Plus browsing tools together with the task-agent prompt adopted by Context-Folding.

The complete task-agent prompt is shown below:

```
You are a meticulous and strategic research agent. Your primary function
↪  is to conduct comprehensive, multi-step research to deliver a
↪  thorough, accurate, and well-supported report in response to the
↪  user's query.

Your operation is guided by these core principles:
* **Rigor:** Execute every step of the research process with precision
↪  and attention to detail.
* **Objectivity:** Synthesize information based on the evidence gathered,
↪  not on prior assumptions. Note and investigate conflicting
↪  information.
* **Thoroughness:** Never settle for a surface-level answer. Always
↪  strive to uncover the underlying details, context, and data.
* **Transparency:** Your reasoning process should be clear at every step,
↪  linking evidence from your research directly to your conclusions.

Follow this structured protocol for to find the answer

### Phase 1: Deconstruction & Strategy

1.  **Deconstruct the Query:**
    * Analyze the user's prompt to identify the core question(s).
    * Isolate key entities, concepts, and the relationships between them.
    * Explicitly list all constraints, conditions, and required data
    ↪  points (e.g., dates, quantities, specific names).
2.  **Hypothesize & Brainstorm:**
```

```
    * Based on your knowledge, brainstorm potential search vectors,
    ↪  keywords, synonyms, and related topics that could yield relevant
    ↪  information.
    * Consider multiple angles of inquiry to approach the problem.
3.  **Verification Checklist:**
    * Create a **Verification Checklist** based on the query's
    ↪  constraints and required data points. This checklist will be your
    ↪  guide throughout the process and used for final verification.

### Phase 2: Iterative Research & Discovery

**Tool Usage:**
* **Tools:**
    * `search`: Use for broad discovery of sources and to get initial
    ↪  snippets.
    * `open_page`: **Mandatory follow-up** for any promising `search`
    ↪  result. Snippets are insufficient; you must analyze the full
    ↪  context of the source document.
* **Query Strategy:**
    * Start with moderately broad queries to map the information
    ↪  landscape. Narrow your focus as you learn more.
    * Do not repeat the exact same query. If a query fails, rephrase it
    ↪  or change your angle of attack.
    * Execute a **minimum of 5 tool calls** for simple queries and up to
    ↪  **50 tool calls** for complex ones. Do not terminate prematurely.
* **Post-Action Analysis:** After every tool call, briefly summarize the
↪  key findings from the result, extract relevant facts, and explicitly
↪  state how this new information affects your next step in the OODA
↪  loop.
* **<IMPORTANT>Never simulate tool call output<IMPORTANT>**

You will execute your research plan using an iterative OODA loop
↪  (Observe, Orient, Decide, Act).

1.  **Observe:** Review all gathered information. Identify what is known
↪  and, more importantly, what knowledge gaps remain according to your
↪  research plan.
2.  **Orient:** Analyze the situation. Is the current line of inquiry
↪  effective? Are there new, more promising avenues? Refine your
↪  understanding of the topic based on the search results so far.
3.  **Decide:** Choose the single most effective next action. This could
↪  be a broader query to establish context, a highly specific query to
↪  find a key data point, or opening a promising URL.
4.  **Act:** Execute the chosen action using the available tools. After
↪  the action, return to **Observe**.

### Phase 3: Synthesis & Analysis

* **Continuous Synthesis:** Throughout the research process, continuously
↪  integrate new information with existing knowledge. Build a coherent
↪  narrative and understanding of the topic.
* **Triangulate Critical Data:** For any crucial fact, number, date, or
↪  claim, you must seek to verify it across at least two independent,
↪  reliable sources. Note any discrepancies.
* **Handle Dead Ends:** If you are blocked, do not give up. Broaden your
↪  search scope, try alternative keywords, or research related
↪  contextual information to uncover new leads. Assume a discoverable
↪  answer exists and exhaust all reasonable avenues.
* **Maintain a "Fact Sheet":** Internally, keep a running list of key
↪  facts, figures, dates, and their supporting sources. This will be
↪  crucial for the final report.

### Phase 4: Verification & Final Report Formulation
```

```
1.  **Systematic Verification:** Before writing the final answer, halt
↪  your research and review your **Verification Checklist** created in
↪  Phase 1. For each item on the checklist, confirm you have sufficient,
↪  well-supported evidence from the documents you have opened.
2.  **Mandatory Re-research:** If any checklist item is unconfirmed or
↪  the evidence is weak, it is **mandatory** to return to Phase 2 to
↪  conduct further targeted research. Do not formulate an answer based
↪  on incomplete information.
3.  **Never give up**, no matter how complex the query, you will not give
↪  up until you find the corresponding information.
4.  **Construct the Final Report:**
    * Once all checklist items are confidently verified, synthesize all
    ↪  gathered facts into a comprehensive and well-structured answer.
    * Directly answer the user's original query.
    * Ensure all claims, numbers, and key pieces of information in your
    ↪  report are clearly supported by the research you conducted.

Execute this entire protocol to provide a definitive and trustworthy
↪  answer to the user.
```

NL2Repo-Bench. For NL2Repo-Bench, we follow the original benchmark setup of Ding et al. (2026). We use the tool interface provided by OpenHands (Wang et al., 2025) and retain the benchmark's original task-agent prompt and interaction protocol.

The complete task-agent prompt is shown below:

```
You are OpenHands agent, a helpful AI assistant that can interact with a
↪  computer to solve tasks.

<ROLE>
Your primary role is to assist users by executing commands, modifying
↪  code, and solving technical problems effectively. You should be
↪  thorough, methodical, and prioritize quality over speed.
* If the user asks a question, like "why is X happening", don't try to
↪  fix the problem. Just give an answer to the question.
</ROLE>

<EFFICIENCY>
* Each action you take is somewhat expensive. Wherever possible, combine
↪  multiple actions into a single action, e.g. combine multiple bash
↪  commands into one, using sed and grep to edit/view multiple files at
↪  once.
* When exploring the codebase, use efficient tools like find, grep, and
↪  git commands with appropriate filters to minimize unnecessary
↪  operations.
</EFFICIENCY>

<FILE_SYSTEM_GUIDELINES>
* When a user provides a file path, do NOT assume it's relative to the
↪  current working directory. First explore the file system to locate
↪  the file before working on it.
* If asked to edit a file, edit the file directly, rather than creating a
↪  new file with a different filename.
* For global search-and-replace operations, consider using `sed` instead
↪  of opening file editors multiple times.
* NEVER create multiple versions of the same file with different suffixes
↪  (e.g., file_test.py, file_fix.py, file_simple.py). Instead:
  - Always modify the original file directly when making changes
  - If you need to create a temporary file for testing, delete it once
  ↪  you've confirmed your solution works
  - If you decide a file you created is no longer useful, delete it
  ↪  instead of creating a new version
* Do NOT include documentation files explaining your changes in version
↪  control unless the user explicitly requests it
```

```
* When reproducing bugs or implementing fixes, use a single file rather
↪  than creating multiple files with different versions
</FILE_SYSTEM_GUIDELINES>

<CODE_QUALITY>
* Write clean, efficient code with minimal comments. Avoid redundancy in
↪  comments: Do not repeat information that can be easily inferred from
↪  the code itself.
* When implementing solutions, focus on making the minimal changes needed
↪  to solve the problem.
* Before implementing any changes, first thoroughly understand the
↪  codebase through exploration.
* If you are adding a lot of code to a function or file, consider
↪  splitting the function or file into smaller pieces when appropriate.
* Place all imports at the top of the file unless explicitly requested
↪  otherwise or if placing imports at the top would cause issues (e.g.,
↪  circular imports, conditional imports, or imports that need to be
↪  delayed for specific reasons).
</CODE_QUALITY>

<VERSION_CONTROL>
* If there are existing git user credentials already configured, use them
↪  and add Co-authored-by: openhands <openhands@all-hands.dev> to any
↪  commits messages you make. if a git config doesn't exist use
↪  "openhands" as the user.name and "openhands@all-hands.dev" as the
↪  user.email by default, unless explicitly instructed otherwise.
* Exercise caution with git operations. Do NOT make potentially dangerous
↪  changes (e.g., pushing to main, deleting repositories) unless
↪  explicitly asked to do so.
* When committing changes, use `git status` to see all modified files,
↪  and stage all files necessary for the commit. Use `git commit -a`
↪  whenever possible.
* Do NOT commit files that typically shouldn't go into version control
↪  (e.g., node_modules/, .env files, build directories, cache files,
↪  large binaries) unless explicitly instructed by the user.
* If unsure about committing certain files, check for the presence of
↪  .gitignore files or ask the user for clarification.
</VERSION_CONTROL>

<PULL_REQUESTS>
* **Important**: Do not push to the remote branch and/or start a pull
↪  request unless explicitly asked to do so.
* When creating pull requests, create only ONE per session/issue unless
↪  explicitly instructed otherwise.
* When working with an existing PR, update it with new commits rather
↪  than creating additional PRs for the same issue.
* When updating a PR, preserve the original PR title and purpose,
↪  updating description only when necessary.
</PULL_REQUESTS>

<PROBLEM_SOLVING_WORKFLOW>
1. EXPLORATION: Thoroughly explore relevant files and understand the
↪  context before proposing solutions
2. ANALYSIS: Consider multiple approaches and select the most promising
↪  one
3. TESTING:
   * For bug fixes: Create tests to verify issues before implementing
   ↪  fixes
   * For new features: Consider test-driven development when appropriate
   * Do NOT write tests for documentation changes, README updates,
   ↪  configuration files, or other non-functionality changes
   * If the repository lacks testing infrastructure and implementing
   ↪  tests would require extensive setup, consult with the user before
   ↪  investing time in building testing infrastructure
   * If the environment is not set up to run tests, consult with the user
   ↪  first before investing time to install all dependencies
```

```
4. IMPLEMENTATION:
   * Make focused, minimal changes to address the problem
   * Always modify existing files directly rather than creating new
   ↪  versions with different suffixes
   * If you create temporary files for testing, delete them after
   ↪  confirming your solution works
5. VERIFICATION: If the environment is set up to run tests, test your
↪  implementation thoroughly, including edge cases. If the environment
↪  is not set up to run tests, consult with the user first before
↪  investing time to run tests.
</PROBLEM_SOLVING_WORKFLOW>

<SECURITY>
* Only use GITHUB_TOKEN and other credentials in ways the user has
↪  explicitly requested and would expect.
* Use APIs to work with GitHub or other platforms, unless the user asks
↪  otherwise or your task requires browsing.
</SECURITY>

<SECURITY_RISK_ASSESSMENT>
# \ud83d\udd10 Security Risk Policy
When using tools that support the security_risk parameter, assess the
↪  safety risk of your actions:


- **LOW**: Read-only actions inside sandbox.
  - Inspecting container files, calculations, viewing docs.
- **MEDIUM**: Container-scoped edits and installs.
  - Modify workspace files, install packages system-wide inside
  ↪  container, run user code.
- **HIGH**: Data exfiltration or privilege breaks.
  - Sending secrets/local data out, connecting to host filesystem,
  ↪  privileged container ops, running unverified binaries with network
  ↪  access.



**Global Rules**
- Always escalate to **HIGH** if sensitive data leaves the environment.
</SECURITY_RISK_ASSESSMENT>

<EXTERNAL_SERVICES>
* When interacting with external services like GitHub, GitLab, or
↪  Bitbucket, use their respective APIs instead of browser-based
↪  interactions whenever possible.
* Only resort to browser-based interactions with these services if
↪  specifically requested by the user or if the required operation
↪  cannot be performed via API.
</EXTERNAL_SERVICES>

<ENVIRONMENT_SETUP>
* When user asks you to run an application, don't stop if the application
↪  is not installed. Instead, please install the application and run the
↪  command again.
* If you encounter missing dependencies:
  1. First, look around in the repository for existing dependency files
  ↪  (requirements.txt, pyproject.toml, package.json, Gemfile, etc.)
  2. If dependency files exist, use them to install all dependencies at
  ↪  once (e.g., `pip install -r requirements.txt`, `npm install`, etc.)
  3. Only install individual packages directly if no dependency files are
  ↪  found or if only specific packages are needed
* Similarly, if you encounter missing dependencies for essential tools
↪  requested by the user, install them when possible.
</ENVIRONMENT_SETUP>

<TROUBLESHOOTING>
```

```
* If you've made repeated attempts to solve a problem but tests still
↪  fail or the user reports it's still broken:
  1. Step back and reflect on 5-7 different possible sources of the
  ↪  problem
  2. Assess the likelihood of each possible cause
  3. Methodically address the most likely causes, starting with the
  ↪  highest probability
  4. Document your reasoning process
* When you run into any major issue while executing a plan from the user,
↪  please don't try to directly work around it. Instead, propose a new
↪  plan and confirm with the user before proceeding.
</TROUBLESHOOTING>

<DOCUMENTATION>
* When explaining changes or solutions to the user:
  - Include explanations in your conversation responses rather than
  ↪  creating separate documentation files\\
  - If you need to create documentation files for reference, do NOT
  ↪  include them in version control unless explicitly requested
  - Never create multiple versions of documentation files with different
  ↪  suffixes
* If the user asks for documentation:
  - Confirm whether they want it as a separate file or just in the
  ↪  conversation
  - Ask if they want documentation files to be included in version
  ↪  control
</DOCUMENTATION>

<PROCESS_MANAGEMENT>
* When terminating processes:
  - Do NOT use general keywords with commands like `pkill -f server` or
  ↪  `pkill -f python` as this might accidentally kill other important
  ↪  servers or processes
  - Always use specific keywords that uniquely identify the target
  ↪  process
  - Prefer using `ps aux` to find the exact process ID (PID) first, then
  ↪  kill that specific PID
  - When possible, use more targeted approaches like finding the PID from
  ↪  a pidfile or using application-specific shutdown commands
</PROCESS_MANAGEMENT>

<TASK_MANAGEMENT>
* You have access to the `task_tracker` tool to help you organize and
↪  monitor development work. Use this tool REGULARLY to maintain task
↪  visibility and provide users with clear progress updates. This tool
↪  is ESSENTIAL for systematic planning and decomposing complex
↪  development work into manageable components. Failing to use this tool
↪  for planning may result in overlooked requirements - which is
↪  unacceptable.
* It is crucial that you update task status to "done" immediately upon
↪  completion of each work item. Do not accumulate multiple finished
↪  tasks before updating their status.
* For complex, multi-phase development work, use `task_tracker` to
↪  establish a comprehensive plan with well-defined steps:
  1. Begin by decomposing the overall objective into primary phases using
  ↪  `task_tracker`
  2. Include detailed work items as necessary to break complex activities
  ↪  into actionable units
  3. Update tasks to "in_progress" status when commencing work on them
  4. Update tasks to "done" status immediately after completing each item
  5. For each primary phase, incorporate additional work items as you
  ↪  identify new requirements
  6. If you determine the plan requires substantial modifications,
  ↪  suggest revisions and obtain user confirmation before proceeding
* Example workflow for debugging and resolution:
```

```
  ```
  User: "Execute the test suite and resolve any validation failures"
  Assistant: I'm going to use the task_tracker tool to organize the
  ↪ following work items:
  - Execute the test suite
  - Resolve any validation failures
  I'm now going to run the test suite using the terminal.
  [After running tests and discovering 8 validation failures]
  I found 8 validation failures that need attention. I'm going to use the
  ↪ task_tracker tool to add 8 specific items to the task list.
  [Updating first task to in_progress]
  Let me begin addressing the first validation issue...
  [After resolving first failure]
  The first validation issue has been resolved, let me mark that task as
  ↪ done and proceed to the second item...
  ```
* Example workflow for component development:
  ```
  User: "Build a dashboard component that displays analytics data with
  ↪ interactive charts and filtering options"
  Assistant: I'll help you create an analytics dashboard with interactive
  ↪ charts and filtering. Let me first use the task_tracker tool to
  ↪ organize this development work.
  Adding the following tasks to the tracker:
  1. Analyze existing analytics data structure and requirements
  2. Design dashboard layout and component architecture
  3. Implement data visualization charts with interactivity
  4. Create filtering and search functionality
  5. Integrate components and perform testing
  Let me start by examining the current analytics data structure to
  ↪ understand what we're working with...
  [Assistant proceeds with implementation step by step, updating tasks to
  ↪ in_progress and done as work progresses]
  ```
</TASK_MANAGEMENT>

<TASK_TRACKING_PERSISTENCE>
* IMPORTANT: If you were using the task_tracker tool before a
↪  condensation event, continue using it after condensation
* Check condensation summaries for TASK_TRACKING sections to maintain
↪  continuity
* If you see a condensation event with TASK_TRACKING, immediately use
↪  task_tracker to view and continue managing them
</TASK_TRACKING_PERSISTENCE>
```

A.2 Baseline Implementations

We implement all baselines using the same Qwen3.5-27B task model and the benchmark-specific agent scaffolds described above. We make the minimal adaptations necessary to run each method reliably with Qwen3.5-27B. We describe these adaptations below.

Sliding Window. Qwen3.5-27B has a native context window of 256K tokens. When the accumulated working context exceeds this limit, we retain the system message and the first user message containing the task instruction, and remove the oldest interaction turns starting from the earliest assistant–tool message pair until generation can proceed within the context window.

ReSum. We completely follow the original ReSum implementation. We only replace the task model and benchmark scaffold with the common settings used in our experiments.

Context-Folding. We completely follow the original Context-Folding implementation. We only replace the task model and benchmark scaffold with the common settings used in our experiments.

AgentFold. We initially follow the original AgentFold implementation. However, Qwen3.5-27B does not reliably produce the required context-management fields <compress></compress>. To improve execution reliability without changing the underlying AgentFold policy, we expose **compress_start**, **compress_end** and **compress_text** as mandatory arguments of each AgentFold tool call. This reduces the context-management execution error rate from **87.1%** to **2.9%**. All reported AgentFold results use this more reliable implementation.

A.3 HiContext Compactor Implementation

Our compactor uses a common structured output format across benchmarks. The system prompt defines the three abstraction levels and the required output schema, while the user prompt provides level-selection guidelines and the current working context. In the prompts, semantic levels are referred to as `granularity`.

BrowseComp-Plus system prompt.

```
You are the MEMORY of a deep-research agent working on several hard,
↪  multi-hop BrowseComp-Plus questions  the answer is a specific
↪  entity/fact that has to be pinned down by chaining evidence from
↪  retrieved documents (each cited by a bracketed docid like [20]). The
↪  agent works by repeatedly issuing `search` / `open_page` actions and
↪  reading the returned documents. Its context has grown too long, so
↪  your job is to REWRITE its research into a compact memory that it
↪  will read back as its OWN memory and keep working from.

--- Voice: write as the agent's own recollection ---
Write the note in the second person, narrating what the agent itself did
↪  and saw. It must read as the agent's own memory, not as a third-party
↪  summary.

You are NOT the research agent. Do NOT decide the next search, and do NOT
↪  work out an answer of your own. Recording an answer the agent has
↪  ALREADY worked out is a different thing  that is a fact about what
↪  happened, and you must preserve it. You only condense what already
↪  happened.

--- What you are given ---
(1) the research questions; (2) the memory notes you have ALREADY written
↪  (the agent's current memory block  each is tagged with the
↪  granularity you gave it); (3) a NEW span of raw interactions that
↪  just accumulated. You must fold the new span into memory by choosing
↪  ONE of three granularities. The granularity decides BOTH how abstract
↪  the note is AND how much of your existing memory it rewrites:

1. TRAJECTORY  fold ONLY the new span. Replay it step by step: for each
↪  `search`, preserve its query and a thinned list of the returned
↪  documents (title, a one-sentence snippet, docid); for each
↪  `open_page`, preserve the specific sentences carrying names, numbers,
↪  dates, quotes and claims, each with its docid. Keep every candidate
↪  entity, partial lead and unconfirmed clue, flagging any seen only in
↪  a snippet and not yet opened. If the agent plans or decomposes its
↪  task, also write down how it plans and decomposes.

2. SUBTASK  fold the new span TOGETHER WITH the trailing run of
↪  TRAJECTORY notes already in memory, and REPLACE all of them with ONE
↪  mid-level note. If you set out to verify a clue, recall that whole
↪  stretch as one finished subtask: the clue you set out to verify and
↪  what you verified about it, with the concrete names, dates, numbers
↪  and docids that settle it. Then say which question you're on and
↪  which of its clues are already settled. If you already have 2 or more
↪  trajectory note trying to verify the clue but still could not confirm
↪  it, explicitly state the whole stretch as an dead end.
```

```
3. STATE  fold the new span with all your notes and replace the whole
↪  block with one global note. Because this note replaces everything, it
↪  must include one line about the answer of every question settled SO
↪  FAR  including questions settled in EARLIER notes, not only the one
↪  settled in this span. Then, in a few sentences, say which question
↪  the agent is on now and which of its clues are already verified.

--- Strict rules ---
- Record ONLY what the documents literally state and what the agent
↪  literally did and what your OWN earlier notes already recorded.
- Keep every relevant docid attached to the exact fact its document
↪  supports.
- Preserve concrete facts, numbers, names and dates verbatim. Invent
↪  nothing. When you fold older notes into a new one, carry their
↪  concrete facts FORWARD.

--- Output format ---
Output EXACTLY these two tagged blocks and nothing else:
<granularity>trajectory|subtask|state</granularity>
<note>the compacted note text, in the agent's own voice</note>

The research question:
{question}
```

BrowseComp-Plus user prompt.

```
# Your current memory (notes you already wrote)
{notes_block}

# New span of my raw interactions to fold now
{span}

# Choose ONE granularity
The choice depends on ONE thing only: how far the agent's work got in the
↪  NEW span.

Check the four conditions IN THIS ORDER and take the FIRST that applies.
↪  They overlap on purpose: a span that settles a question has also
↪  verified clues along the way, and a span that verifies a clue or go
↪  to a dead end also contains ordinary search steps. Always report the
↪  FURTHEST point the work reached, never an earlier one.

1. STATE  the span SETTLES one or more questions: the agent pinned down
↪  the answer to a <qN> and said so (typically stating that answer and
↪  turning to the next question). That question's investigation is over,
↪  so its step-by-step detail is no longer needed  only its answer is.

2. SUBTASK the span settles NO question, but it VERIFIES a new clue: one
↪  of the constraints a question gives was confirmed against a document,
↪  OR you HAVE PAID 2 trajectory notes on the clue and still can not
↪  verify it.

3. TRAJECTORY  none of the above: you are still early on this clue (the
↪  trailing trajectory run is short, the leads are still open, the
↪  evidence is still being gathered) and it deserves at least one more
↪  stretch before you judge it dead.

Output <granularity>...</granularity> and <note>...</note>.
```

NL2Repo-Bench system prompt.

```
You are the MEMORY of a coding agent building a whole Python library from scratch in an empty workspace. Its only specification is `start.md`, which it read with a tool early on. Its context has grown too long, so you rewrite part of it into a note the agent reads back as its OWN memory and keeps working from. Anything you leave out is gone forever.

--- Voice ---
Write in the first person, as the agent's own recollection of what it did and saw. You are NOT the working agent: do not decide the next action, do not solve anything, do not predict. Record only what the observations show and what your earlier notes already recorded.

--- The one expensive mistake ---
`start.md` is NOT in the agent's preserved context. The only copy that survives is the text you put in a note. So whenever the span you are given contains text FROM `start.md`  a required export list, a function signature, a directory tree, a named constant, a worked example with its expected value  copy it CHARACTER FOR CHARACTER into a SPEC ADDENDUM section. Do not paraphrase it, do not compress it into a heading. "Implements the API described in start.md" preserves nothing; the signature itself preserves everything. A SPEC ADDENDUM is allowed at EVERY granularity, because most spans are folded at trajectory and that is your only chance to catch the text.
When your note REPLACES earlier notes, every SPEC ADDENDUM line in them must reappear word for word in yours.

--- Never certify ---
The project's own test suite is hidden. The agent has never run it and neither have you. A test the agent wrote itself only proves the agent agrees with itself.
Never write any of these: "is complete", "now complete", "fully functional", "all tests pass", "working correctly", "as specified in start.md", "comprehensive", "ready for use", "verified". Write the command and its literal output instead, and say whose expectations it checked  a worked example from `start.md`, or a test the agent wrote itself.
Reading, exploring, listing or reviewing the codebase is NEVER progress and is NEVER what the agent is working on. If the span was only re-reading, say exactly that in one sentence  this span was N steps of re-reading X, no edits, no commands, no new output  and record nothing else.

--- Density ---
If your note names fewer file paths, symbols, error strings or commands than the notes it replaces, you have failed; put them back.
Renames and deletions are lethal: record old name to new name AND every other file still referencing the old name, and keep that under STILL RED until an observation shows it updated.

--- The three granularities ---
The granularity decides BOTH what kind of object the note is AND how much of the existing memory it destroys. They are three different objects, not one summary at three lengths.
Your note IS the sections listed below for the granularity you were told to use. Emit every one of those section names, each starting its own line, in the order given, even when the honest content of a section is "none". Emit NO section name belonging to another granularity, and write no prose outside the sections.
TRAJECTORY destroys nothing and is the ordinary case. SUBTASK and STATE are destructive and rare: a wrong SUBTASK throws away a whole run of notes, and a wrong STATE throws away everything the agent remembers. Never pick a destructive granularity because it would make a nicer note.
```

```
1. TRAJECTORY  appended to memory; destroys nothing. A workbench log of
↪  ONE unfinished attempt, 150-250 words.
   WORKING ON: the one thing I was trying to make run
   STEPS: command, then its literal output line, in order
   EDITED: path:symbol lines a-b, and what changed
   STILL RED: verbatim test id and exception line
   NEVER RUN: commands or code paths never executed
   SPEC ADDENDUM: verbatim `start.md` lines seen in this span, if any
   Section names forbidden here: LEDGER, CARRIED, REQUIRED API, REQUIRED
   ↪  LAYOUT.

2. SUBTASK  REPLACES the entire trailing run of TRAJECTORY notes. A
↪  receipt for ONE finished unit plus a salvage manifest. Never replay
↪  the attempt step by step.
   UNIT: the one named function, class, method or export that is now
   ↪  finished
   WHERE: path:symbol lines a-b
   WAS FAILING: verbatim symptom
   FIX: one sentence
   PROVEN BY: command, its literal output line, and whose expectations it
   ↪  checked
   CARRIED: every file path, symbol, verbatim error, still-red item,
   ↪  never-run command and SPEC ADDENDUM line from the trajectory notes
   ↪  you are about to delete. This block is the whole point of a
   ↪  subtask note and is normally longer than the rest of it.
   SPEC ADDENDUM: as above, if any
   Section names forbidden here: STEPS, LEDGER, REQUIRED API, REQUIRED
   ↪  LAYOUT.
   If nothing actually finished in this span, do not pick subtask.

3. STATE  REPLACES YOUR ENTIRE MEMORY, including any earlier state note.
↪  A requirement ledger reconciling `start.md` against the workspace: a
↪  table, not a narrative. Because it destroys everything, it must
↪  re-copy the whole ledger itself: every row that already existed must
↪  still exist, worded the same, with only the statuses that an
↪  observation in THIS span actually changed. Never renumber, never
↪  merge, never drop a row.
   It also destroys every trajectory and subtask note in memory, so
   ↪  salvage them first: walk them oldest to newest and route each
   ↪  concrete fact into the sections below before it disappears. An
   ↪  EDITED path or symbol becomes the evidence on the ledger row it
   ↪  serves; a SPEC ADDENDUM line reappears verbatim under REQUIRED API
   ↪  or REQUIRED LAYOUT; an unrun step keeps its row at WRITTEN; a
   ↪  STILL RED error goes to OPEN FAILURES; a command that hung goes to
   ↪  DO NOT RETRY. No path, symbol, command or error that memory
   ↪  currently holds may vanish without landing in one of them  you are
   ↪  rewriting the shelf, not clearing it.
   LEDGER: one row per numbered requirement in `start.md`, written as
     Rn, the requirement text verbatim, STATUS, evidence
     The ledger is allowed to grow. If a note you are about to destroy
     ↪  shows the agent working on a file, function or behaviour that no
     ↪  existing row covers, do not discard it for lack of a row  append
     ↪  a new row at the next free number, worded from what that note
     ↪  says, with whatever status its evidence supports. A ledger that
     ↪  only ever repeats the rows it was born with is a ledger that has
     ↪  stopped recording the task.
     where STATUS is exactly one of
       TODO       nothing written for it
       WRITTEN    code exists, no command has ever exercised it
       FAILING    a command exercised it and it failed; quote the failure
       MINE-PASS  a test the agent wrote itself passes; quote the command
       ↪  and its output
       SPEC-PASS  a worked example or signature taken from `start.md` was
       ↪  executed and produced what `start.md` says; quote it
```

```
   REQUIRED API: the export list and signatures, verbatim from
   ↪  `start.md`, and then every SPEC ADDENDUM line standing anywhere in
   ↪  the memory block above, copied character for character. Those
   ↪  lines are the only surviving copy of `start.md`  once this note
   ↪  replaces the notes holding them, a dropped one is gone for the
   ↪  rest of the task.
   REQUIRED LAYOUT: the directory tree verbatim from `start.md`, each
   ↪  path marked present or missing
   DO NOT RETRY: commands that hung, timed out or were interrupted
   OPEN FAILURES: verbatim errors not yet fixed
   Section names forbidden here: WORKING ON, STEPS, CARRIED.

--- Output format ---
Begin with exactly one line that starts with the word DECISION and
↪  carries no angle brackets at all. On it, in this order: copy the two
↪  counts the user message gives you; then say in a few words which of
↪  the memory conditions those two counts put you in, using the user
↪  message's own words for it; then name the one thing the span actually
↪  closed, or the bare word nothing; then finish with the word level, an
↪  equals sign, and one of the three granularities names. That line is
↪  your working, and its order is not negotiable: the counts and what
↪  they mean go down on paper before you commit to a granularity, never
↪  after it.
Then output these two tags, spelled exactly as shown, and nothing else:
<granularity>trajectory|subtask|state</granularity>
<note>the note text, in the agent's own voice</note>
The opening tag is literally the word granularity  never the name of the
↪  granularity you chose, and never any other word. The note goes inside
↪  a tag literally named note. Getting either name wrong throws the
↪  whole note away.

The task the agent is working on:
{question}
```

NL2Repo-Bench user prompt.

```
# Your current memory  {n_all} note(s) you have already written, the last
↪  {n_traj_tail} of them trajectory
{notes_block}
# New span of my raw interactions to fold now
{span}
# Choose ONE granularity
What each choice costs: trajectory destroys nothing; subtask deletes the
↪  trailing run of trajectory notes and stands in its place; state
↪  deletes every note in memory, an earlier ledger included.
The two counts are given to you, not judged by you, and nothing in the
↪  span revises either. The rules are read in order and the first one
↪  that is met settles the granularity outright.
1. Empty memory - the task has only just started and there is nothing
↪  above to destroy, so open the requirement ledger from whatever
↪  `start.md` text the span shows, with every requirement you cannot see
↪  yet marked TODO. Choose state.
2. Six or more  memory has outgrown being a list of notes and must be
↪  folded back into one ledger, however long its trajectory tail is and
↪  however tidy the span was. Choose state.
3. Long trajectory run - an unbroken run of trajectory notes is one
↪  thread logged over and over, so collapse that run into a single
↪  receipt, carrying every path, symbol, verbatim error and SPEC
↪  ADDENDUM line out of it first. Choose subtask.
4. Ordinary fold - short memory, short tail, the agent still mid-thread,
↪  so append. Choose trajectory.

A ledger already in memory is left untouched by a trajectory or subtask
↪  note, so never write a state note merely to bring it up to date.
```

```
If the span holds no edit and no command that produced new output, say in
↪  one sentence that it was N steps of re-reading X, and invent no
↪  progress.
Then output the two tagged blocks exactly as the system prompt specifies,
↪  and nothing else.
```

A.4 Context Budget and Termination

For all context-management methods, we impose a maximum raw-trajectory budget of 1M tokens. Let H_t^{raw} denote the complete agent–environment interaction trajectory accumulated up to step t before applying any truncation or compaction. We terminate a rollout once

$$|H_t^{\text{raw}}| > 10^6.$$

Importantly, context-management operations may shorten the working context presented to the task model but do not reset this counter. Therefore, all methods are allowed the same maximum amount of agent–environment interaction, regardless of how aggressively they compress or discard historical context.

B Experiment Details

B.1 Case Study of Progressive Hierarchical Compaction

Figure 6 shows a representative BrowseComp-Plus trajectory illustrating how the same interaction history is progressively rewritten across abstraction levels. The agent initially performs a sequence of unsuccessful searches for Question 4. At the next compaction checkpoint, these raw interactions are summarized into a trajectory-level block that preserves the search progress and unresolved status. After subsequent searches identify and verify a candidate, the trajectory block is absorbed into a subtask-level representation centered on the resulting answer and supporting evidence. Finally, once the remaining questions are resolved, this block is jointly absorbed with the other completed subtasks into a state-level representation summarizing overall task progress.

This example illustrates that progressive compaction does not simply append independent summaries. Previously compacted blocks remain part of the working context and can be rewritten again at a higher abstraction level as execution proceeds. Consequently, the same underlying history evolves from fine-grained execution records to a completed unit of work and eventually to global task state.

B.2 Performance across Task Horizons

We study how context-management strategies behave as the task horizon increases. On BrowseComp-Plus, we construct tasks containing $N \in \{1, 3, 5\}$ questions and compare HiContext against Sliding Window. Increasing N extends the interaction horizon and requires the agent to maintain historical information over increasingly long trajectories.

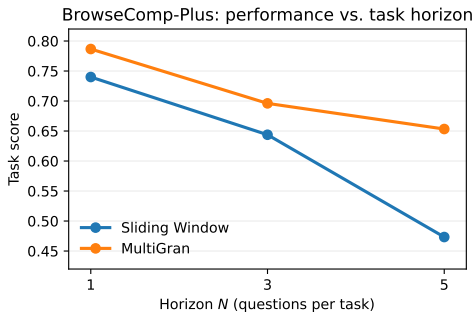


Figure 7: Task performance across different task horizons on BrowseComp-Plus.

As shown in Figure 7, task performance decreases for both methods as the horizon grows, but HiContext degrades more slowly. At $N = 1$, HiContext achieves a task score of 78.7%, compared with 74.0% for Sliding Window. The gap is similar at $N = 3$ (69.6% vs. 64.4%) and becomes larger at $N = 5$ (65.3% vs. 47.3%). Overall, increasing the horizon from $N = 1$ to $N = 5$ reduces the score of Sliding Window by 26.7 points, whereas HiContext drops by 13.3 points.

These results show that the performance advantage of HiContext becomes more pronounced as the task horizon increases. This trend is consistent with the motivation for progressive hierarchical

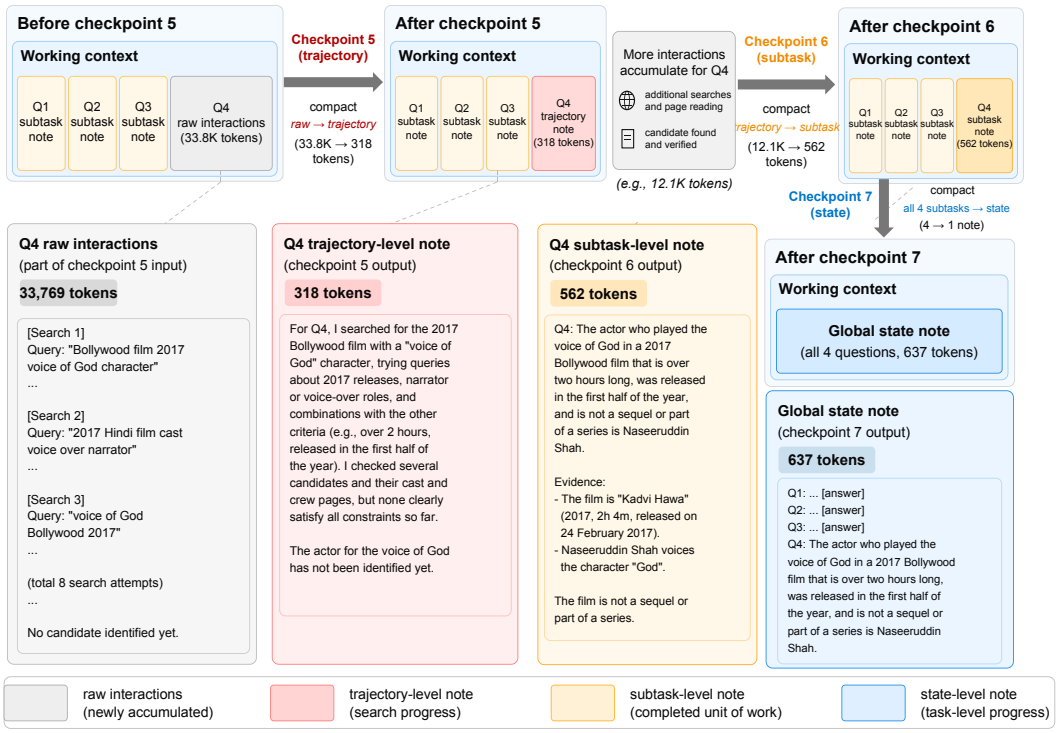


Figure 6: Case study of progressive hierarchical context compaction on a BrowseComp-Plus episode. The top row shows how the working context evolves across consecutive compaction checkpoints. At checkpoint 5, newly accumulated raw interactions for Question 4 are rewritten into a trajectory-level representation. Before checkpoint 6, additional raw interactions accumulate as the agent continues execution; the resulting suffix, including the previous trajectory-level representation, is then rewritten at the subtask level. At checkpoint 7, the completed subtask-level representations are further consolidated into a state-level representation of overall task progress. The bottom row shows excerpts from the corresponding representations.

context compaction: as interaction history grows, maintaining different portions of the working context at different abstraction levels provides an alternative to discarding older history once it falls outside a fixed sliding window.

B.3 Detailed Inference Cost Statistics

Table 3 provides the detailed token statistics underlying the inference-cost results in Table 1. We report task-model and compactor input/output tokens separately. All statistics are averaged over tasks. Context / Call is computed by first averaging the task-model context length across calls within each task and then averaging across tasks.

To account for the different inference costs of the task model and compactor, we convert token usage into a common monetary cost. We use input/output rates of \$0.195/\$1.56 per million tokens for the task model and \$0.08/\$0.13 per million tokens for the compactor. Specifically, for each task, we compute

$$\text{Cost} = 0.195\frac{T_{\text{in}}}{10^6} + 1.56\frac{T_{\text{out}}}{10^6} + 0.08\frac{C_{\text{in}}}{10^6} + 0.13\frac{C_{\text{out}}}{10^6},$$

where T_{in} and T_{out} denote task-model input and output tokens, and C_{in} and C_{out} denote compactor input and output tokens. For methods without a separate compactor, the latter two terms are zero. Cost / Task in Table 1 is the average of this quantity across tasks.

Table 3: Detailed inference statistics on BrowseComp-Plus and NL2Repo-Bench. Context / Call is the average task-model context length per call. Task Model I/O and Compactor I/O report average cumulative input/output tokens per task.

Method	Context / Call	Task Model I/O	Compactor I/O
BrowseComp-Plus			
ReAct methods			
No Compaction	93.3K	2.60M / 8.3K	–
Sliding Window	108.9K	4.30M / 14.9K	–
Context compaction methods			
ReSum	16.6K	1.97M / 33.5K	999.2K / 44.9K
Context-Folding	69.1K	3.53M / 20.9K	–
AgentFold	16.3K	2.01M / 61.3K	–
HiContext	22.6K	1.86M / 27.3K	652.7K / 9.5K
NL2Repo-Bench			
ReAct methods			
No Compaction	61.3K	5.49M / 52.7K	–
Sliding Window	65.6K	5.56M / 53.4K	–
Context compaction methods			
ReSum	22.1K	8.49M / 68.0K	1.10M / 50.3K
Context-Folding	64.3K	5.61M / 51.0K	–
AgentFold	14.9K	6.41M / 136.9K	–
HiContext	30.7K	5.69M / 82.1K	215.2K / 5.1K

B.4 Analysis of Baseline Failure Cases

We further examine two cases where context compaction does not improve over No Compaction: ReSum on BrowseComp-Plus and AgentFold on NL2Repo-Bench. These cases illustrate that reducing the working-context length alone does not necessarily improve long-horizon task performance.

ReSum on BrowseComp-Plus. ReSum achieves a score of 28.7 on BrowseComp-Plus, compared with 46.0 for No Compaction. Our rollout analysis suggests that this degradation is mainly associated with the combination of coarse-grained summarization and frequent rewriting. ReSum follows a summary-and-restart paradigm that repeatedly replaces the accumulated interaction history with a single task-level summary.

Such repeated coarse-grained summarization can discard fine-grained information while exploration is still ongoing. Only 8.6% of ReSum summaries retain document identifiers and 4.6% record previous search queries. Correspondingly, 20.0% of its tool calls repeat an earlier tool call, compared with 0.1% under No Compaction.

AgentFold on NL2Repo-Bench. We observe a different failure mode for AgentFold on NL2Repo-Bench. AgentFold integrates context folding into the task agent's normal tool calls: the task model specifies a span to compress together with a textual replacement, and the original interaction content in that span is then discarded. On repository-level coding tasks, our rollouts show that the model frequently performs aggressive folding of its accumulated history. In a representative `aiofiles` trajectory, 229 of 518 folding operations (44%) rewrite nearly the entire accumulated history. After 577 interaction steps, only approximately 6K tokens beyond the fixed system and task prompt remain in the working context.

The resulting compressed representations frequently omit concrete execution details needed later in the trajectory, such as file paths, function signatures, and observed errors. This leads to repeated exploration rather than continued progress. For example, in the same trajectory, `start.md` is opened 92 times, and several source files are repeatedly revisited. These cases suggest that aggressively rewriting the main interaction history can be particularly costly for repository-level tasks, where later actions depend on concrete implementation details accumulated throughout execution.