LIGHTCODE: DISTILLING EXECUTION CONTEXT FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) agents tackle complex software tasks through extended sequences of reasoning, action, and observation. However, redundancy in incoming observations and accumulated histories inflates cumulative token costs and can impair reasoning, while indiscriminate compression risks discarding information needed later. To address this challenge, we propose LightCode, a framework for context-efficient coding agents that controls how incoming observations are represented and how accumulated histories are retained over successive interaction steps. For incoming observations, self-evolving observation compression uses execution experience to identify recurring redundancy and refine reusable compression programs, extending their coverage across formats while constraining updates to preserve required information. By integrating long-horizon history consolidation, LightCode maintains a compact task state while retaining historical details separately for selective retrieval. Evaluations across three benchmarks and four LLM backbones demonstrate consistent gains in token efficiency, with LightCode reducing token usage by 14–50% while maintaining or improving task performance. Source code is available at: `https://anonymous.4open.science/r/lightcode-research-4664`

1 INTRODUCTION

Large language models (LLMs) have enabled agents that reason, act, and interact with their environments to accomplish complex tasks (Yao et al., 2023). In software engineering, coding agents apply these capabilities to tasks such as resolving issues and implementing changes in a codebase (Yang et al., 2024; Wang et al., 2025). Such tasks unfold through repeated cycles of reasoning, action, and observation. The context available at each step connects these cycles, bringing together task requirements, previous decisions, and environmental feedback to support continued progress.

As execution proceeds, each interaction adds information that may be included in many subsequent model inputs. The resulting overhead depends on both how much information enters the context and how long it remains there. As illustrated in Figure 1, redundancy arises in both incoming observations and accumulated histories. Incoming observations often contain repeated or incidental details beyond what is needed to interpret an outcome. Over time, even initially useful information can become less relevant as questions are answered, intermediate goals are completed, or earlier results are superseded. Retaining these details increases cumulative token costs and adds noise that can obscure relevant information and degrade performance in long-context tasks (Liu et al., 2024).

Prior work reduces context at several levels of granularity. LLMLingua uses a coarse-to-fine procedure to compress prompts under a token budget (Jiang et al., 2023), while LLMLingua-2 distills compression examples into an encoder trained to select which tokens to retain (Pan et al., 2024). RECOMP summarizes retrieved documents before incorporating them into the model's context, using extractive or abstractive compressors trained to balance summary length with downstream performance (Xu et al., 2024). For extended interactions, observation masking removes older observations while retaining recent exchanges, whereas summarization replaces accumulated interactions with compact descriptions (Lindenbauer et al., 2025). MemGPT introduces a hierarchy of active context and external storage, with explicit operations for transferring information between them and retrieving archived content when needed (Packer et al., 2023). However, fixed compression policies do not improve with execution experience, and summaries may omit historical details needed later.

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.

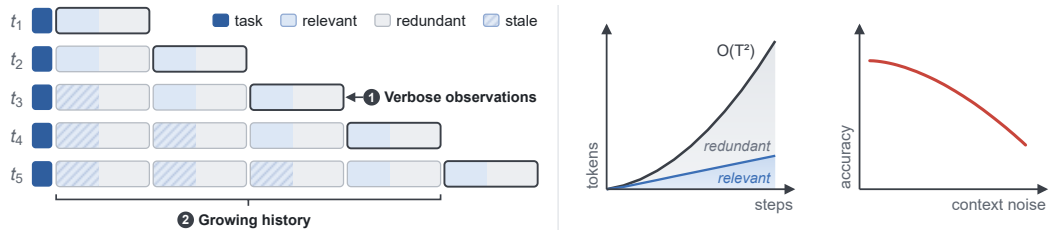

Figure 1: Redundant observations and stale history details are repeatedly included in the context, inflating cumulative token cost (schematic; fixed-size observations retained for T steps).

This leads to two challenges: *i) How should new information be represented?* Downloading a file may produce a long sequence of progress updates—10%, 20%, and so on—before completion. If the next action only depends on whether the download succeeded, a brief final status can replace the entire progress sequence. What can be omitted, however, depends on the form and purpose of the observation. Because the agent acts on compressed observations, its later behavior may not reveal what was omitted. The challenge is to learn reusable compression rules while constraining information loss as their coverage expands. *ii) How should past information be retained?* Once a repair is validated, an earlier failure report may add little to the next decision, yet its details can become relevant again if the problem recurs. Retaining every detail throughout execution incurs unnecessary overhead, while discarding it can disrupt later reasoning. Retention must distinguish the state needed to continue the task from details that may only need to be revisited later.

To address these challenges, we propose LightCode, which distills execution context for coding agents. Across tasks, LightCode evolves reusable compression programs from execution experience, with updates governed by preservation constraints. Within each task, long-horizon history consolidation separates a compact task state from stored observation details. This separation limits repeated inclusion of past observations while supporting selective retrieval. We evaluate LightCode with four LLM backbones on three benchmarks covering coding and long-horizon retention. LightCode reduces token usage by 14–50% while maintaining or improving task performance, with coding-task pass rates increasing by up to 7.5 percentage points. Our contributions are as follows:

- We propose LightCode, a framework for context-efficient coding agents that controls how observations are represented and how interaction histories are retained.
- LightCode introduces two key mechanisms for distilling execution context: i) self-evolving observation compression, which refines reusable compression programs from execution experience under preservation constraints; and ii) long-horizon history consolidation, which decouples task-state abstraction from historical details retained for selective retrieval.
- Evaluations across three benchmarks and four LLM backbones demonstrate that LightCode consistently improves token efficiency without sacrificing overall task performance, reducing token usage by 14–50% and increasing coding-task pass rates by up to 7.5 percentage points.

2 PRELIMINARIES

We consider coding agents that receive a task description and an execution environment, with success evaluated by task-specific checks (Jimenez et al., 2024; Yang et al., 2024). Each *interaction step* consists of one model decision followed by environmental feedback. For each distinct observation representation z_i, let L_i denote the number of steps during which it remains in the active context. The cumulative input-token cost summed over all interaction steps of a task is

$$\mathcal{C}_{\text{ctx}} = \sum_i \underbrace{|z_i|}_{\text{representation length}} \times \underbrace{L_i}_{\text{context residence}} + \mathcal{C}_{\text{other}}. \tag{1}$$

where $|\cdot|$ counts tokens and $\mathcal{C}_{\text{other}}$ covers instructions, actions, summaries, and other context. Observation compression reduces representation length, while history consolidation limits context residence, targeting lower cumulative cost while maintaining task success.

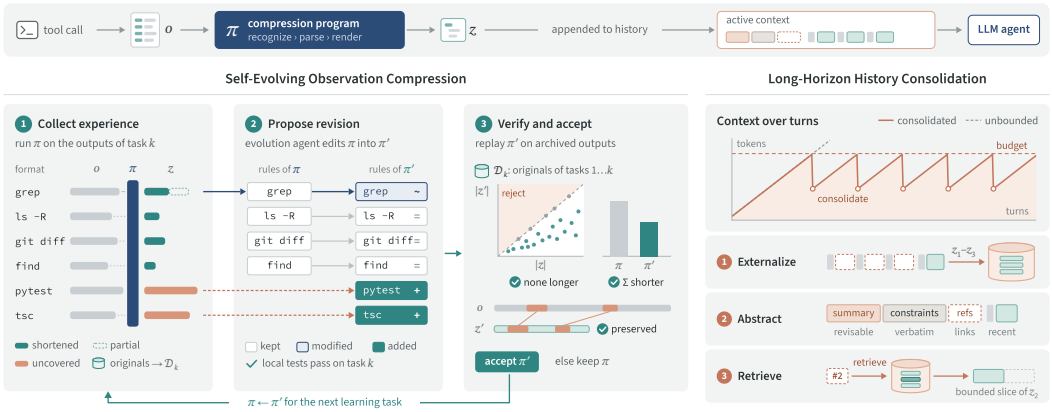

Figure 2: Overall framework of LightCode. Observation compression evolves across learning tasks (left); history consolidation externalizes, abstracts, and retrieves observations within a task (right).

3 METHODOLOGY

LightCode reduces cumulative context cost through self-evolving observation compression and long-horizon history consolidation, as shown in Figure 2. These mechanisms act on the two factors in Eq. 1: shorter representations reduce the contribution of each observation, while history consolidation limits its residence in the active context. Compression knowledge is refined across tasks, while history consolidation updates the representation of task state as execution progresses.

The two mechanisms require different forms of control. An omission in an incoming observation can prevent the agent from recognizing that relevant information exists, making the loss difficult to diagnose from its subsequent behavior. Historical observations, once read, can be stored separately from the goals and decisions that guide continued execution.

3.1 LEARNING TO COMPRESS FROM EXPERIENCE

A useful compression rule captures structure that recurs across executions while retaining distinctions relevant to later decisions. We formulate self-evolution as a search over reusable compression programs. Execution experience guides the search toward uncovered regularities, while verification constrains which revisions are inherited by subsequent tasks.

3.1.1 COMPRESSION AS REUSABLE PROGRAMS

A reusable policy specifies both a transformation and when it applies. Motivated by prior work on rule-based output compression (Szymkowiak, 2026), LightCode represents the policy π as an executable program with three functions: i) recognizing observation formats; ii) parsing their structure; and iii) rendering compact representations. Given an observation o and metadata m describing the action and its arguments, $z = F_\pi(o, m)$ denotes the representation presented to the agent. The policy separates structural interpretation from compact rendering:

- **Recognition and parsing.** Identify which parts of an observation share a common structural role.
- **Transformation.** Express these structural relationships with less repetition.

For example, a search result may repeat a file path before every match. Grouping line numbers and matched text under a single path preserves their locations while removing the repeated prefix.

Action arguments can change the meaning of an output segment, so compression is conditioned on both the producing action and the structure of its result. When the format cannot be recognized or the transformed result fails a validity check, the policy retains the observation unchanged. The initial hand-written rules are extended as learning tasks expose new formats.

Policy revisions can take three forms: i) extending format recognition; ii) refining a parser; or iii) introducing a transformation reusable across observation types. Once adopted, the program applies these decisions deterministically without an additional model inference for each observation.

3.1.2 Evolution under Preservation Constraints

Optimizing compression directly from task outcomes provides an incomplete learning signal. Success or failure depends on many decisions beyond the representation of a particular observation, and a successful trajectory does not establish that omitted content was dispensable. LightCode therefore separates the generation of promising revisions from their acceptance, using completed trajectories to guide program changes and original observations to evaluate those changes.

Experience-driven refinement. After a learning task, an evolution agent receives the current policy along with: **i) the trajectory**, providing the execution context; **ii) observation pairs**, containing original and compressed content; and **iii) policy decisions**, recording how each representation was produced. Original observations are archived separately during execution. These inputs expose uncovered formats, residual redundancy, and content omitted from the acting agent's view.

Execution context helps distinguish a transferable rule from a shortening that happens to work on one observation. Similar text may represent different kinds of content under different action arguments, and repeated entries can refer to distinct locations. The learner therefore considers the conditions that justify a reduction together with the reduction itself, so that a proposed generalization can be examined and tested on other recorded observations. In search results, the shared path structure transfers across tasks, whereas the relevance of particular matches may change.

The learner translates recurring patterns into program revisions. An uncovered format can motivate a new rule; a partially effective rule can motivate a revised parser or a more compact rendering. The learner's local tests support candidate development on the current experience; acceptance separately evaluates the growing collection $\mathcal{D}_k$ of original observations and their metadata from learning tasks through task k. This separation keeps a locally promising revision from being adopted solely on the cases that motivated it. For efficiency updates, the replay objective is

$$\min_{\pi'} \sum_{(o,m)\in\mathcal{D}_k} \left|F_{\pi'}(o,m)\right| \quad \text{subject to preservation requirements.} \tag{2}$$

This objective targets the representation-length factor of Eq. 1. Recorded observations receive equal weight because their residence in future executions is unknown. Comparing policies on shared inputs isolates changes in compression from changes in the agent's subsequent behavior.

Preservation-constrained acceptance. Efficiency alone would favor aggressive deletion, including content whose importance becomes apparent only later. We restrict candidate policies using explicit preservation requirements: input-specific conditions identifying content that must remain available after compression, such as diagnostic messages, exact source literals, and application output that resembles removable formatting. These requirements are specified independently of the acting agent's compressed view. A candidate satisfies a requirement by retaining the specified content or by leaving the input unchanged. Requirements are defined by the system designers and remain fixed within each update, but may be extended between learning tasks as new failures emerge.

Candidates that meet the preservation requirements are compared with the current policy on the accumulated replay set. An efficiency update must satisfy both criteria:

- **Aggregate improvement.** The total representation length over $\mathcal{D}_k$ must decrease, advancing the compression objective in Eq. 2 on the accumulated observations.
- **Per-observation stability.** No recorded observation may become longer under the candidate. This prevents gains on new formats from masking regressions on earlier inputs.

If no candidate satisfies both criteria, the existing policy is retained for subsequent execution.

When a newly added requirement invalidates the current policy, acceptance prioritizes restoring that requirement even if more content must be retained. Appendix A.3 details the acceptance criteria for efficiency updates and preservation repairs.

3.2 Decoupling Task State from Historical Detail

An interaction history serves two purposes: it records the state needed to continue a task and contains details that may support later reconsideration. Goals, decisions, and unfinished work must remain readily available, whereas an earlier observation may be consulted only if a new result makes it relevant again. We use *history consolidation* to denote the reorganization of accumulated interactions into a compact task state with references to stored observations. This separates access to a detail from its repeated inclusion in the model's input. The procedure remains fixed across tasks while maintaining the evolving state of the current execution.

Observation externalization and abstraction. When context exceeds its budget, an older segment is consolidated while recent interactions remain intact. The replacement has three components:

- *Task summary.* Goals, established findings, decisions, rejected approaches, and unfinished work.
- *Exact constraints.* User requirements preserved verbatim outside the generated summary.
- *Historical references.* Links that keep earlier observations accessible outside the active context.

Findings and plans can be revised, whereas an explicit user requirement should not change merely because the history is summarized again. References identify the observations behind earlier conclusions, enabling reconsideration without including those observations in the summary.

Summarization alone may omit or alter historical content, while adding references afterward still leaves the summarizer processing full observations. We therefore externalize eligible observations before abstraction. Each stored observation must match the representation z originally shown to the agent and remain retrievable; its position in the history is replaced by a reference and a short excerpt. The reduced history preserves the order of actions and reasoning, together with observations that cannot be externalized. A summarizer condenses this shorter input into the task summary, while references preserve access to the stored details, as illustrated in Figure 7 in Appendix A.1.

Selective retrieval over long horizons. The acting agent selects historical observations according to the information needed for its next decision. When the task summary lacks a required detail, the agent uses the summarized findings and decisions to identify which observation to revisit, then queries the corresponding reference. Only the requested, length-limited content is added to the active context, leaving unrelated history externalized. References are carried forward and rechecked as the task summary changes, preserving this access while their stored observations remain available.

4 Experiments

We evaluate LightCode with four LLM backbones on the software engineering domain of EvoAgentBench (EvoAgentBench-SE), LongCLI-Bench, and MemoryAgentBench through five research questions: **RQ1:** How effective is LightCode at reducing token consumption without compromising task success? **RQ2:** How well does the evolved policy generalize to longer tasks from an unseen benchmark? **RQ3:** How much does each component contribute to LightCode's performance? **RQ4:** How well does LightCode retain information over long histories? **RQ5:** How does the policy evolve, and do the accepted revisions respect the preservation constraints?

4.1 Experimental Setup

To rigorously isolate the impact of the context manager, all methods run in DeepSeek Harness (DeepSeek, 2026) with an identical agent loop, tool set, and evaluation protocol. For each backbone, the compression policy is evolved from the hand-written initial rules π_0 on all 87 training tasks of the official EvoAgentBench-SE split (Gao et al., 2026) and then frozen as π_k; history consolidation involves no learning. The frozen policy is evaluated on all 56 EvoAgentBench-SE test tasks, applied without further evolution to all 20 LongCLI-Bench tasks (Feng et al., 2026), and assessed for long-horizon retention on five MemoryAgentBench tasks (Hu et al., 2026). Baselines are the native compaction of DeepSeek Harness (DSH Basic), token-level observation compression (LLMLingua-2 (Pan et al., 2024)), periodic history summarization (Periodic Summary), observation

Table 1: EvoAgentBench-SE results on all 56 test tasks: pass rate (mean ± SD over three runs) and average tokens per task; bold marks the best pass rate and lowest total per backbone.

Model	Method	Pass rate (%)	Input (avg.)	Output (avg.)	Total (avg.)
DeepSeek V4 Flash	DSH Basic	83.33 ± 1.03	1,397,909	23,291	1,421,199
	Periodic Summary	79.76 ± 3.72	1,213,129	26,396	1,239,524
	LLMLingua-2	76.79 ± 4.72	1,933,826	32,289	1,966,115
	LightCode	$\mathbf{85.12 \pm 1.03}$	854,867	22,872	**877,739**
MiniMax-M3	DSH Basic	78.57 ± 1.79	1,691,447	16,899	1,708,346
	Periodic Summary	63.10 ± 2.73	538,905	8,376	**547,281**
	LLMLingua-2	50.60 ± 9.83	1,714,508	13,882	1,728,390
	LightCode	$\mathbf{79.17 \pm 3.72}$	1,300,062	23,055	1,323,117
GLM-5.3 Flash	DSH Basic	72.62 ± 5.74	845,008	16,892	861,901
	Periodic Summary	65.48 ± 2.73	841,557	20,117	861,674
	LLMLingua-2	66.67 ± 2.73	1,131,135	20,583	1,151,718
	LightCode	$\mathbf{77.38 \pm 3.72}$	671,401	15,602	**687,002**
Qwen3.8 Flash	DSH Basic	$\mathbf{84.52 \pm 2.06}$	1,843,172	31,109	1,874,281
	Periodic Summary	83.93 ± 3.09	1,714,595	42,505	1,757,099
	LLMLingua-2	82.74 ± 2.73	2,258,312	33,943	2,292,255
	LightCode	$\mathbf{84.52 \pm 2.73}$	1,588,815	27,071	**1,615,886**

masking (Complexity Trap (Lindenbauer et al., 2025)), and guideline-based compression of observations and history (ACON (Kang et al., 2026)). We report task success rate and total model tokens, including cache reads and auxiliary calls (Appendix A.4).

4.2 Performance on EvoAgentBench-SE (RQ1)

To evaluate the effectiveness of LightCode in reducing token consumption while preserving task success, we compare it with DSH Basic and two single-factor baselines, Periodic Summary and LLMLingua-2, on all 56 EvoAgentBench-SE test tasks with four backbones. Periodic Summary restricts observation residence but loses retrievability, while LLMLingua-2 compresses representations without preserving their structure. Table 1 presents the results.

Comparison with DSH Basic. Across all four backbones, LightCode matches or surpasses the pass rate of the native manager while using 14–38% fewer tokens per task. With DeepSeek V4 Flash, it attains the highest pass rate (85.12% vs. 83.33%) with 38% fewer tokens, a reduction almost entirely in input tokens, showing that shortened observations and bounded residence remove content the agent does not revisit rather than content it needs. With GLM-5.3 Flash, MiniMax-M3, and Qwen3.8 Flash, it gains 4.76%, 0.60%, and 0% while saving 14–23%.

Comparison with single-factor baselines. Both Periodic Summary and LLMLingua-2 underperform DSH Basic in pass rate on every backbone. Periodic Summary reduces token cost on three backbones and leaves it unchanged on GLM-5.3 Flash, yet incurs an accuracy penalty on all four, indicating that summarization without retrievable observations discards details required later. LLMLingua-2 is both less accurate and more expensive: its token-level omissions are invisible to the agent, which recovers the missing context through additional interactions that outweigh the per-observation saving. These findings show that reducing context pays off only when the omitted content remains recoverable, which is what preservation constraints and retrievable history provide.

4.3 Generalization to LongCLI-Bench (RQ2)

To assess the generalization of the evolved policy beyond the tasks on which it was evolved, we apply the frozen policy π_k to all 20 LongCLI-Bench tasks without further evolution and compare it with DSH Basic and the initial rules π_0, whose shared consolidation procedure isolates the effect of the accepted revisions. As shown in Table 2, π_k matches or surpasses DSH Basic in pass rate on all four backbones while reducing token consumption by 25–50%, and it matches or exceeds π_0 in accuracy with an additional 12–27% token reduction, whereas π_0 alone yields inconsistent accuracy, from a 10% deficit to a 7.5% gain over DSH Basic. These results demonstrate strong generalization

Table 2: LongCLI-Bench results on all 20 tasks (two-run average) with the initial rules π_0 and the evolved policy π_k; bold marks the best pass rate and lowest total per backbone.

Model	Method	Pass rate	Input (avg.)	Output (avg.)	Total (avg.)
DeepSeek V4 Flash	DSH Basic	**42.5%**	7,731,810	60,219	7,792,029
	LightCode (π_0)	32.5%	6,512,921	141,381	6,654,302
	LightCode (π_k)	**42.5%**	5,543,756	115,658	**5,659,415**
GLM-5.3 Flash	DSH Basic	32.5%	13,243,000	37,000	13,280,000
	LightCode (π_0)	**40.0%**	8,755,500	34,000	8,789,500
	LightCode (π_k)	**40.0%**	6,576,000	38,500	**6,614,500**
MiniMax-M3	DSH Basic	25.0%	27,677,026	78,232	27,755,258
	LightCode (π_0)	27.5%	22,158,288	83,876	22,242,164
	LightCode (π_k)	**30.0%**	16,143,948	55,791	**16,199,739**
Qwen3.8 Flash	DSH Basic	42.5%	12,955,458	99,931	13,055,388
	LightCode (π_0)	42.5%	10,993,833	92,225	11,086,058
	LightCode (π_k)	**47.5%**	9,673,856	84,418	**9,758,274**

of the evolved policy: the accepted revisions capture recurring structural properties of tool outputs rather than task-specific patterns, so the policy retains its accuracy and efficiency advantages on longer tasks from an unseen benchmark without re-evolution. The savings even exceed those on EvoAgentBench-SE, since unrecognized formats are left intact while longer executions extend the residence over which each compressed observation accrues savings.

4.4 Ablation Study (RQ3)

To validate the contribution of each component, we remove one at a time from the full system with DeepSeek V4 Flash and MiniMax-M3: F_{π_k} only removes history consolidation and retains the native compaction of DSH Basic; Ψ only removes observation compression; $F_{\pi_0} + \Psi$ removes evolution by using the initial rules. As shown in Figure 3, two key findings emerge.

Both mechanisms provide complementary savings. The Ψ only variant, which removes observation compression, increases tokens per task by 54% and 22% on DeepSeek V4 Flash and MiniMax-M3, showing that format-aware shortening of tool outputs remains necessary even when the history is consolidated, since consolidation bounds how long an observation stays but not how much it costs while present. The F_{π_k} only variant, which removes history consolidation, increases tokens by 40% and 11%, indicating that shortened observations still accumulate until native compaction. The ordering of the two effects mirrors the structure of the cost: the larger saving comes from shortening what enters the context, the smaller from bounding how long it stays.

Evolution turns cost saving into accuracy gain. The $F_{\pi_0} + \Psi$ variant, which retains consolidation but reverts to the initial rules, already uses 30% and 15% fewer tokens than DSH Basic; the evolved policy saves a further 12% and 9% while raising the pass rate by 2.38% and 1.79%, reaching the highest accuracy of all configurations on both backbones. Since most adopted revisions refine existing rules under preservation requirements rather than add new ones, evolution repairs hand-written rules that dropped needed content while extending coverage to formats they leave uncompressed, so the additional compression removes only content the agent does not need.

4.5 Information Retention on MemoryAgentBench (RQ4)

To evaluate long-horizon information retention, we compare LightCode with DSH Basic, LLMLingua-2, Complexity Trap, and ACON on MemoryAgentBench, whose five tasks cover accurate retrieval, test-time learning, long-range understanding, and selective forgetting over histories exceeding the context budget. As shown in Figure 4 and Table 3, two key findings emerge.

LightCode achieves the best retention at lower cost. The envelope of LightCode in the radar plots encloses those of all baselines on nearly every axis for all four backbones, yielding the highest Micro and Macro scores while consuming the fewest tokens of all methods. LLMLingua-2 collapses on

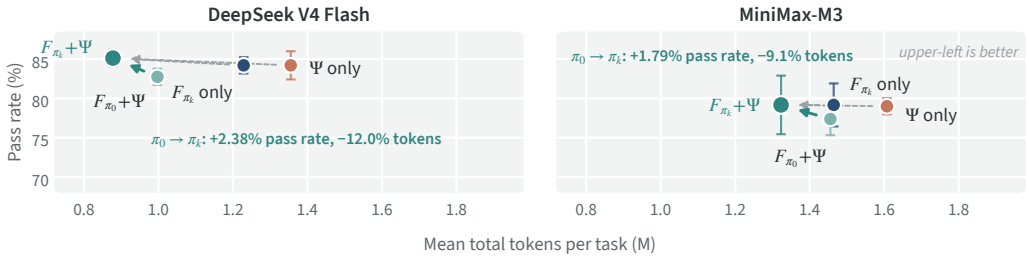

Figure 3: EvoAgentBench-SE ablations: pass rate (mean ± SD) versus mean tokens per task. Each variant removes one component from $F_{\pi_k} + \Psi$; arrows point to the full system.

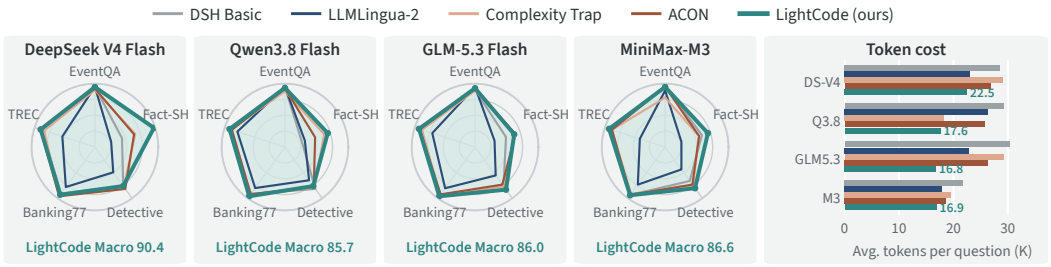

Figure 4: MemoryAgentBench results. Radar plots show task accuracy (%) per backbone (rings every 25%; LightCode filled); bars show average tokens per question.

four of the five axes, as token-level pruning discards the content on which later questions depend, whereas ACON, the strongest baseline, tracks LightCode closely but at a 10–57% higher token cost.

The gains are concentrated on selective forgetting. On Fact-SH32K, which requires recalling the current value of facts revised over a long history, LightCode leads on every backbone except GLM-5.3 Flash and reaches 98.0% with DeepSeek V4 Flash against 67.0% for the strongest baseline. We attribute this advantage to externalization before abstraction: an irreversible manager fixes a paraphrase at compression time, whereas LightCode retains observations behind references that are rechecked whenever the summary is regenerated. These findings demonstrate that history consolidation preserves access to information after it leaves the active context, which, rather than the volume of history retained, distinguishes LightCode from irreversible managers.

4.6 Policy Evolution Analysis (RQ5)

To examine how the policy evolves and whether the accepted revisions respect the preservation constraints, we analyze the recorded evolution with DeepSeek V4 Flash in Figure 5. During learning (a), the share of observations handled by the policy rises from 38% over the first 20 tasks to 68% thereafter (partly reflecting the repository order of the tasks), and 48 of the 67 adopted revisions refine existing rules rather than add new ones, so coverage grows mainly by refinement. Among the 405 acceptance checks (b), 148 candidates are rejected for yielding no aggregate reduction, 8 for lengthening an observation, and 9 for violating a preservation requirement, confirming that acceptance filters both ineffective and unsafe revisions. On the test tasks (c), the 22 rules added during evolution account for 82% of the tokens saved by the frozen policy and 21 of them fire on tasks never seen during learning, with the saving concentrated on a few recurring formats. These results demonstrate that evolution expands the policy under binding constraints and that the accepted rules capture formats recurring across tasks rather than task-specific patterns, which underlies the cost and accuracy gains of the evolved policy. The other three backbones exhibit the same pattern with fewer additions (Appendix A.6).

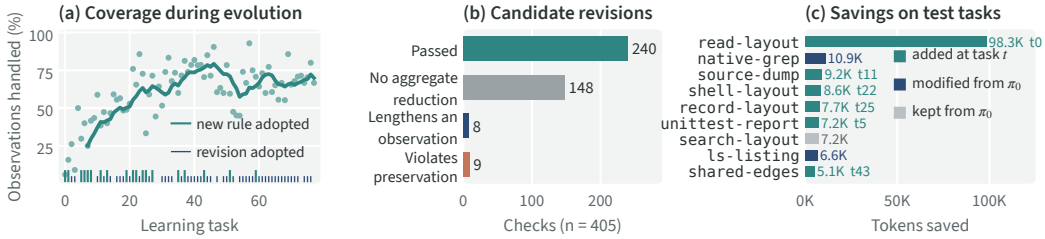

Figure 5: Policy evolution with DeepSeek V4 Flash: (a) observation coverage per learning task and adopted revisions; (b) acceptance-check outcomes; (c) test-task token savings per rule, by origin.

5 Related Work

5.1 LLM-Based Coding Agents

LLM-based agents extend language generation into interactive problem solving by interleaving reasoning with actions, learning when to invoke external functions, and expressing actions as executable code (Yao et al., 2023; Schick et al., 2023; Wang et al., 2024). For software engineering, agent harnesses and structured workflows support repository navigation, code localization, editing, and validation through specialized interfaces and program-aware retrieval (Yang et al., 2024; Wang et al., 2025; Xia et al., 2025; Zhang et al., 2024). Multi-agent systems distribute planning, implementation, and verification across specialized roles coordinated through structured communication (Hong et al., 2024; Qian et al., 2024; Wu et al., 2024). Feedback also supports iterative solution refinement and reflection (Madaan et al., 2023; Shinn et al., 2023); beyond individual solutions, automated agent design and self-evolution use empirical evaluation to revise executable components and control flow (Hu et al., 2025; Zhang et al., 2026). Across these designs, repeated processing of observations and interaction histories remains a source of execution overhead.

5.2 Context Management for LLM Agents

Context management spans input compression and selective historical retention. Information-based and query-aware selection remove less useful content before inference (Li et al., 2023; Jiang et al., 2023; 2024), while learned approaches compress inputs through token selection, document summarization, or reusable latent representations (Pan et al., 2024; Xu et al., 2024; Mu et al., 2023). Repository-oriented systems combine structural code representations with selective retrieval to supply relevant code and development state (Li et al., 2026; 2025). For longer interactions, hierarchical storage, relevance-based retrieval, and memory updating organize information beyond the immediate context (Packer et al., 2023; Zhong et al., 2024; Xu et al., 2025). Observation masking and trajectory summarization limit retained history (Lindenbauer et al., 2025); adaptive approaches learn to fold histories at multiple granularities (Ye et al., 2026) or refine guidelines for compressing observations and histories using execution feedback (Kang et al., 2026). Balancing compactness with information preservation remains a shared challenge in long-horizon agent execution.

6 Conclusion

We presented LightCode, which distills execution context into compact observation representations and task state. Self-evolving observation compression converts execution experience into reusable compression knowledge, refining executable programs under preservation constraints as new formats and recurring structures are encountered. Long-horizon history consolidation separates the evolving task state from historical details, preserving continuity of execution while allowing earlier observations to be revisited selectively. Evaluations across three benchmarks and four LLM backbones demonstrate reductions in token usage alongside maintained or improved task performance.

AI USE STATEMENT

Generative AI tools were used in this work primarily to assist with language translation and editing for clarity and readability, paper formatting, and the preparation and refinement of some figures. All AI-assisted content was reviewed and, where necessary, revised by the authors. The authors take full responsibility for the final content of the paper, including all text, claims, results, and figures.

REFERENCES

DeepSeek. DeepSeek Harness: Everything is a plugin. Official project documentation, 2026. URL https://deepseek.com/harness/en/. Developer preview; accessed September 22, 2026.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, Jie Sun, Yang Xiao, Sizhuo Zhou, Wenxiao Wu, Yiming Liu, Pengfei Liu, Shenglin Zhang, and Kaipeng Zhang. LongCLI-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 29952–29963, 2026. doi: 10.18653/v1/2026.findings-acl.1497. URL https://aclanthology.org/2026.findings-acl.1497/.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, Yafeng Deng, and Teng Li. EvoAgentBench: Benchmarking agent self-evolution via ability transfer. arXiv preprint arXiv:2607.05202, 2026. URL https://arxiv.org/abs/2607.05202.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in LLM agents via incremental multi-turn interactions. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=DT7JyQC3MR.

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

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. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.91. URL https://aclanthology.org/2024.acl-long.91/.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. ACON: Optimizing context compression for long-horizon LLM agents. In *International Conference on Machine Learning*, 2026.

Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 6342–6353. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.391. URL https://aclanthology.org/2023.emnlp-main.391/.

Zhonghang Li, Zongwei Li, Yuxuan Chen, Han Shi, Jiawei Li, Jierun Chen, Haoli Bai, and Chao Huang. FastCode: Fast and cost-efficient code understanding and reasoning. *arXiv preprint arXiv:2603.01012*, 2026. URL https://arxiv.org/abs/2603.01012.

Zongwei Li, Zhonghang Li, Zirui Guo, Xubin Ren, and Chao Huang. DeepCode: Open agentic coding. *arXiv preprint arXiv:2512.07921*, 2025. URL https://arxiv.org/abs/2512.07921.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management. In *Fourth Deep Learning for Code Workshop (DL4C), co-located with NeurIPS 2025*, 2025.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9/.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://doi.org/10.52202/075280-2019.

Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://doi.org/10.52202/075280-0848.

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

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. 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. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.57. URL https://aclanthology.org/2024.findings-acl.57/.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://doi.org/10.52202/075280-2997.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://doi.org/10.52202/075280-0377.

Patrick Szymkowiak. RTK: Rust token killer. Open-source software, 2026. URL https://github.com/rtk-ai/rtk. Accessed September 26, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel 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. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE): FSE037:1–FSE037:24, 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. Recomp: Improving retrieval-augmented lms with context compression and selective augmentation. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 43478–43502, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/bda88ed2892f5e61c9a9bf215c566913-Paper-Conference.pdf.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://doi.org/10.52202/085713-0593.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

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*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

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

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=pUpzQZTvGY.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2024. doi: 10.1145/3650212.3680384.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

A Appendix

A.1 Algorithm and Notation

Algorithm 1 gives the complete LightCode procedure; Figures 6 and 7 provide additional examples. In the algorithm, p_θ is the acting LLM, C is the active context, $\mathcal{R}$ is the reference store for previously visible observations, and $\mathcal{A}$ archives original observations for policy evolution. The consolidation procedure is denoted by Ψ, and $\widehat{T}$ estimates token counts. The experience $\mathcal{E}_k$ includes the completed trajectory, original and compressed observation pairs, and the corresponding policy decisions. Preservation requirements $\mathcal{P}_k$ remain fixed within each update and may be extended between learning tasks.

Algorithm 1 LightCode

Require: learning tasks $x_0, \ldots, x_{K-1}$; initial policy π_0; preservation requirements $\mathcal{P}_0$; context budget τ

$\mathcal{D}_{-1} \leftarrow \varnothing$
for $k = 0, \ldots, K-1$ **do**
  $C \leftarrow$ description of x_k; $\mathcal{A} \leftarrow \varnothing$; $\mathcal{R} \leftarrow \varnothing$
  while x_k is not finished **do**
    $a_t \sim p_\theta(\cdot \mid C)$; $(o_t, m_t) \leftarrow \mathrm{Exec}(a_t)$ ▷ retrieval requests are actions answered from $\mathcal{R}$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{(o_t, m_t)\}$; $C \leftarrow C \oplus \big(a_t,\ F_{\pi_k}(o_t, m_t)\big)$ ▷ observation compression
    if $\widehat{T}(C) > \tau$ **then**
      $C \leftarrow \Psi(C)$ ▷ history consolidation using $\mathcal{R}$
  $\mathcal{D}_k \leftarrow \mathcal{D}_{k-1} \cup \mathcal{A}$; $\mathcal{P}_k \leftarrow$ requirements in force after task x_k
  $\pi_{k+1} \leftarrow \mathrm{Verify}\big(\mathrm{Propose}(\pi_k, \mathcal{E}_k);\ \pi_k, \mathcal{D}_k, \mathcal{P}_k\big)$ ▷ evolution
return $\pi_\star = \pi_K$, frozen for evaluation

	A. Removable progress	**B. Possible application output**
Action	`pytest -rsx -q tests/test_cli.py | tail -2`	`pytest -rP -q tests/test_cli.py | tail -2`
Observation o	`.......... [100%]` `1 passed in 0.03s`	`..........` `1 passed in 0.03s`
Observed z	Progress line removed; result line kept.	Both lines kept.
Reason	The report selector `-rsx` does not print captured output, so the dots are progress.	`-rP` prints captured output, and the truncated window hides the header that would identify the dots.
In LightCode	Expressed by the policy's parsing of the command and report structure.	Encoded as a preservation requirement that every revision must satisfy.

Figure 6: Two similar observations with opposite compression decisions: progress dots are removable in A but may be application output in B, which a preservation requirement protects.

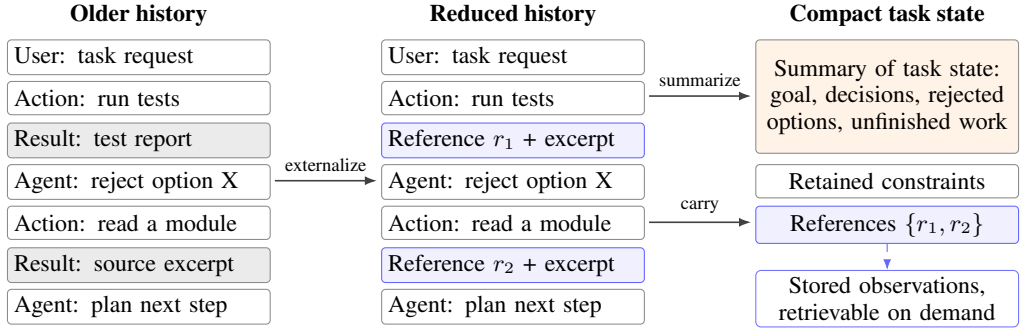

Figure 7: Input to history consolidation: verified observations are replaced by references with short excerpts, and the summarizer condenses the reduced history into a compact task state.

A.2 Preservation Requirements

Figure 6 illustrates how preservation requirements constrain revisions when similar observations call for different compression decisions.

Concretely, each requirement is a pair of a recorded input observation and the content that any policy must retain when processing it; a policy meets it if its output contains that content or if it abstains and retains the original observation. These checks constrain the tested cases only; they do not establish preservation of all information relevant to unseen tasks.

A.3 Policy Acceptance Criteria

Acceptance uses the accumulated replay set $\mathcal{D}_k$, containing original observations and action metadata from learning tasks completed through task k. Both the current and candidate policies are applied to these inputs, and their outputs are measured with the same token-count estimator. Observations left unchanged by a policy contribute their original token count, so the comparison accounts for the full replay set, including inputs outside the policy's coverage.

Both policies are checked against the same current preservation requirements. The candidate must satisfy every requirement, either by preserving the specified content or by retaining the input unchanged. A violation disqualifies the candidate regardless of token savings. Whether the current policy satisfies these requirements determines the acceptance condition:

- **Efficiency updates.** If the current policy satisfies all requirements, the candidate must strictly reduce the total estimated token count over the replay set without lengthening any individual representation. Unchanged totals are rejected, and savings on other inputs cannot offset an increase.
- **Preservation repairs.** If a new requirement invalidates the current policy, the candidate must satisfy all requirements, including those from earlier updates. Both length conditions are waived for this repair so that required content can be restored even at increased token cost. Efficiency criteria resume for later revisions once the current policy satisfies all preservation requirements.

An accepted revision takes effect at the next learning task; otherwise, the current policy is retained. The final policy is frozen for evaluation on disjoint tasks under matched conditions.

A.4 Experimental Details

Backbones. We use DeepSeek V4 Flash, MiniMax-M3, GLM-5.3 Flash, and Qwen3.8 Flash as backbones.

Policy evolution. Policy evolution is run separately for each backbone, with the evolution agent using the same underlying model as the acting agent. Every run starts from the same hand-written initial policy π_0 with 81 rules. The frozen policies contain 103 (DeepSeek V4 Flash), 91 (MiniMax-M3), 91 (GLM-5.3 Flash), and 89 (Qwen3.8 Flash) rules. No evolution takes place on LongCLI-Bench or MemoryAgentBench.

Baselines. DSH Basic is the native context manager of DeepSeek Harness with its standard compaction. LLMLingua-2 (Pan et al., 2024) shortens each observation with a learned token classifier, independently of the producing action and the structure of the output. Periodic Summary periodically replaces accumulated history with a summary; summarized observations are not kept retrievable. Complexity Trap (Lindenbauer et al., 2025) masks older observations while retaining recent exchanges. ACON (Kang et al., 2026) is adapted to DeepSeek Harness as a fixed-guideline compressor of both observations and history.

Metrics and token accounting. EvoAgentBench-SE pass rates are the mean and standard deviation over three runs; LongCLI-Bench results are averaged over two runs of each of the 20 tasks. Token counts include input tokens with cache reads, output tokens, and auxiliary model calls such as consolidation summaries. An adopted compression policy runs as a program without per-observation model inference and therefore adds no model tokens at deployment. The five MemoryAgentBench tasks cover all four memory competencies defined by the benchmark (Hu et al.,

2026): accurate retrieval (EventQA), test-time learning (TREC, Banking77), long-range understanding (DetectiveQA), and selective forgetting (Fact-SH32K). Micro weights task scores by the number of questions in each task (Appendix A.5); Macro averages the task scores.

A.5 MemoryAgentBench Task-Level Results

Table 3 gives the exact task accuracy (%), Micro and Macro aggregates, and average tokens per question behind Figure 4, computed over 871 test questions (500 EventQA, 100 TREC, 100 Banking77, 71 DetectiveQA, 100 Fact-SH32K).

Table 3: MemoryAgentBench task accuracy (%), Micro and Macro averages, and average tokens per question; bold marks the best value in each column within each backbone.

Method	EventQA	TREC	Banking77	DetectiveQA	Fact-SH32K	Micro	Macro	Avg. tokens
DeepSeek V4 Flash								
DSH Basic	**95.20**	88.00	91.00	76.06	45.00	86.57	79.05	28.5K
Complexity Trap	90.00	84.00	**96.00**	77.46	67.00	86.34	82.89	29.1K
LLMLingua-2	**95.20**	54.00	78.00	49.30	27.00	76.92	60.70	23.0K
ACON	92.33	**90.33**	**96.00**	**81.69**	65.00	88.52	85.07	26.8K
LightCode	94.80	90.00	93.00	76.06	**98.00**	**92.88**	**90.37**	**22.5K**
Qwen3.8 Flash								
DSH Basic	94.40	**92.00**	91.33	**81.69**	28.00	85.11	77.48	29.2K
Complexity Trap	88.00	90.00	94.00	78.87	65.00	85.53	83.17	18.3K
LLMLingua-2	**94.93**	79.00	80.00	64.79	33.00	81.82	70.34	26.3K
ACON	92.67	88.00	92.00	77.46	50.00	85.92	80.03	25.7K
LightCode	93.40	**92.00**	**96.00**	76.06	**71.00**	**89.55**	**85.69**	**17.6K**
GLM-5.3 Flash								
DSH Basic	90.00	**93.33**	93.00	76.53	51.33	85.19	80.84	30.3K
Complexity Trap	89.20	88.00	94.00	76.53	66.00	85.92	82.75	29.2K
LLMLingua-2	**94.80**	71.00	80.33	55.40	32.33	80.02	66.77	22.9K
ACON	91.73	92.00	92.33	73.24	**67.00**	87.49	83.26	26.4K
LightCode	92.80	93.00	**96.00**	**83.10**	65.00	**89.21**	**85.98**	**16.8K**
MiniMax-M3								
DSH Basic	**96.20**	88.67	93.33	66.20	61.00	88.52	81.08	21.7K
Complexity Trap	78.00	88.33	91.33	74.65	56.33	77.96	77.73	19.6K
LLMLingua-2	89.20	42.00	73.33	43.66	27.00	71.11	55.04	17.9K
ACON	95.20	88.33	93.00	73.24	56.33	87.91	81.22	18.6K
LightCode	94.80	**93.00**	**94.00**	**80.28**	**71.00**	**90.59**	**86.62**	**16.9K**

A.6 Policy Evolution Statistics

Table 4 summarizes policy evolution for all four backbones. Every policy grows mainly by refining existing rules rather than by adding new ones: between 8 and 22 rules are added, whereas 38 to 57 adopted revisions refine rules already present.

Table 4: Policy evolution on the 87 learning tasks of EvoAgentBench-SE; all four backbones start from the same initial policy π_0.

	DeepSeek V4 Flash	MiniMax-M3	GLM-5.3 Flash	Qwen3.8 Flash
Rules, $\pi_0 \to \pi_k$	$81 \to 103$	$81 \to 91$	$81 \to 91$	$81 \to 89$
added during evolution	22	10	10	8
Adopted revisions	67	49	67	46
introducing new rules	19	10	10	8
refining existing rules	48	39	57	38