STATETAPE: ACTION-CONDITIONED EVIDENCE LIFECYCLE MODELING FOR LONG-HORIZON CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Despite the recent success of coding agents built on large language models, it remains challenging to run them over long horizons, since every observation is appended to the context and the context grows with each one. *History-based maintenance* is a common remedy, which masks or summarizes old observations, or prunes what a model reads as useless, and bounds the context at little cost. However, it decides from the text of the history alone and sees nothing of how the code is connected. Since a coding agent edits code many times over a single task, and each write can change what code elsewhere means, such maintenance may keep records a write has falsified, drop ones that still hold, and miss code the agent needs next. To overcome these challenges, this paper proposes StateTape, a novel and scalable framework that rewrites a coding agent's context as the repository changes rather than as the context grows. The key idea of StateTape is to model the repository as a symbol-level code graph, whose dependencies and language rules expose which symbols a write can affect. Upon this graph, a tape marks the symbols each write changed, which turns staleness from an inference about text into an observation of the agent's writes. We propose a per-write procedure in which the tape nominates the records a write could have falsified while a small manager model settles what the write log cannot, and further provide a theoretical analysis and TraceBench, a benchmark that labels what an agent is holding against what is actually needed. Empirically, we demonstrate that StateTape can effectively clear falsified records and retrieve what is needed, and thus achieve a higher resolve rate in all experiments spanned by six coding agents and three edit-heavy benchmarks with little computational overhead.

1 INTRODUCTION

Coding agents built on large language models have shown promising results on real software engineering work, running for hundreds of steps on a single task where they read files, run tests and edit code (Yang et al., 2024; Wang et al., 2025a; 2024; Zhang et al., 2024). Despite their great promise, these agents meet significant challenges on long-horizon coding tasks that require many edits. On benchmarks built from such tasks (Deng et al., 2025; Le et al., 2025; Desai et al., 2026; Orlanski et al., 2026), the resolve rate declines as the task grows longer, even when every individual step lies within the model's ability (Kwa et al., 2026; Sinha et al., 2026). Part of this difficulty is mechanical: every observation is appended to the context and a long context costs tokens and degrades the model's use of what it already holds (Liu et al., 2024a). To tackle this challenge, context management, which decides what an agent carries from one request to the next, has attracted fast-increasing attention and has become standard practice in agent harnesses (Lindenbauer et al., 2025; Wang et al., 2025b; Liu et al., 2026).

A key challenge in context management lies in retaining the observations that later steps depend on while keeping the context small enough to stay affordable and usable. On the one hand, carrying the entire history preserves every observation the agent has made, whereas the agent pays for that history on every later request and the cost grows without bound as the task proceeds. On the other hand, trimming the history bounds the context, while it can discard evidence the agent still needs and force the agent to recover that evidence through additional steps. More recently, deciding what to keep directly from the recorded history has been widely adopted and achieved strong results in

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

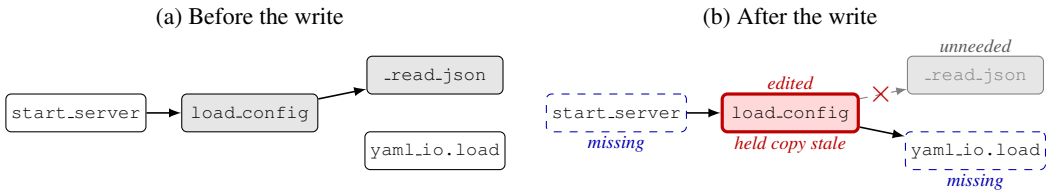


Figure 1: A single write leaves the context wrong in three ways. Boxes are functions and arrows are calls; gray fill marks what the agent has read. (a) The agent reads `load_config` and its helper `_read_json`. (b) It rewrites `load_config` to parse YAML and raise `ConfigError`. Its held copy of `load_config` is now *stale*; `_read_json` has lost its only caller and is *unneeded*; the caller `start_server` and the new callee `yaml_io.load` are needed next but *missing* (dashed).

long-horizon coding agents (Lindenbauer et al., 2025; Wang et al., 2025b; Liu et al., 2026; Xiao et al., 2025; Yi et al., 2026; Li et al., 2026b). Specifically, by masking or collapsing observations older than a window (Lindenbauer et al., 2025), summarizing the history once it passes a size limit (Wang et al., 2025b; Liu et al., 2026), or prompting a model to read each record and discard the ones it judges useless or expired (Xiao et al., 2025; Yi et al., 2026; Li et al., 2026b), such methods, which we call *history-based maintenance*, bound the context at a cost small next to the agent's own.

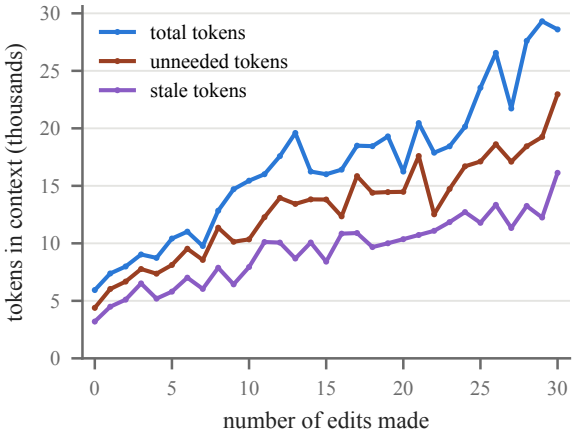


Figure 2: Mean held tokens at the TraceBench decision points against the number of edits made before them. **73.4%** of the tokens Claude Code holds are unneeded, and **58.5%** of the tokens it needs are missing.

However, history-based maintenance is limited by the signal it decides on: the text of the history. What makes a coding agent different from an agent that only remembers is that its own writes change what its evidence means. Figure 1 gives an example. Consider an agent that is asked to make `load_config` validate its input. It reads `load_config`, which parses the file through a helper `_read_json`, and it reads that helper too. It then rewrites `load_config` so that the function parses YAML with `yaml_io.load` and raises `ConfigError` when a key is missing. This single write leaves the context wrong in three ways. First, the held copy of `load_config` is *stale*, because it shows a version that never raises and that the file no longer contains. Second, the held copy of `_read_json` is *unneeded*. Its text is still correct, but the write deleted its only call, so no later step will use it. Third, the evidence for the next edit is *missing*. The function `start_server` calls `load_config` and must now catch `ConfigError`, and the agent has read neither that caller nor `yaml_io.load`. All three errors follow from the write and the call edges around it. A mask, a summary or a pruning model sees only records, so it cannot tell that `_read_json` lost its caller, and no record names `start_server`. Figure 2 measures these errors at the 403 moments where real Claude Code runs are about to edit. Unneeded tokens grow with every write rather than settling. Every later request carries them, costing tokens and lowering the resolve rate (§4).

Closing this gap means deciding, at every write, what to drop and what to add, which is hard for two reasons: 1). *Deciding staleness from writes rather than from text.* A write and a held record have to be comparable at the same granularity. Rewriting `load_config` should mark its held copy as stale, but it should not mark every definition in the same file, and no stale record should be missed. 2). *Adding the evidence the next edit needs but the agent has not read.* What is missing is rarely the edit site itself but a definition it calls, a caller that must change with it, or a contract it must meet, all of which lie along dependency edges that a keyword search cannot enumerate. In the example, the task never mentions `start_server`, so a search over it misses this caller.

Our Contribution. To jointly address all challenges above, we propose **StateTape**, a context maintenance framework towards the trade-off between the cost and quality of the context. In StateTape, we design a symbol-level *code graph* of definitions and call edges that is updated at every write,

so that writes, held records and the dependencies among them are comparable in the same granularity. Based on this graph, we use *tape* to formulate context maintenance as a *write-conditioned* procedure, where what leaves the context and what enters it are both decided by the symbols the current write changed. Building on the tape, we further propose a lightweight *manager* that removes what is stale and useless in the context, and retrieves what the task needs by walking the code graph outward from the changed symbols. In the example, the tape marks `load_config` as changed and refreshes its held copy. The manager then drops `_read_json`, which has lost its caller, and retrieves `start_server` and `yaml_io.load`, which lie one call edge from the write. Finally, we perform extensive evaluations over five comparison methods on three edit-heavy benchmarks with six coding agents, where StateTape attains the best resolve rate in every case at few computational overhead.

2 Related Work

Context management for coding agents. Coding agents act on a repository through tools (Yang et al., 2024; Wang et al., 2025a; Zhang et al., 2024; Wang et al., 2024; Xia et al., 2025), and on benchmarks that need many edits (Deng et al., 2025; Le et al., 2025; Desai et al., 2026; Orlanski et al., 2026; Jin et al., 2026) resolve rate falls with horizon even when each step is within the model's competence (Kwa et al., 2026; Sinha et al., 2026). Their context is managed by age or by a model: collapsing or masking old observations (Lindenbauer et al., 2025), summarizing past a size threshold (Wang et al., 2025b; Liu et al., 2026), or pruning what a model judges useless or expired (Xiao et al., 2025; Liu et al., 2025a; Wang et al., 2026), with AdaCoM and ContextCurator (Yi et al., 2026; Li et al., 2026b) training a second model to edit a frozen agent's context, the same two-model split as our manager. CORVUS (Zheng et al., 2026) re-injects registered files' on-disk contents every cycle, but unconditionally, whole-file, and for file reads alone; coherence-debt measurements (Mohammadi et al., 2026) name the missing invalidation bit, and CoAgent (Lyu et al., 2026) supplies it only when *another* agent writes the object. None of them reads the event that falsifies a record: the agent's own write. StateTape takes that write as its signal and resolves it to the symbols it changed, so falsified records are named rather than inferred, and the same symbols seed retrieval for the next edit.

Repository structure for retrieval and localization. Repository-level code models bring in cross-file context through structure: iterative retrieval (Zhang et al., 2023), cross-file dependencies (Ding et al., 2024), and graphs over definitions, references and dependencies (Ouyang et al., 2025; Liu et al., 2025b; Tao et al., 2025; Liu et al., 2024b), which an agent may traverse to localize code (Chen et al., 2025) and which benchmarks score against one gold context per task (Li et al., 2026a; Zhang et al., 2026). On the evaluation side, Slipstream (Chen et al., 2026) derives what a compaction must preserve from the agent's next few steps, the nearest existing per-step target. These works decide what to evict and what to retrieve from text or from a graph fixed at task start, and none of them measures what the agent holds against what it still needs. StateTape instead keeps that graph updated at every write, so that the symbols a write changed decide both what leaves the context and what enters it, and TraceBench measures both sides of that gap at every decision point.

3 Method

Given a coding agent's context and the actions it takes on a repository, StateTape maintains that context at every write: it removes evidence the task no longer needs and adds the code the next edit is missing. We first measure, on TraceBench, how far the context an agent holds is from the context it needs (§3.1), and state that gap as two objectives under one token budget (§3.2). StateTape consists of three components that run inside the agent harness. First, a symbol-level code graph gives writes and held records a shared coordinate system in which the two can be compared (§3.3.1). Next, the tape logs what each write changed in that graph and nominates every held record the change could have falsified (§3.3.2). Finally, a small manager model evicts records that are stale or no longer used, and retrieves the unread code the next edit needs by following the graph outward from the symbols that changed (§3.3.3). The key idea is that the agent's own writes are the only events that can make its evidence stale, so staleness is observed when it happens rather than inferred from what a record says, and the same observation locates the missing evidence. Figure 4 shows the procedure.

3.1 Motivation: The Held Context Is Far from the Needed Context

At every step a coding agent acts on the context it holds, C_t, while what it should act on is the gold context G_t, the smallest body of evidence sufficient to finish the task from that step. As is shown in Figure 3, the two sets split the evidence into three regions: held but not needed, held and needed, and needed but not held. Context management for coding agents keeps what the agent has

recently seen and trims it at the margin, which works only if the first and last regions are small (§2). No benchmark measures them, so §3.1.1 builds TraceBench, which labels both regions at decision points inside long, write-dense trajectories. §3.1.2 shows that most of what the agent holds is not needed and most of what the next edit needs is not held, the gap StateTape addresses.

3.1.1 TRACEBENCH

Existing benchmarks cannot measure this gap. Issue-resolution benchmarks score only the terminal patch, long-context benchmarks place evidence that nothing the model does can falsify, and localization benchmarks label the code a task needs once per task and never label what the agent holds but no longer needs. We therefore introduce TraceBench, which labels the held context at decision points inside long, write-dense agent trajectories. Each decision point is annotated with both regions of Figure 3 under one protocol, and every quantity is measured in tokens, the unit the agent pays in.

We run an unmodified agent, Claude Code with Opus 4.8, on 48 SWE-EVO instances (Le et al., 2025) from seven repositories, so every labeled context was actually held. SWE-EVO tasks are version-to-version feature evolutions with many-hunk patches, so runs are long and write-dense. We cut each trajectory before every edit burst, yielding 403 decision points, each labeled with $C_t \setminus G_t$ and $G_t \setminus C_t$ (Appendix E).

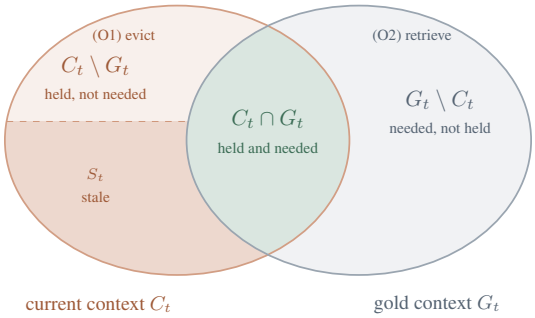


Figure 3: Held context C_t and gold context G_t. The stale set S_t lies strictly inside $C_t \setminus G_t$, since a true record can still be unused. O1 drains the left region and O2 pulls from the right under one token budget.

3.1.2 OBSERVATIONS

Most of what the agent holds is not needed, and the share grows with edits. Across the 403 decision points, 73.4% of held tokens lie in the left region of Figure 3 (Table 5). Figure 2 shows how this builds up as a run accumulates edits: the context grows from 5.9k tokens before the first edit to 28.6k after the thirtieth, and the part of it the imminent edit does not need grows with it from 4.4k to 23.0k tokens, 74% to 80% of what is held. The unneeded mass is therefore not a fixed overhead that a larger window absorbs.

Stale records are a majority of the held context. Stale records, those a later write falsified, carry 72.3% of the left region's tokens. This measures the strict inclusion of Figure 3. The stale part can be identified from the agent's writes alone, since only a write can falsify a record.

Most of what the edit needs is not held. In TraceBench, 58.5% of gold tokens are not in the context when the edit that needs them is made, and the context covers 41.5% of the gold context on average. What is missing is rarely the edit site itself but the definitions, callers, and contracts a call edge away from it. Retrieval therefore has a starting point that a query over the task description does not give it, namely the symbols the agent is about to change.

Together, the three observations point at a single signal. The unneeded side grows with the agent's writes and is mostly stale. The missing side lies along the call and dependency edges of the code that those writes touch. §3.2 formulates the gap as two optimization objectives and turns this into the requirements a context management policy must meet.

3.2 PROBLEM FORMULATION

Setting and goal. We study context maintenance for a coding agent over a long, write-dense run on a repository. At each write, a maintenance policy sees what the harness sees: the agent's context, the write, and the repository before and after it, but neither the reference patch nor any label of what the task needs. The goal is to rewrite the context so that it carries few tokens the next edit does not need and covers the evidence that edit does need, within the agent's token budget and at a cost small relative to the agent's own. Performance is measured at the end of a run by whether the agent resolves the task and how many tokens it spends, and at each decision point by the two context objectives and action-level efficiency scores defined below.

Setup. A coding agent acts on a repository whose state at step t is w_t. Actions split into observations (reads, searches, test runs), which leave $w_{t+1} = w_t$, and writes $\mathcal{A}_{\text{wr}}$ (edits, creations, deletions), which make w_{t+1} the state a_t produces from w_t. The agent's context is a finite set of

records C_t. A record $r = (c_r, \rho_r, \tau_r)$ holds text c_r, a referent ρ_r naming the part of the repository that text is about, and the step τ_r at which it entered the context, and costs $n(r) = |c_r|$ tokens, extended to sets by $n(S) = \sum_{r \in S} n(r)$. A referent may be a byte range, a symbol, the solution set of a query, or the repository state a test ran under.

At step t, let the **gold context** G_t be a smallest set of records sufficient to complete the task from t. Its overlap with the held context defines coverage, $\mathrm{Rec}(C_t) = n(C_t \cap G_t)/n(G_t) \in [0, 1]$, while $\sigma(C_t) = n(C_t \setminus G_t)$ is the **unneeded mass**. A maintenance policy acts at each write by removing $D_t \subseteq C_t$ and adding A_t, where $A_t \cap C_t = \emptyset$, producing $C_t^+ = (C_t \setminus D_t) \cup A_t$; the agent's next observation extends this context into C_{t+1}. The main agent's token budget B bounds both contexts.

Each record is a timestamped claim: its text says something about the repository that was true when the agent observed it and that a later edit can make false. That claim is a predicate ϕ_r on repository states, where $\phi_r(w)$ asks whether the record still holds of w. Record r is **stale** at t once its claim has become false, $\neg\phi_r(w_t)$; the stale set is $S_t = \{r \in C_t : \neg\phi_r(w_t)\}$, containing $n(S_t)$ stale tokens.

Staleness is action-conditioned and monotone. Observations leave w_t fixed, so writes are the only events that can falsify a record: if no write $a_s \in \mathcal{A}_{\mathrm{wr}}$ with $\tau_r \le s < t$ touches ρ_r, then $\phi_r(w_t) = \phi_r(w_{\tau_r}) = \mathrm{true}$. Two things follow. First, every stale record traces back to the write that falsified it, and that write is an event the harness already observes, so staleness need not be guessed from the text of the record. Second, without drops or rebuilds the stale set only grows:

$$S_t \subseteq S_{t+1}, \qquad \text{so} \qquad n(S_t) \le n(S_{t+1}). \tag{1}$$

The gap between what the agent believes and what the repository holds therefore never shrinks on its own, and it grows with the number of writes. A run with few writes builds up almost none of it. This is the precise sense in which evidence lifecycle is a long-horizon problem rather than ordinary context management, and it is why the effects in §4 follow the number of edits rather than of steps.

Optimization objective. The general context-maintenance problem is to reduce held but unneeded evidence (Xiao et al., 2025; Liu et al., 2025a) and increase coverage of needed repository evidence (Zhang et al., 2023) under the agent's token budget. At each write, we express these two aims as

$$\min\ \sigma(C_t) \qquad \max\ \mathrm{Rec}(C_t) \qquad \text{s.t.}\quad n(C_t) \le B, \tag{2}$$

The first targets the left region of Figure 3 and the second targets the right-region deficit through gold-context coverage. These are the two gaps measured in §3.1.2, stated as objectives.

From objectives to scoring surrogates. Eq. 2 states the two aims, but a policy cannot optimize it as written. Eq. 2 describes a good context and not the action that produces one. At a write the policy chooses D_t and A_t, and every pair with $n(C_t^+) \le B$ is feasible. An eviction that clears unneeded tokens usually destroys needed ones too, and an addition that raises coverage adds unneeded ones.

We therefore score the action, not the context it leaves, and charge every action for its cost. **Collateral efficiency** measures the unneeded tokens an eviction D_t clears per needed token it destroys,

$$\mathrm{CE}(D_t) \;=\; n(D_t \setminus G_t) \,/\, n(D_t \cap G_t), \tag{3}$$

and **retrieval efficiency** measures the share of the right region an addition A_t recovers per token it adds,

$$\mathrm{RE}(A_t) \;=\; \kappa \cdot n\big(A_t \cap (G_t \setminus C_t)\big) \,/\, \big(n(G_t \setminus C_t) \cdot n(A_t)\big), \tag{4}$$

where $\kappa > 0$ is a scale constant. It does not change the maximizer of RE and only sets the unit in which RE is reported. The two aims then become objectives over the action taken at each write,

$$\textbf{(O1)}\quad \max_{D_t}\ \mathrm{CE}(D_t) \qquad \textbf{(O2)}\quad \max_{A_t}\ \mathrm{RE}(A_t) \qquad \text{s.t.}\quad n(C_t^+) \le B. \tag{5}$$

Technical requirements. Approximating O1 and O2 from observable signals at every write requires addressing three requirements, one for each observation of §3.1.2. **Requirement 1 (Write-grounded staleness):** decide which held records are stale from the writes that falsified them, without a model call and without letting a stale record escape, since the stale set grows with every write (Eq. 1). **Requirement 2 (Forward-looking eviction):** remove records that are still true but no longer used, the smaller part of $C_t \setminus G_t$, which depends on where the task goes next. **Requirement 3 (Dependency-targeted retrieval):** add the missing part of G_t, which lies along the call and dependency edges of the code being edited. Existing context management does not address these requirements. StateTape assigns these responsibilities to its three components: the code graph gives Requirements 1 and 3 a shared coordinate system (§3.3.1), the tape meets Requirement 1 without a model (§3.3.2), and the manager meets Requirements 2 and 3 on top of the tape (§3.3.3).

3.3 STATETAPE

Figure 4 provides the detailed end-to-end flow of StateTape. The tape receives the write before it reaches the repository and applies it to the code graph; the symbols that drifted return as nominations of the records they falsify, and the manager acts on those nominations and searches the graph outward from the drifted symbols for the code the next edit needs. The agent's next request then carries a maintained context rather than the one it accumulated.

3.3.1 A SYMBOL-LEVEL CODE GRAPH

Requirement 1 decides staleness from writes, and Requirement 3 finds missing evidence along dependencies. Both need writes, records and dependencies in shared units, so a write can be matched against the records it affects and the code depending on it; the code graph provides them.

The code graph describes the repository as named definitions and the calls between them. Its unit is the symbol (a function, method or class), the smallest unit whose change alters meaning, not position.

Formally, the code graph of a repository state at step t is $\Phi(t) = (V_t, E_t, b_t)$ with $E_t \subseteq V_t \times V_t$, where V_t is the set of symbols, E_t holds call edges, and $b_t(v)$ is the body text of symbol v.

A symbol keeps its identity when an edit moves it, and two states can be compared symbol by symbol through V_t and b_t. The edges E_t carry the relationships between symbols.

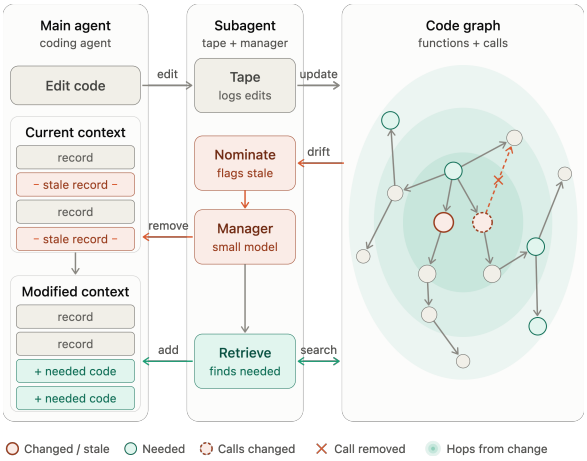


Figure 4: Overview of StateTape. *Left:* the agent edits code, and its context, which holds stale records, becomes a modified context with those records removed and needed code added. *Middle:* the tape applies each edit to the code graph; the drifted symbols nominate the records they falsify, and the manager removes records and retrieves code. *Right:* the code graph of functions and calls. A write changes a function's body (solid red) or where a call resolves (dashed; removed call crossed out); shaded rings mark hops from the change, and green nodes are code the next edit needs. Red paths serve eviction (O1), green paths retrieval (O2).

3.3.2 THE TAPE: ACTION-CONDITIONED STALENESS CHECKER

The tape decides which held records are stale (Requirement 1). At every write, it compares the code graph before and after the write, logs the symbols that changed, and nominates every held record that covers one of them. Because it only compares two graphs, it makes no model call.

A write can change a symbol in two ways. It can edit the symbol's body, or it can leave the body untouched and change which definition one of the symbol's calls reaches. Suppose the agent has read `read_config`, which calls `load`. If the agent then changes the import from `json_io.load` to `yaml_io.load`, the body of `read_config` stays the same, but the record of `read_config` no longer describes what the code does. The tape detects the first through symbol bodies and the second through call edges; a symbol changed either way has **drifted** (Appendix C).

Formally, with $\Phi(t) = (V_t, E_t, b_t)$ as in §3.3.1, the write a_t produces two sets of drifted symbols,

$$\begin{aligned} \mathcal{X}_t &= \big(V_{t-1} \setminus V_t\big) \cup \big\{\, v \in V_{t-1} \cap V_t \;:\; b_t(v) \neq b_{t-1}(v) \,\big\}, \\ \partial\mathcal{X}_t &= \left(\bigcup_{(x,y) \in (E_{t-1} \setminus E_t) \cup (E_t \setminus E_{t-1})} \{x, y\} \right) \cap \big(V_{t-1} \cap V_t\big) \setminus \mathcal{X}_t. \end{aligned} \tag{6}$$

$\mathcal{X}_t$ holds the symbols the write removed or whose body it edited. $\partial\mathcal{X}_t$ holds the remaining symbols that gained or lost a call edge; in the example above, it contains `read_config`. Both sets are empty after an observation, since an observation leaves the graph unchanged. The tape logs them with the step t, so each change stays attributed to the write that made it.

To check a record, the tape collects its drift since τ_r, $\mathcal{X}_{(\tau_r,t]} = \bigcup_{\tau_r < u \le t} \mathcal{X}_u$, and $\partial\mathcal{X}_{(\tau_r,t]}$ likewise; a record is **nominated** when it covers a symbol in either,

$$\mathcal{N}_t = \big\{\, r \in C_t \;:\; \rho_r \cap \big(\mathcal{X}_{(\tau_r,\,t]} \cup \partial\mathcal{X}_{(\tau_r,\,t]}\big) \neq \emptyset \,\big\} \supseteq S_t. \tag{7}$$

The inclusion $\mathcal{N}_t \supseteq S_t$ states that no stale record escapes nomination; Theorem 1 gives the condition under which it holds. A nominated record need not be stale; the manager decides its fate (§3.3.3).

The tape nominates by rule, not by a model, for two reasons. First, the rule is exact. The code graph expresses writes and held records in the same symbols, so each write is matched directly against the records that cover what it changed. Second, it costs no inference (Theorem 2).

3.3.3 THE MANAGER

The manager handles what the tape cannot: records that are still true but no longer used (Requirement 2), and code the next edit needs but the agent has not read (Requirement 3). It is a small LLM, invoked in the agent's request path only when the tape carries a nomination, and it sees the repository only through read-only skills that the graph and tape answer without a model call (Table 7).

At each invocation the manager makes three decisions. First, it assigns each nominated record one of four actions. REFRESH replaces the record with the current body $\mathrm{fresh}(r)$. DROP leaves a short tombstone $\mathrm{tomb}(r)$ naming the changed symbols, so the agent knows what it lost. RERUN replaces a test result with a stub $\mathrm{stub}(r)$ asking the agent to run the test again. KEEP leaves a record that was nominated but is still correct. Second, it evicts records that no write falsified but that the task has finished with. We write this set D_t^m; the tape cannot find it, because it depends on where the task goes next. Third, it retrieves the code the next edit needs, R_t^m. Since that code usually lies one call edge from the code being changed (§3.1.2), the manager follows call edges outward from the drifted symbols for at most two hops and checks each candidate against source.

In the running example, the body of `read_config` is unchanged, so refreshing it would restore the same text. The manager drops it instead, and the tombstone tells the agent that the call now resolves elsewhere. It also evicts an old test output on an unrelated module, and retrieves the new callee `yaml_io.load` with a caller of `read_config` that indexes the returned dictionary.

Sorting nominations by action into N_t^{DROP}, N_t^{REFRESH}, N_t^{RERUN} and N_t^{KEEP}, the policy is

$$\begin{aligned} D_t &= N_t^{\text{DROP}} \cup N_t^{\text{REFRESH}} \cup N_t^{\text{RERUN}} \cup D_t^m, \\ A_t &= \mathrm{tomb}\big(N_t^{\text{DROP}}\big) \cup \mathrm{fresh}\big(N_t^{\text{REFRESH}}\big) \cup \mathrm{stub}\big(N_t^{\text{RERUN}}\big) \cup R_t^m. \end{aligned} \tag{8}$$

The first line serves O1, the second O2, and the manager trades them off within B.

3.4 THEORETICAL ANALYSIS

Soundness. We analyze what the tape guarantees under any manager (proofs in Appendix A).

Theorem 1. *Under the symbol-locality condition of Assumption 1, at every write step t: (a) every stale read, edit or test result in C_t is in $\mathcal{N}_t$; (b) for any manager output whose retrievals cover no drifted symbol, the stale mass of reads, edits and test results in the maintained context C_t^+ is at most $n(N_t^{\text{KEEP}} \cap S_t)$.*

Theorem 1 says that the tape finds every held record the agent's writes broke without calling a model, and that no manager decision can reintroduce a stale record.

Cost. Here we analyze the cost of StateTape over a run of T requests to the main agent, W of which follow a write, with $\mathcal{T}_N$ the requests that carry a nomination.

Theorem 2. *(a) The tape makes no model calls, and per write it costs $\mathcal{O}\big(d \sum_{r \in C_t} |\rho_r|\big)$, with d the largest out-degree of a held symbol, beyond parsing the written files, independent of repository size. (b) If a manager call costs $\mathcal{O}(|\mathcal{N}_t|)$ tokens, the manager's tokens K^{mgr} and the main agent's tokens K^{agt} satisfy $K^{\text{mgr}}/K^{\text{agt}} = \mathcal{O}\big(|\mathcal{T}_N|\,\bar{N}/(T\,n_{\min})\big)$, where $\bar{N}$ is the mean nomination count over $\mathcal{T}_N$ and $n_{\min}$ the smallest context sent. (c) If each write falsifies at least δ held tokens, the stale tokens sent to the agent over the run are at least $\Omega(\delta W^2)$ without maintenance and at most $\sum_t n(N_t^{\text{KEEP}} \cap S_t)$ with StateTape.*

Theorem 2 shows that StateTape's overhead is bounded.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Models and Datasets. We evaluate six main agents: Claude Haiku 4.5, Sonnet 5 and Opus 4.8 in Claude Code, and GPT-5.6 luna, terra and sol in the Codex CLI. The manager is Haiku 4.5 for the Claude agents and GPT-5.4 nano for the GPT agents. Resolve rate is measured on three benchmarks

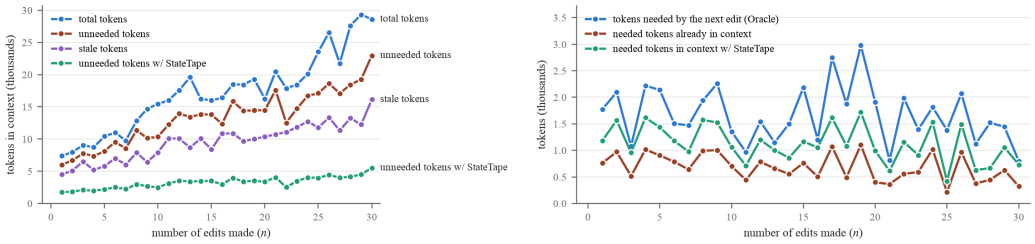


Figure 5: Context eviction and retrieval on TraceBench as a function of the number of edits made at each decision point. *Left:* total held tokens $n(C_t)$, unneeded tokens $n(C_t \setminus G_t)$, the stale subset of those tokens, and the unneeded tokens remaining after StateTape. *Right:* tokens needed by the next edit $n(G_t)$, those already held $n(C_t \cap G_t)$, and those held after StateTape $n(C_t^+ \cap G_t)$.

Main agent	Configuration	SWE-EVO	SlopCode	SWE-Pro	Main agent	Configuration	SWE-EVO	SlopCode	SWE-Pro
Haiku 4.5	Control	25.0	4.6	26.3	GPT-5.6 luna	Control	56.3	4.1	64.3
	Obs. masking	27.1	5.1	27.1		Obs. masking	58.3	4.6	65.8
	LLM summary	27.1	6.1	26.7		LLM summary	56.3	5.1	64.7
	CORVUS	25.0	5.6	26.3		CORVUS	58.3	6.1	64.3
	Text-only mgr	25.0	7.1	27.1		Text-only mgr	58.3	5.1	66.2
	StateTape	**29.2**	**8.1**	**27.8**		*StateTape*	**60.4**	**6.6**	**67.7**
Sonnet 5	Control	41.7	9.2	63.5	GPT-5.6 terra	Control	52.1	6.6	65.8
	Obs. masking	41.7	10.2	65.0		Obs. masking	54.2	7.1	67.7
	LLM summary	43.8	9.7	63.9		LLM summary	54.2	7.7	66.2
	CORVUS	45.8	11.2	63.5		CORVUS	52.1	7.1	65.8
	Text-only mgr	43.8	10.7	65.4		Text-only mgr	56.3	7.1	67.7
	StateTape	**47.9**	**12.8**	**68.4**		*StateTape*	**58.3**	**8.2**	**68.4**
Opus 4.8	Control	58.3	9.7	69.2	GPT-5.6 sol	Control	58.3	8.2	64.6
	Obs. masking	58.3	10.7	71.1		Obs. masking	60.4	8.7	69.2
	LLM summary	58.3	11.2	69.5		LLM summary	58.3	9.2	67.7
	CORVUS	60.4	10.7	69.2		CORVUS	58.3	9.7	67.3
	Text-only mgr	60.4	12.2	68.4		Text-only mgr	60.4	9.7	68.1
	StateTape	**62.5**	**12.8**	**71.4**		*StateTape*	**62.5**	**10.2**	**69.5**

Table 1: End-to-end resolve rate (%) on SWE-EVO, SlopCodeBench (SlopCode) and SWE-bench Pro (SWE-Pro), reported for six main agents from the Claude and GPT-5.6 families. Each main agent is run under Control, four context-management baselines and StateTape (shaded). Within each main-agent block, bold denotes the best result on each benchmark.

whose tasks require many edits: SWE-EVO (Le et al., 2025), SlopCodeBench (Orlanski et al., 2026) and SWE-bench Pro (Deng et al., 2025). O1 and O2 are scored with CE and RE (§3.2) on TraceBench (§3.1). With $\kappa = 10^5$, RE is in percentage points of $G_t \setminus C_t$ recovered per thousand tokens added. Besides the unmodified agent (Control), we compare four baselines in the same shim: **observation masking** and **LLM summary** (Lindenbauer et al., 2025), **CORVUS** (Zheng et al., 2026), and a **text-only manager** in the style of AgentDiet (Xiao et al., 2025) (Appendix D).

4.2 Main Results

StateTape improves resolve rate for every main agent. StateTape has the highest resolve rate in all 18 agent–benchmark cells of Table 1. Averaged over agents, it adds 4.9 points over Control on SWE-EVO, 2.7 on SlopCodeBench and 3.3 on SWE-bench Pro. Sonnet 5 gains most (6.2, 3.6, 4.9).

StateTape has little computational overhead. Across the 18 cells of Table 4, the manager uses 0.8% to 4.7% as many tokens as the main agent, and 2.6% when summed over all of them.

StateTape evicts and retrieves more selectively than other methods. On TraceBench (Table 2), StateTape reaches the highest CE (6.18) and RE (10.47). It clears 74.2% of unneeded tokens and recovers 68.1% of $G_t \setminus C_t$ with 6,502 added tokens per decision point, at every edit count (Figure 5).

4.3 Ablation Studies

Table 3 replaces one component at a time (main agent Sonnet 5): files replace symbols in the code graph, a text judgment replaces the tape, and default actions the manager.

Removing the code graph causes the largest drop (-3.4). CE halves to 3.14, because a file-level tape cannot tell an edit to one method from an edit that only shifts the definitions below it. Coverage falls to 44.3%, because retrieval can no longer follow call edges to the 2,721 of 3,545 gold regions that are not the edit site.

Variant	Resolve rate (%)				TraceBench		
	SWE-EVO	SlopCode	SWE-Pro	Δ	CE ↑	Missing retrieved (%)	RE ↑
StateTape (full)	**47.9**	**12.8**	**68.4**	—	**6.18**	**68.1**	**10.47**
w/o code graph (file-level tape)	43.8	10.2	65.0	−3.37	3.14	44.3	4.86
w/o tape (text-judged staleness)	43.8	10.7	65.4	−3.07	2.86	21.7	3.94
w/o manager (tape defaults only)	45.8	11.2	66.2	−1.97	4.91	31.2	4.57

Table 3: Ablation of StateTape's three core components. Δ is the mean change in resolve rate over the three benchmarks. Bold denotes the best result.

Main agent	Benchmark	Agent	Manager	Manager/ Agent (%)	Main agent	Benchmark	Agent	Manager	Manager/ Agent (%)
	SWE-EVO	206.9	8.4	4.1		SWE-EVO	70.3	0.5	0.8
Haiku 4.5	SlopCodeBench	526.7	19.8	3.8	GPT-5.6 luna	SlopCodeBench	95.2	1.9	2.0
	SWE-bench Pro	1039.3	24.0	2.3		SWE-bench Pro	206.3	1.6	0.8
	SWE-EVO	185.8	6.1	3.3		SWE-EVO	44.2	0.4	1.0
Sonnet 5	SlopCodeBench	575.9	4.8	0.8	GPT-5.6 terra	SlopCodeBench	76.5	1.8	2.3
	SWE-bench Pro	730.9	22.9	3.1		SWE-bench Pro	150.8	2.7	1.8
	SWE-EVO	77.6	2.9	3.7		SWE-EVO	43.4	0.5	1.1
Opus 4.8	SlopCodeBench	94.3	4.5	4.7	GPT-5.6 sol	SlopCodeBench	73.9	0.8	1.1
	SWE-bench Pro	444.2	18.9	4.3		SWE-bench Pro	151.0	2.8	1.8

Table 4: Token cost of StateTape, in millions of tokens summed over each benchmark's instances (SlopCodeBench: all checkpoints of the 36 problems). *Agent* counts the main agent's input tokens, cache reads included, plus output tokens. *Manager* counts the manager's input and output tokens (Haiku 4.5 for the Claude agents, GPT-5.4 nano for the GPT agents), including calls made in retried attempts. *Manager/Agent* is the manager's tokens as a percentage of the main agent's.

Removing the tape costs −3.1. Its CE of 2.86 is close to the 2.75 of EVICT-ALL, since a model reading a record's wording cannot tell whether a later write falsified it. Retrieval also drops to 21.7%, since it has no drifted symbol to start from. Removing the manager costs the least (−2.0), since the tape's default actions still run. CE stays at 4.91, but finished records survive and coverage falls to the 31.2% of REFRESH.

O1: eviction	Cleared (%)	Lost / DP	CE ↑
KEEP-ALL	0.0	0	–
EVICT-ALL	100.0	4,823	2.75
recency, $k = 5$	96.8	4,400	2.92
recency, $k = 10$	91.5	4,188	2.90
recency, $k = 20$	80.1	3,891	2.73
recency, $k = 40$	57.9	3,371	2.28
age window, 25	73.3	3,733	2.61
age window, 50	47.1	3,125	2.00
token budget, 16k	23.6	2,678	1.17
token budget, 32k	0.4	1,396	0.04
evict non-code	45.5	2,430	2.49
span-overlap	22.1	626	4.69
span-overlap + non-code	67.6	3,056	2.94
StateTape	**74.2**	**1,595**	**6.18**
O2: retrieval	Found (%)	Added / DP	RE ↑
RETRIEVE-NOTHING	0.0	0	–
whole next-edit file	38.4	8,692	4.42
whole files in context	50.9	64,592	0.79
StateTape	**68.1**	**6,502**	**10.47**

Table 2: CE and RE on TraceBench. Cleared: unneeded tokens cleared; Lost: needed tokens lost; Found: missing tokens retrieved; DP: decision point.

5 CONCLUSION

Context management is a natural remedy for running coding agents over long horizons, since it bounds what the agent carries across requests. However, a coding agent falsifies its own evidence at every write, and history-based maintenance cannot tell from the text alone which records are stale and which code is needed. In this paper, we present StateTape, a novel and scalable framework that rewrites a coding agent's context at every write rather than by age or size. We model the repository as a symbol-level code graph and build a tape of graph diffs upon it, which turns staleness from an inference about text into an observation of the agent's writes. At each write, the tape nominates the records the write could have falsified, and a manager subagent built on a small LLM settles what the write log alone cannot. We also analyze StateTape theoretically and build TraceBench, a benchmark that labels what an agent holds against what it needs. Extensive experiments with six coding agents on three edit-heavy benchmarks demonstrate that StateTape clears falsified records and retrieves needed code effectively. It thus attains the highest resolve rate in every setting at marginal cost.

AI USE STATEMENT

In this work, we used generative AI tools solely to polish the writing of the manuscript: grammar and spelling correction, minor rewording for concision, and LaTeX formatting fixes. We did not use generative AI tools for research ideation, method design, theoretical analysis, implementation

of StateTape or of the baselines, experiment design or execution, data analysis, or literature search and citation generation; all such contributions are the authors' own. All AI-assisted edits were made at the sentence level on text the authors had already written, and every suggestion was reviewed and accepted or rejected by the authors, who verified that no claim, number, or citation was altered. Large language models otherwise appear in this work only as the systems under study, i.e., the coding agents and the manager model evaluated in our experiments. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, 2025.

Zhuofu Chen, Rui Pan, Yinwei Dai, and Ravi Netravali. Slipstream: Trajectory-grounded compaction validation for long-horizon agents. *ArXiv*, abs/2605.08580, 2026. URL https://api.semanticscholar.org/CorpusID:288253859.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, et al. Swe-marathon: Can agents autonomously complete ultra-long-horizon software work? *arXiv preprint arXiv:2606.07682*, 2026.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue (eds.), *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 3433–3445, Torino, Italia, May 2024. ELRA and ICCL. URL https://aclanthology.org/2024.lrec-main.305/.

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*, volume 2024, pp. 54107–54157, 2024.

Qirui Jin, Lingching Tung, Kenan Li, Qiyang Shi, Yushi She, Huanzhong Jia, Harrison Zhao, Kejing Xia, Zhenbang Du, Yi-Kai Zhang, Jia-Xin Pei, Zhenyu Zhang, Zhenghao Qi, Yuyan Duan, Wenke Lee, and Zijian Jin. Chainswe: Benchmarking coding agents on multi-bug software maintenance. *ArXiv*, abs/2607.02606, 2026. URL https://api.semanticscholar.org/CorpusID:289923029.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring ai ability to complete long software tasks. *Advances in Neural Information Processing Systems*, 38:92213–92266, 2026.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents. *arXiv preprint arXiv:2602.05892*, 2026a.

Xiaozhe Li, Tianyi Lyu, Yizhao Yang, Shan Liang, Siyi Yang, Ligao Zhang, Zhuoyi Huang, Qingwen Liu, and Yang Li. Escaping the context bottleneck: Active context curation for llm agents via reinforcement learning. *ArXiv*, abs/2604.11462, 2026b. URL https://api.semanticscholar.org/CorpusID:287432965.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as llm summarization for agent context management. *ArXiv*, abs/2508.21433, 2025. URL https://api.semanticscholar.org/CorpusID:280984917.

Jiacheng Liu, Xiaohan Zhao, Xin-Yi Shang, and Zhiqiang Shen. Dive into claude code: The design space of today's and future ai agent systems. *ArXiv*, abs/2604.14228, 2026. URL https://api.semanticscholar.org/CorpusID:287508324.

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, 2024a.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. *ArXiv*, abs/2512.22087, 2025a. URL https://api.semanticscholar.org/CorpusID:284275525.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model, 2024b. URL https://arxiv.org/abs/2406.07003.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. CodexGraph: Bridging large language models and code repositories via code graph databases. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 142–160, Albuquerque, New Mexico, April 2025b. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.7. URL https://aclanthology.org/2025.naacl-long.7/.

Hongtao Lyu, Dingyan Zhang, Mingyu Wu, Xingda Wei, and Hai-Bo Chen. Coagent: Concurrency control for multi-agent systems. *ArXiv*, abs/2606.15376, 2026. URL https://api.semanticscholar.org/CorpusID:289301286.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024.

Bardia Mohammadi, Lars Henning Klein, Aman Chadha, Akhil Arora, and Laurent Bindschaedler. The working set of a coding agent: Coherence debt in repository-scale tasks. 2026. URL https://api.semanticscholar.org/CorpusID:291182756.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In *International Conference on Learning Representations*, volume 2025, pp. 30098–30121, 2025.

Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, et al. Locobench-agent: An interactive benchmark for llm agents in long-context software engineering. *arXiv preprint arXiv:2511.13998*, 2025.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in llms. In *International Conference on Learning Representations*, volume 2026, pp. 35184–35211, 2026.

Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Linchao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks, 2025. URL https://arxiv.org/abs/2505.16901.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The openhands software agent sdk: A composable and extensible foundation for production agents. *ArXiv*, abs/2511.03690, 2025b. URL https://api.semanticscholar.org/CorpusID:282758003.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiao-Dong Gu. Swe-pruner: Self-adaptive context pruning for coding agents. *ArXiv*, abs/2601.16746, 2026. URL https://api.semanticscholar.org/CorpusID:284981955.

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):801–824, 2025.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Ying-Fei Xiong. Reducing cost of llm agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3:1241 – 1263, 2025. URL https://api.semanticscholar.org/CorpusID:281675918.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Lu Yi, Runlin Lei, Liuyi Yao, Yuexiang Xie, Yu-Yang Li, Wenhao Zhang, Zhewei Wei, Yaliang Li, and Jian-Yun Nie. Learning agent-compatible context management for long-horizon tasks. *ArXiv*, abs/2605.30785, 2026. URL https://api.semanticscholar.org/CorpusID:288814268.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Fuwei Zhang, Yanzhao Zhang, Mingxin Li, Dingkun Long, Lexiang Hu, Pengjun Xie, Zhao Zhang, and Fuzhen Zhuang. Core-bench: A comprehensive benchmark for code retrieval in the era of agentic coding. *arXiv preprint arXiv:2606.11864*, 2026.

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*, pp. 1592–1604, 2024.

Ming-Wei Zheng, David OBrien, Siwei Cui, Pardis Pashakhanloo, Rajdeep Mukherjee, Myeongsoo Kim, and Sachit Kuhar. Corvus: Context optimization and reduction via underlying synchronization for llm coding agents. *ArXiv*, abs/2607.22711, 2026. URL https://api.semanticscholar.org/CorpusID:290622531.

A Proofs for Section 3.4

Notation. Write $S(C; w) = \{r \in C : \neg\phi_r(w)\}$ for the records of C that are false of w, so $S_t = S(C_t; w_t)$, and recall the maintained context $C_t^+ = (C_t \setminus D_t) \cup A_t$ of Eq. 8. A record is **tracked** if its referent is a set of symbols (a read or an edit) or a repository state (a test result); search results and assistant text are untracked and are left to the manager's off-tape eviction. For a set of records C, C^{tr} denotes its tracked records. We extend $\mathcal{N}_t$ to contain every state-referent

record with a write in $(\tau_r, t]$, the records that RERUN acts on. The *local view* $\lambda_w(v)$ of a symbol v is its body $b_w(v)$ together with its outgoing edges in E_w, or $\perp$ if $v \notin V_w$. We write $\widetilde{C}_t$ for the context sent to the main agent at step t and $\Lambda = \sum_{t=1}^{T} n\big(S(\widetilde{C}_t^{\mathrm{tr}}; w_t)\big)$ for the stale tracked tokens it is sent over the run.

Assumption 1 (Symbol locality)**.** *For every record r with a symbol referent, $\rho_r \subseteq V_{w_{\tau_r}}$, and the truth of r depends on the repository only through the local views of its symbols: if $\lambda_w(v) = \lambda_{w'}(v)$ for all $v \in \rho_r$, then $\phi_r(w) = \phi_r(w')$.*

Assumption 1 is the symbol granularity of §3.3.2 stated as a condition on ϕ_r. It fails when a function's behavior changes through state that belongs to no symbol, such as a module-level constant.

Theorem 1(b) follows from the set inclusion, valid for every manager output,

$$S\big((C_t^{+})^{\mathrm{tr}}; w_t\big) \;\subseteq\; \big(N_t^{\mathrm{KEEP}} \cap S_t\big) \;\cup\; S(R_t^m; w_t), \tag{9}$$

whose last term is empty when no retrieved region covers a symbol of $\mathcal{X}_{(0,t]} \cup \partial\mathcal{X}_{(0,t]}$.

Throughout, $\mathcal{X}_u$ and $\partial\mathcal{X}_u$ compare $\Phi(u-1)$ with $\Phi(u)$ as in Eq. 6, and a record is true of the state it was observed in, $\phi_r(w_{\tau_r}) = \mathrm{true}$.

Theorem 1(a). Fix a tracked record $r \in C_t$ with $r \notin \mathcal{N}_t$. If r has a state referent, no write lands in $(\tau_r, t]$, and since observations leave the state fixed, $w_t = w_{\tau_r}$ and $\phi_r(w_t) = \phi_r(w_{\tau_r}) = \mathrm{true}$. Otherwise ρ_r is a set of symbols and $\rho_r \cap (\mathcal{X}_{(\tau_r,t]} \cup \partial\mathcal{X}_{(\tau_r,t]}) = \emptyset$. We show by induction on $u \in (\tau_r, t]$ that every $v \in \rho_r$ satisfies $v \in V_u$ and $\lambda_{w_u}(v) = \lambda_{w_{u-1}}(v)$. By Assumption 1, $v \in V_{\tau_r}$, and by the induction hypothesis $v \in V_{u-1}$. Since $v \notin \mathcal{X}_u$, v was not removed, so $v \in V_u$, and its body is unchanged, $b_u(v) = b_{u-1}(v)$. If an outgoing edge (v, x) were removed from E_{u-1} or added to E_u, then v would be an endpoint of a changed edge that lies in $V_{u-1} \cap V_u$ and outside $\mathcal{X}_u$, so $v \in \partial\mathcal{X}_u$, a contradiction. Hence $\lambda_{w_u}(v) = \lambda_{w_{u-1}}(v)$. Chaining over u gives $\lambda_{w_t}(v) = \lambda_{w_{\tau_r}}(v)$ for every $v \in \rho_r$, and Assumption 1 gives $\phi_r(w_t) = \phi_r(w_{\tau_r}) = \mathrm{true}$, so $r \notin S_t$. The argument uses only that the local views at τ_r and t agree. A rule that compares each record's outgoing edges at the two ends of its window, as the implementation does, is therefore sound as well, and the incoming-edge part of $\partial\mathcal{X}$ only adds nominations.

Theorem 1(b). Split C_t^{+} into $C_t \setminus D_t$ and A_t. Take a tracked $r \in C_t \setminus D_t$. If $r \notin \mathcal{N}_t$, then $r \notin S_t$ by (a). If $r \in \mathcal{N}_t$, then r received exactly one action, the manager's or the default. Since D_t contains $N_t^{\mathrm{DROP}} \cup N_t^{\mathrm{REFRESH}} \cup N_t^{\mathrm{RERUN}}$, $r \in N_t^{\mathrm{KEEP}}$. The off-tape evictions D_t^m only remove records. Now take $r \in A_t$. A tombstone states that the named symbols changed after τ_r, a fact about writes already made, so it holds at w_t. A refreshed body is sliced from w_t, so it is true of w_t. A RERUN stub is an instruction and makes no claim about the repository. The remaining records of A_t are R_t^m, which contribute $S(R_t^m; w_t)$. This proves Eq. 9, and the token bound follows by applying $n(\cdot)$. A retrieved region is a symbol-referent record observed in w_0, so $\tau_r = 0$; if its symbols avoid $\mathcal{X}_{(0,t]} \cup \partial\mathcal{X}_{(0,t]}$, part (a) applied to it shows that it is true of w_t.

Theorem 2(a). The drift sets are computed from parse trees and resolved call edges, so no model is called. An observation leaves w unchanged, so $\mathcal{X}_u = \partial\mathcal{X}_u = \emptyset$, and a record that enters on an observation step has an empty window. A write changes bytes only in the files it touches, and whether a symbol exists and what its body is are functions of its own file, so updating the index of Eq. 10 requires parsing only the written files. With the index stored in a hash map, $v \in \mathcal{X}_{(\tau_r,t]}$ holds exactly when $\mathrm{last}_t(v) > \tau_r$, one lookup per (r, v) pair. For the edge term, the remark at the end of the proof of Theorem 1(a) lets each record compare the outgoing edges of its symbols at τ_r with those at t, which costs $\mathcal{O}(\deg^{+}_{w_t}(v)) \subseteq \mathcal{O}(d)$ per symbol. Edges are a function of the tree, so they are re-resolved only after a write. Summing over held records gives the bound.

Theorem 2(b). Let a manager call at step t cost at most $\gamma_0 + \beta\,|\mathcal{N}_t|$ tokens. The manager is invoked only at steps in $\mathcal{T}_N$, so $K^{\mathrm{mgr}} \le \sum_{t\in\mathcal{T}_N}(\gamma_0 + \beta\,|\mathcal{N}_t|) = |\mathcal{T}_N|\,(\gamma_0 + \beta\,\bar{N})$. Each request sends its whole context, cache reads included, and compaction is disabled, so $K^{\mathrm{agt}} \ge \sum_{t=1}^{T} n(\widetilde{C}_t) \ge T\,n_{\min}$. Dividing gives $K^{\mathrm{mgr}}/K^{\mathrm{agt}} \le \frac{|\mathcal{T}_N|}{T} \cdot \frac{\gamma_0+\beta\,\bar{N}}{n_{\min}} \le (\gamma_0 + \beta)\,\frac{|\mathcal{T}_N|\,\bar{N}}{T\,n_{\min}}$, using $|\mathcal{N}_t| \ge 1$ on $\mathcal{T}_N$, which is the bound in Theorem 2(b). The live manager meets the hypothesis: its prompt holds one row per nomination (location and at most five symbol names), previews of at most 40 other records

Algorithm 1 StateTape maintenance step at a write (nominate and default)

Require: context C_t; tape state after $a_t \in \mathcal{A}_{\text{wr}}$; repository state w_t
Ensure: nominations $\mathcal{N}_t$, each paired with a tape default

append to the tape the increment $\mathcal{X}_t, \partial\mathcal{X}_t$ that a_t produced ▷ Eq. 6
$\mathcal{N}_t \leftarrow \emptyset$
for each held record $r \in C_t$ with a symbol referent **do**
  $H_r \leftarrow \rho_r \cap (\mathcal{X}_{(\tau_r,t]} \cup \partial\mathcal{X}_{(\tau_r,t]})$ ▷ Eq. 7
  if $H_r = \emptyset$ **then continue**
  end if
  if $H_r \cap \mathcal{X}_{(\tau_r,t]} \cap V_t \neq \emptyset$ **then**
    $\mathcal{N}_t \leftarrow \mathcal{N}_t \cup \{(r, H_r, \text{REFRESH})\}$
  else
    $\mathcal{N}_t \leftarrow \mathcal{N}_t \cup \{(r, H_r, \text{DROP})\}$
  end if
end for
return $\mathcal{N}_t$

at 80 characters each, and the first 1,500 characters of the task, and its reply is capped at 2,000 tokens, so γ_0 does not grow with the context.

Theorem 2(c). Without maintenance no record leaves the context, and by Eq. 1 a stale record stays stale. Order the writes $s_1 < \cdots < s_W$. The at least δ tokens that write s_k falsifies are therefore in every request after s_k, and there are at least $W - k + 1$ such requests, the one that returns the result of s_k and one per later write. Hence $\Lambda \geq \sum_{k=1}^{W} \delta\,(W - k + 1) = \delta\, W(W+1)/2 = \Omega(\delta W^2)$. With StateTape, staleness arises only at writes, the context of every request after a write is the maintained C_t^+, and records added by observations since then are true of the unchanged state. Theorem 1(b) bounds each term of Λ, and summing gives $\Lambda \leq \sum_t n(N_t^{\text{KEEP}} \cap S_t)$ when retrievals cover no drifted symbol. The default policy keeps no nomination and retrieves nothing, so $\Lambda = 0$.

B METHOD DETAILS

Maintaining the code graph. The graph is maintained per write rather than rebuilt per task. A write changes bytes only in the files it touches, and whether a symbol exists and what its body is are functions of its own file, so only the written files are re-parsed. Call edges are re-resolved only after a write, and the edge comparison is restricted to edges incident to a symbol in a written file, so nondeterminism in call resolution, which lives in files nothing touched, cannot register as change. Appendix D gives the per-symbol index that answers every query of the tape exactly from these incremental updates, without materializing the graph at each write.

Drift windows. The tape's entries are per-write increments rather than a running comparison against the repository the run started from, so every change is attributed to the write that made it. For a fixed left endpoint, the window union $\mathcal{X}_{(s,t]}$ only grows in t, which is Eq. 1 restated at symbol granularity. Symbols in $V_t \setminus V_{t-1}$ are excluded from Eq. 6: a symbol the write created did not exist for any earlier record to have captured, so it cannot falsify one. Records whose referent is a repository state rather than a set of symbols, such as test results, are nominated whenever a write falls inside their window.

Maintenance actions. Agent transcripts append rather than mutate, so the tape acts by rewriting a nominated record in place. The tombstone left by DROP has constant size and names the referent and the symbols that changed, so the agent learns that it lost the evidence and why, and can re-read rather than proceed on a gap it cannot see. REFRESH slices the current bodies of the drifted symbols from w_t; in the terms of §3.2, the referent appears in both D_t and A_t. A test result observed before later writes cannot be repaired by any current text, only by executing the test again, which is why state referents take RERUN. KEEP is for nominations that over-fired. The defaults alone form a complete deterministic policy.

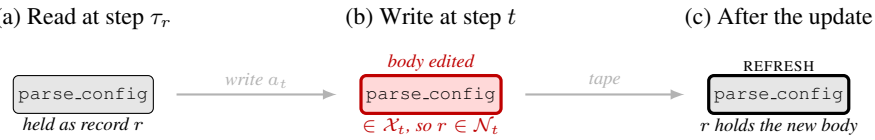


Figure 6: Example 1, a body edit. Gray fill marks what the agent holds. (a) The agent reads `parse_config`, which enters the context as record r. (b) A later write edits its body, so `parse_config` $\in \mathcal{X}_t$ and r is nominated. (c) Since the symbol still exists, the tape assigns REFRESH and r is replaced by the current body.

The per-write procedure. Algorithm 1 runs after every write. Line 1 appends the write's drift sets to the tape, the only step that reads the repository. Lines 3–4 intersect each held record's referent with the drift in its own window, and Line 5 skips records that no drifted symbol touches, so a record is examined only against the writes that followed it. Lines 7–11 pair the nomination with REFRESH when some covered symbol still exists and changed its body, so a current body can be produced, and with DROP when every hit is a removed symbol or a caller whose dispatch moved. The resulting tuple records the symbols that justify the nomination, which is what the manager later reads.

The manager's evidence. The manager is not handed the repository or the transcript. Its skills (Table 7, Appendix G) expose four kinds of evidence: the nominations, with their reasons, their defaults and the text a REFRESH would inject; the drift, meaning which symbols the writes changed, which callers lie within two hops of them in the code graph, and which held records a given symbol falsifies; the held context, with each record's text and the step and action that produced it, from which a record that is finished with is told apart from one still in use; and the base-state source, to check a region before retrieving it. Every skill is answered by the graph and the tape without a model. No skill returns the reference patch, and none returns a staleness verdict.

Retrieval labels. The manager ranks graph candidates, checks the ones it keeps against source, and labels each retrieval with the rule that admits it: the edit site, a definition the edit references, a site that must stay consistent with the change, or a contract the change must satisfy. These are the rules G1 to G4 under which TraceBench labels gold regions (Appendix E).

C Worked Examples

The examples below illustrate the tape (§3.3.2) and the manager (§3.3.3). They are constructed for exposition rather than taken from a logged run. Figures 6 and 7 draw them in the style of Figure 1.

Example 1: a body edit. Early in a run the agent reads `parse_config(path)`, whose body returns the parsed dictionary, and the read enters the context as a record r whose referent covers `parse_config`. Several steps later the agent edits that body to validate the dictionary against a schema and raise on missing keys. The symbol exists in both states and its body differs, so `parse_config` $\in \mathcal{X}_t$ by Eq. 6. The write lies inside the window $(\tau_r, t]$, so $r \in \mathcal{N}_t$ by Eq. 7. Because the symbol still exists and its body changed, Algorithm 1 assigns REFRESH, and the record's text is replaced by the current body, so the agent no longer holds a version that never raises.

Example 2: caller drift. Suppose instead the agent edits a module and changes the line `from .json_io import load` to `from .yaml_io import load`. A function `read_config` in that module that calls `load()` is byte for byte what it was, and a record holding its text still matches the file. The behavior the record describes has nevertheless changed, because the call now reaches a different function. The import line belongs to no symbol body, so $\mathcal{X}_t$ does not register the write. The edge from `read_config` moves from `json_io.load` to `yaml_io.load`, so the pair lies in $(E_{t-1} \setminus E_t) \cup (E_t \setminus E_{t-1})$ and $\partial\mathcal{X}_t$ contains `read_config`. The record is nominated, and since its only hit is in $\partial\mathcal{X}_t$, the default is DROP: re-slicing the unchanged body would restore the same text, while the tombstone tells the agent that the call now resolves elsewhere. Any change that alters dispatch without touching the caller's text, such as a changed import, a new override in a subclass or a removed definition that a call now falls through to, reaches the tape through $\partial\mathcal{X}_t$ in the same way.

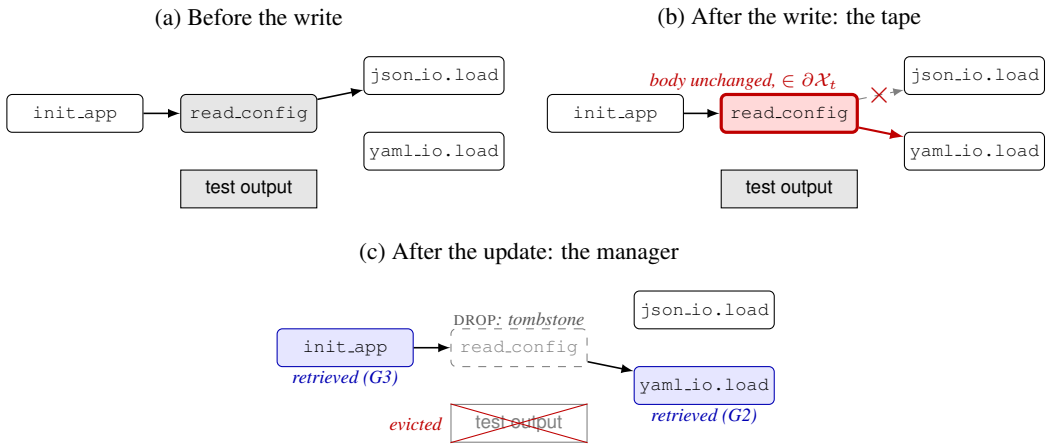


Figure 7: Examples 2 and 3, an import change. Gray fill marks what the agent holds. (a) The agent holds `read_config` and the output of an earlier test run. (b) A write changes the import behind `load()`. The body of `read_config` is unchanged, but its call edge moves from `json_io.load` to `yaml_io.load`, so it enters $\partial\mathcal{X}_t$ and is nominated. (c) The manager keeps the default DROP and leaves a tombstone, evicts the test output that no later step uses, and retrieves the new callee (G2) and the caller `init_app` (G3).

Example 3: the manager after the import change. Continuing Example 2, the manager is invoked because the tape carries a nomination, and it makes the three decisions of §3.3.3. For the nomination, it accepts the default DROP of `read_config`. For eviction, it finds through the held-context skills the output of a test run from an earlier fix to an unrelated module; no write falsified it, so the tape does not nominate it, but no later step uses it, and the manager adds it to D_t^m. For retrieval, it starts from `read_config` and follows call edges: the new callee `yaml_io.load` is a definition the next edit references (G2), and `init_app`, which calls `read_config` and indexes the returned dictionary must stay consistent with the new format (G3). It checks both against source and adds them to R_t^m. The resulting update is Eq. 8 with one tombstone, one eviction and two retrievals.

D Implementation Details

Baselines. All four baselines are implemented as modes of the same shim as StateTape. **Observation masking** replaces tool outputs older than the last 10 turns with a placeholder. **LLM summary** folds the history into a summary once 21 turns have accumulated and keeps the last 10. It uses the main agent's own model. Both follow the settings of Lindenbauer et al. (2025). **CORVUS** (Zheng et al., 2026) registers every file the agent reads and re-injects its current contents on each request. The **text-only manager** follows the style of AgentDiet (Xiao et al., 2025). It uses the same manager model as StateTape, but it sees only previews of recent reads and searches, and it can only delete them.

Execution environment. Each instance runs in its own Docker container with a 90-minute limit (30 minutes per SlopCodeBench checkpoint) and no turn limit. All runs use a single host with an Apple M5 Pro (18 CPU cores) and 48 GB of memory.

Evaluating a window without keeping the graphs. Materializing Φ at both ends of every write states Eq. 6 directly, but it would put a repository-wide static analysis inside the agent's request path, and a window query would need every graph in the run rather than two. Neither is necessary. Each write is parsed only for the symbols it touched, and those are folded into a per-symbol last-write index

$$\mathrm{last}_t(v) \;=\; \max\{\, s \le t \;:\; v \in \mathcal{X}_s \,\}, \qquad \mathrm{last}_t(v) = -\infty \text{ if no write has touched } v, \tag{10}$$

which is all a window query needs, since $v \in \mathcal{X}_{(s,t]}$ exactly when $\mathrm{last}_t(v) > s$. The index is an exact evaluator of Eq. 6, not an approximation of it, and it refers to no base state, so files the

Trajectories and decision points		**Left region $C_t \setminus G_t$ (O1)**	
instances / repositories	48 / 7	unneeded tokens	5,349,888
steps / trajectory (median / max)	71 / 356	share of held tokens	73.4%
gold hunks / instance (median / max)	21 / 379	stale	72.3%
decision points (all pre-edit)	403	consumed	10.4%
instances with ≥ 1 point	43	distractor	17.3%
depth in steps (median / max)	75 / 278	tokens in records touching G_t	1,943,649
Held context C_t		**Right region $G_t \setminus C_t$ (O2)**	
context records scored	36,873	gold tokens	697,727
held tokens	7,293,537	share of gold not held	58.5%
token encoding	`cl100k`	coverage $\mathrm{Rec}(C_t)$	0.415
judge	`gpt-5.4`, seed 0	gold regions G3 / G4	1,338 / 595
		oracle tokens / point	1,012

Table 5: TraceBench at a glance. Every quantity is a token count under one pinned encoding. The two right-hand blocks are the two regions of Figure 3 and the two surrogates of Eq. 2. The left region is scored per record, so a held record that overlaps G_t counts in full and each repeated read counts again. The right region is scored over deduplicated gold spans. The two intersections $C_t \cap G_t$ therefore differ in size, 1,943,649 tokens on the left against about 289,600 on the right.

agent creates mid-run are handled like any other. The edge term is refreshed only over the symbols currently under watch and only when the tree has moved since the last rebuild, which is likewise exact, because edges are a function of the tree and the watched seeds and a step that wrote nothing cannot change them.

E TraceBench Annotation and Scoring

Reading Table 5. §3.1.2 discusses what the two regions contain. For scoring, note that the left region is 73.4% of held tokens at a pre-edit decision, so O1 has a high base rate and F_1 alone rewards indiscriminate deletion; precision at fixed retained-useful mass is the informative view.

Source trajectories and decision points. Each raw event stream is normalized into typed steps by one parser per agent wire format; nothing downstream of the parser is agent-specific. Sub-agent activity is excluded from C_t on purpose: only a sub-agent's returned summary enters the main thread, so a file it rewrote and did not surface is genuinely absent from the main agent's context and is a real source of both regions. Typed steps are $\{\textsc{read}, \textsc{edit}, \textsc{run}, \textsc{search}, \textsc{say}\}$, carrying either a $(\mathit{file}, \mathit{span})$ or a command. A decision point t means the context is $s_1 \ldots s_t$ and the label describes the decision faced at s_{t+1}. Three anchor families are implemented: **P**, pre-edit, at $t = j - 1$ for each edit *burst* beginning at s_j, a maximal run of consecutive edits, so that a ten-line multi-edit to one function is one decision and not ten; **T**, post-test, at each genuine test run; and **F**, final, at $t = n$. The shipped label set is P only, because every pre-edit decision point has an imminent edit to anchor G_t on and T and F carry none. Of 551 pre-edit candidates, 148 are skipped where the imminent burst is purely shell or diagnostic and has no localized edit in a gold file to anchor on. Trajectory lengths reach 356 steps and reference patches 379 hunks; decision-point depth reaches 278 steps.

Gold regions. Each gold region is admitted under one of four rules, cited on the label: **G1**, the edit site, the enclosing definition of the imminent hunk in its pre-edit state; **G2**, a referenced definition the hunk calls, imports or subclasses; **G3**, the blast radius, an existing site that must stay consistent with the change; and **G4**, the contract, the interface, base class, registry or schema the change must satisfy. These are the rules the retrieval proposals of §3.3.3 are named for. Four bounds keep the set tight: co-location in a touched file without a traceable dependency is not gold, mechanical boilerplate is not gold, test code is never gold since it encodes the answer, and a region already held is the intersection of Figure 3 rather than the right region.

The held context C_t. C_t is materialized as an inventory of records, each $\{\mathit{id}, \mathit{kind}, \mathit{file}, \mathit{span}, \mathit{born_step}, \mathit{text}, n\}$, which is the record (c_r, ρ_r, τ_r) of §3.2 made con-

crete: file-region items, from reads and edits, carry a referent, while run outputs, search results and assistant text do not. All text is tokenized with a pinned `cl100k_base` encoding, and every quantity reported is a token count.

Outstanding hunks. A deterministic pre-pass parses the reference patch into source hunks, drops the test patch, and marks a hunk outstanding at t if no agent edit at or before t overlaps its base span. The judge then selects, among outstanding hunks, the one the imminent burst is positioned to make, and the G1–G4 sweep is anchored on that selection rather than on the patch as a whole.

Annotation protocol. The needed pass runs a per-file sweep over the repository at the base commit, reading files until every outstanding gold file is covered and emitting regions with a rule and a justification; the orchestration around it is deterministic rather than model-chosen, so coverage of the gold files is guaranteed rather than hoped for. The stale pass is trajectory-framed and works from the item record inventory with deterministic supersession flags pre-attached. Reproducibility is pinned throughout: endpoint, deployment, model version, prompt version and seed are echoed in the manifest, the request cache is content-addressed and authoritative so replay needs no key, and a prompt change re-namespaces the cache. Labels carry the transcript hash and are byte-reproducible.

Scoring. Predictions and labels are expanded to token sets, gold regions from the base-commit file slice and held records from their observed text, and overlap is resolved at the $(\mathit{file}, \text{token index})$ level, so a partially read gold region contributes only its unread tokens. The headline numbers are the two ratios of §3.2, CE (Eq. 3) against the left region and RE (Eq. 4) against the right, each micro-averaged over the decision points of a split and each paired with its volume; we also report the underlying token-weighted precision, recall and F_1 for O1 and raw tokens added per decision for O2, since those are the quantities the ratios are built from. Reference points are reported with every result: KEEP-ALL, EVICT-ALL, recency and age windows, and a span-overlap supersession detector for O1; RETRIEVE-NOTHING, whole-file heuristics and the oracle for O2.

Provenance and limitations. The labels are silver: deterministic rules plus an oracle model, with a human review queue built and a verification server in place but not yet filled, so scoring a detector measures agreement with the judge rather than with a person. The labeled set is one agent on one suite, and although the pipeline is harness-agnostic downstream of the parser, we do not claim the base rates transfer to a different scaffold. G_t is defined relative to the reference patch, so a correct solution the reference did not take is scored as not needed; this bounds O2's ceiling rather than distorting the ranking of policies against each other.

F Benchmark Comparison

Standard evaluation scores a run once, when it ends, while the gap between C_t and G_t moves at every write. A pass rate can say that a maintenance policy helped without saying which tokens it should have dropped at step 40, or which region of the repository it should have pulled in before the edit at step 41. Three families of benchmark bear on the gap. SWE-bench (Jimenez et al., 2024) and its descendants (Deng et al., 2025; Le et al., 2025; Orlanski et al., 2026; Desai et al., 2026) pair a repository issue with held-out tests and score a run by whether its terminal patch passes. Long-context and memory benchmarks (Maharana et al., 2024; Qiu et al., 2025) measure whether a model can find and use information placed inside a large given window. Localization benchmarks (Li et al., 2026a; Zhang et al., 2026; Chen et al., 2025) annotate the code an agent should have found for a task, which is an instantiation of G_t and therefore of the right region.

None of them supervises how an agent maintains its context over a long horizon. First, outcome-only supervision is too coarse: an instance yields one bit, and that bit is produced by the agent's model, so a correct eviction the model then ignores and an incorrect one it happens to survive are indistinguishable at the metric, and the effect of a maintenance policy reaches the score only through a capability the policy does not control. Our own runs (§4) show how loose the coupling is: StateTape rewrites the context at nearly every write, and the resolve rate moves by a few points, but that one bit per instance never says which of those rewrites was the right one. Second, the evidence in long-context benchmarks is static with respect to the model: the distractors were placed by the benchmark, not caused by the model, and nothing the model does falsifies what it was shown,

Family	scored unit	label site	self-falsifying	annotated regions $C_t \setminus G_t$	annotated regions $G_t \setminus C_t$
Issue resolution	terminal patch	task	yes	no	no
Long context / memory	answer	task	no	no	no
Retrieval / localization	retrieved set	task	no	no	yes
TraceBench	context state	decision point	yes	yes	yes

Table 6: Where the evidence lifecycle falls between benchmark families, represented by SWE-bench and SWE-bench Pro (Jimenez et al., 2024; Deng et al., 2025), LoCoMo and LoCoBench-Agent (Maharana et al., 2024; Qiu et al., 2025), and ContextBench and LocAgent (Li et al., 2026a; Chen et al., 2025). *Self-falsifying* asks whether the agent's own actions can make its evidence wrong, the condition that makes staleness a long-horizon property rather than a retrieval failure (Eq. 1). The last two columns are the left and right regions of Figure 3, what a maintenance policy should evict and what it should retrieve.

Table 7: The manager's skills. Nine read-only queries against the tape, and one that commits the decision. The manager is given a budget of 25 calls per invocation and a graph depth of 2.

Group	Skill	Returns
Nominations	`list_candidates`	$\mathcal{N}_t$ with each reason
	`candidate_fresh`(r)	$\mathrm{fresh}(r)$, the text a REFRESH would inject
Drift	`symbols_touched`	the symbols the run's writes changed
	`dependents`(v, h)	callers of v
	`evidence_for`(v)	$\{\, r \in C_t : v \in \rho_r \,\}$, the records v falsifies
Context	`list_context`	C_t with kind, referent, birth step and token cost
	`read_context`(r)	the untruncated text of one held record
	`provenance`(r)	τ_r and the action that produced r
Repository	`read_file`(p, s, e)	verify a region before retrieving it
Decision	`submit`	$(D_t^m,\ R_t^m)$

whereas a coding agent's own writes falsify what it read and are the only source of its staleness (§3.2). Third, localization labels are attached to a task rather than to a step, so a policy that acts at every write has no per-write target and the gold context cannot move as the repository does; and none of these datasets annotates $C_t \setminus G_t$, so the left region, the one that carries the token cost, is unmeasured. Their selection criteria also run the other way: Loc-Bench excludes issues whose patch touches more than five files or ten functions, which is the right filter for one-shot localization and removes exactly the write-dense instances in which stale evidence accumulates. Table 6 summarizes the comparison.

G MANAGER SKILLS

Table 7 lists the skills of §3.3.3: nine read-only queries answered by the code graph and the tape, and one skill that commits the decision.