WHEN REPAIRS STOP BEING RECIPES: RE-GROUNDING REPAIR MEMORY FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent coding-agent systems store patches and debugging strategies from resolved issues as *repair memory*, then retrieve these records to guide later repairs and avoid repeated investigation. However, reusing a past fix can misdirect the next repair: a record may identify useful code while recommending an edit that no longer resolves the current issue. This *repair applicability gap* raises a question: how can an agent benefit from an earlier investigation when its fix no longer works? To study this question, we introduce REPAIRMEM, a benchmark that pairs 99 current repair tasks with related historical repairs, covering both controlled faults and real repository issues. We evaluate five coding models on these tasks and find that giving them historical records unchanged usually reduces repair success compared with providing no memory. To make these records useful for the current task, we propose REGROUND. Before the agent begins its repair, REGROUND checks the historical patch against the current tests, uses the outcome to revise the historical repair advice, and supplies relevant code from the current repository to guide further debugging. This simple procedure requires at most one replay of the historical patch and no additional model calls for preparation. Across five models and 99 repair tasks, REGROUND improves aggregate repair success by 16.4 and 27.7 percentage points over memory-free repair and direct historical reuse, respectively, while using 20.2% and 22.5% fewer model calls.

1 INTRODUCTION

Repository-level repair requires a coding agent to locate relevant code, understand a failure, and validate a proposed change (Jimenez et al., 2024; Yang et al., 2024; Zhang et al., 2024; Bouzenia et al., 2025). During the process, the agent identifies which functions are involved, how they interact, and which tests reproduce the failure. A later issue may involve some of the same code, making these findings potentially reusable. Earlier program-repair methods likewise exploit past changes and recurring fix patterns to guide new repairs (Le et al., 2016; Bader et al., 2019; Koyuncu et al., 2020). Recent coding-agent systems store patches, debugging records, and lessons from previous repairs as *repair memory*, then retrieve them when addressing related issues (Wang et al., 2026; Mu et al., 2026; Liu et al., 2026). These records can reduce repeated investigation, but effective reuse still requires deciding what can be carried forward to the current task.

Although retrieving a past repair record from memory can help an agent locate relevant code, following its suggested edit can misdirect the current repair (Lindenbauer et al., 2025). The historical patch may already be part of the current code, or it may apply cleanly without fixing the current issue. We call this the *repair applicability gap*: finding a related past repair does not establish whether its recorded edit resolves the current issue. The challenge is to reassess the stored edit without discarding the useful context that may still guide the current investigation.

More critically, related failures can nevertheless require changes in different parts of the code. Figure 1 illustrates this situation schematically with configuration paths that begin with a tilde (~). The historical patch uses `os.path.expanduser` in `config.py`'s `expand_path` function to replace the tilde with the user's home directory, fixing the earlier issue. In the current issue, however, paths in nested includes remain unexpanded because they are handled by `parse_include` in `parser.py`. Applying the old patch leaves the current test failing: path expansion is needed 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.

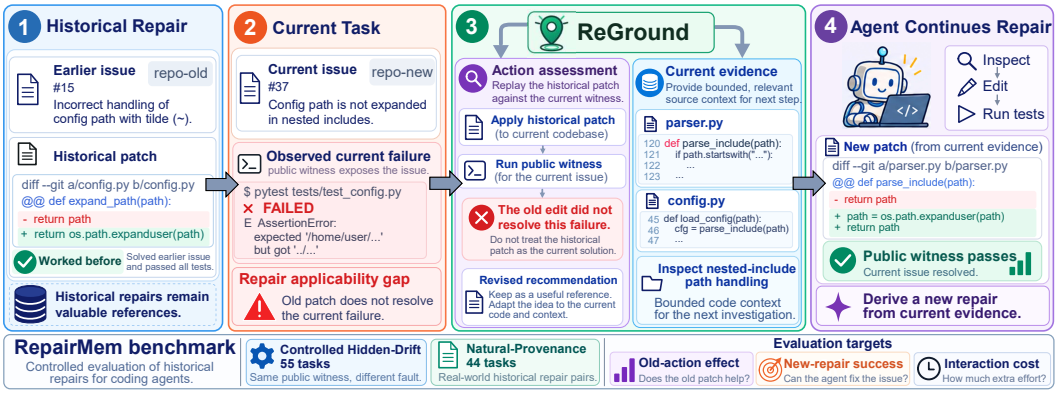

Figure 1: **Overview of RepairMem and ReGround.** A historical repair may remain useful even when its stored patch no longer solves a related current task. REGROUND uses current execution evidence to prepare the historical repair for reuse before the existing agent continues the investigation and derives a new patch. RepairMem evaluates this process on 55 Controlled Hidden-Drift and 44 Natural-Provenance tasks, measuring old-action effect, new-repair success, and interaction cost.

this different function. The historical record still provides a useful repair idea, but the agent must inspect the current code and failure to determine where to apply it.

To evaluate this reuse directly, we introduce REPAIRMEM, a benchmark of 99 executable tasks, each pairing a current issue with a related historical repair. It includes 55 controlled tasks across three repositories and 44 naturally sourced tasks across ten repositories. The controlled tasks test whether previously successful patches also fix different, deliberately introduced faults, while the naturally sourced tasks pair earlier developer repairs with later repository issues. RepairMem separately measures the historical patch's current effect, downstream repair success, and the interaction required to produce that repair. To help agents benefit from these historical records even when the old patches no longer solve the current issue, we propose REGROUND, an execution-grounded framework that adapts past repair advice to the current task. Its key idea is to connect assessment of the old patch with preparation for the remaining repair: the replay outcome revises the historical advice, while current failure evidence identifies code for further investigation. This allows useful historical context to guide the agent while redirecting its attention when the old edit fails. The resulting advice and code are delivered before the agent's first model call, requiring at most one historical replay and no additional model calls for preparation. Our main contributions are summarized as follows:

- **RepairMem: the first executable benchmark for repair-memory reuse.** We formulate the *repair applicability gap* and introduce controlled and naturally sourced tasks that distinguish the stored patch's current effect from the agent's subsequent repair and the work required.
- **ReGround: execution-grounded adaptation of repair.** REGROUND reassesses the stored patch against the current task and grounds the remaining investigation in current code, allowing historical experience to guide repair without treating its old edit as the current solution.
- **Empirical evaluation of repair-memory reuse.** Across five models and 99 tasks, REGROUND increases aggregate repair success by 16.4 percentage points over agents without repair memory and 27.7 points over agents given unchanged historical records, while reducing total model calls by 20.2% and 22.5%, respectively. Comparisons on controlled tasks further show equal or higher success with fewer calls when both conditions receive the same current evidence.

2 REPAIRMEM: EVALUATING REPAIR MEMORY REUSE

RepairMem evaluates historical repair reuse when the stored patch does not provide an established solution to the current task. It measures the patch's current effect separately from the agent's subsequent repair and interaction cost. Two tracks establish the relationship between past and current repairs: Controlled Hidden-Drift constructs failures from a shared repository state, while Natural-Provenance pairs earlier developer repairs with later issues. Both provide support for the historical

repair, a reason to consult its record, and a current failure with a validated repair. Executable bug benchmarks similarly use reproducible program states and tests to distinguish faulty and repaired behavior (Just et al., 2014; Widyasari et al., 2020; Madeiral et al., 2019).

2.1 The Repair Applicability Gap

Tasks and actions. A current task is $x = (q, r, w)$, where q describes the issue, r is the repository checkout, and w is a disclosed, task-specific executable check, termed the *public witness*. Each task also supplies a historical repair record m and its exact patch a_H. We write $r[a]$ for the checkout after successfully applying an edit a. When the check yields a usable outcome, $w(r[a]) = 1$ denotes a pass and 0 a failure. The historical record and patch are fixed across memory-use comparisons; the agent produces its own edit a by inspecting, modifying, and testing the current repository.

Evidence and applicability. Each pair records what supports the earlier repair and why it is supplied for the current task. Historical support comes from an executed construction or an earlier developer repair. The connection to the current task is established through shared public behavior, a matched function, or documented source or test relationships. These connections explain why the record may be useful, but do not establish whether its edit resolves the current issue. The patch may be present or may address a different failure, while the record still identifies code worth inspecting. We call this the *repair applicability gap*: a related historical repair need not provide an effective current edit.

Observed current effect. RepairMem distinguishes the old edit's relationship to the checkout from the evidence obtained by execution. The stored edit may apply as a new change, already be incorporated, or be unavailable for replay. Replay tests whether the historical patch resolves the disclosed check, without attributing the cause of failure. Accordingly, PASS and FAIL summarize execution outcomes, PRESENT identifies an incorporated change, and UNAVAILABLE indicates that no usable replay result was obtained. The last case denotes missing evidence, not a negative verdict. These labels are action-scoped: they characterize the current effect of the stored patch, not the usefulness of the surrounding repair record.

2.2 Controlled Hidden-Drift Comparisons

The *Controlled Hidden-Drift* track changes the internal failure while keeping the public entry and witness fixed. Historical and current variants branch from a common passing checkout, maintaining a concrete connection between the tasks while allowing the required edit to differ. These are constructed branches, not consecutive Git revisions. Recent SWE-agent data pipelines likewise construct executable repository tasks by introducing test-visible faults (Yang et al., 2026).

Historical success. We construct an earlier failure around an executed consumer–provider relation. For example, a provider-side change replaces a scalar return value with a mapping, causing the existing consumer to fail. The historical patch modifies that consumer to read the `value` entry. In the resulting historical checkout r_H, the witness fails before this edit and passes after it: $w(r_H) = 0$ and $w(r_H[a_H]) = 1$. This establishes the earlier repair's success through execution.

Current-task validation. A separate internal mutation creates the current checkout r. We first confirm that it fails and that a reference repair a_C resolves the failure, giving $w(r) = 0$ and $w(r[a_C]) = 1$. We then evaluate the historical patch a_H on the same task. The primary construction also tests a_H after removing the current fault. If the restored checkout passes before the old patch but fails afterward, the patch introduces a regression under that check, rather than merely leaving an existing failure unresolved. The track contains 55 tasks across three repositories.

2.3 Natural-Provenance Repair Pairs

Developer-repair pairing. The *Natural-Provenance* track pairs earlier developer repairs with later SWE-bench issues (Jimenez et al., 2024). A pair must have a documented code or test relationship. The repairs either concern the same bounded function with a stable signature and shared identifiers, or involve distinct functions linked by a supported relation, such as a call or override. We exclude duplicate and import-only links; repository membership alone does not establish a pair. Historical support comes from the earlier developer repair rather than re-execution of its original failure.

Current-state validation. For each pair, we install the disclosed current tests, reproduce the failure, and verify that the current reference patch passes. A reverse-application check identifies historical changes already present in the checkout. Patches that apply as new changes are evaluated on the current witness. The 44 tasks span ten repositories and include already-present changes, unsuccessful replays, and unavailable outcomes. They complement the controlled construction by evaluating actual earlier repairs under these different current states.

3 REGROUND: EXECUTION-GROUNDED REPAIR MEMORY

REGROUND checks the stored repair against the current checkout, uses the resulting assessment to revise its recommendation, and supplies bounded current-code evidence for the next repair attempt. It produces revised historical context $\widetilde{m}$ and a *source frontier* F of current code excerpts. The assessment determines how the old edit is presented; the frontier supplies code to inspect. Both enter the agent's context before its first model call, leaving the harness responsible for a new repair.

3.1 REVISING THE HISTORICAL RECOMMENDATION

Checking the stored action. Let $e_0 = \mathrm{Run}(r, w)$ denote the current check's output and available repository-local failure evidence. When replay is available, REGROUND applies the exact historical patch and obtains $e_H = \mathrm{Run}(r[a_H], w)$. The same witness evaluates both the current failure and the attempted historical repair. Using the recorded diff makes this an observation about a_H, rather than about an edit inferred from the historical description. For natural records, reverse-application checking identifies already-present changes without replaying the patch.

Revising the recommendation. The outcome determines how the historical advice is revised. When replay fails, REGROUND withdraws the recommendation to reuse the stored edit. A passing replay supports it under the current witness, while an already-incorporated change provides no new modification. An unavailable replay supplies no new execution evidence about the edit. These judgments concern the specific action a_H: its failure does not rule out a different repair at the same location.

3.2 GROUNDING THE CURRENT INVESTIGATION

The assessment reports what the old patch achieved, but does not identify the next edit. REGROUND therefore constructs F by selecting code associated with the current failure: it identifies initial candidates from e_0, expands them through execution or source relationships, ranks them, and extracts their code from the current checkout. Repository-level systems similarly use retrieval, dependency structure, or code graphs to localize relevant source context (Zhang et al., 2023; Chen et al., 2025b).

Anchoring and expanding candidates. A failure frame identifies where an error is reported, but the relevant behavior involved may come from elsewhere. We use failure frames and error text to identify initial regions, then consider related code supported by execution or local source inspection. The historical location may guide this expansion without fixing where the investigation must begin.

With full traces, additional candidates are currently executed scopes in files identified through failure evidence, execution changes, or the historical location. Native extraction instead inspects supported subscript and call expressions near repository-local failure frames. It resolves one-hop provider relationships through keyed registries and module-level containers, exposing code that supplies values used at the failure location without collecting full execution traces.

Prioritizing and extracting evidence. Trace-based selection first selects scopes containing failure frames. It ranks the remaining candidates primarily by overlap with the current error text. Secondary cues include whether a scope executes only in the current run, line-hit differences between e_0 and e_H, and historical-owner signals. Strong matches to the terminal error are selected before the remaining slots are filled, allowing current error text to identify useful code even when replay produces little trace difference. Native selection prioritizes resolved providers over their enclosing failure-frame scopes. Selected scopes are deduplicated and their code is extracted from r. Each entry in F includes a code excerpt, its location, and the reason for selection. Limits on the number of scopes and total source length bound the initial context without restricting later repository inspection. The frontier provides code to investigate, not a proposed patch.

3.3 Delivering the Repair Context

The agent receives the task instructions together with $\widetilde{m}$ and F. After failed replay, the trace-based configuration retains the old action and location as assessed historical context, whereas the native configuration omits their text. Both revise the recommendation without restricting the agent's edit permissions. Appendix Table 10 specifies the rendering and fallback rules.

In a pytest example, applying the historical helper edit leaves a plugin-initialization error unresolved. ReGround reports this outcome and supplies the plugin-specification parser, selected from the current error text. This directs the agent toward the outstanding failure without prescribing a replacement patch. The agent then inspects, edits, and tests the current checkout using its existing harness and tools. ReGround changes the starting context; it does not generate or select the final patch. Preparation requires at most one historical replay and no verifier-model calls.

4 Experiments

We evaluate whether preparing repair history improves repair success and reduces model calls. We first compare ReGround with memory-free repair and direct historical reuse across five backbones, then test whether alternative presentations of the same history offer comparable gains.

4.1 Experimental Setup

We use two RepairMem tracks: Controlled-55, with 55 controlled hidden-drift tasks, and Natural-44, with 44 naturally sourced repair pairs. We use five backbones: GPT-5.6 Sol (OpenAI, 2026b), GPT-5.6 Luna (OpenAI, 2026a), Claude Opus 5 (Anthropic, 2026), Grok-4.7 (xAI, 2026), and a locally deployed Qwen3.8-27B (Qwen Team, 2026). Within each backbone, we compare three memory-use strategies. *No Memory* receives the current task without historical context; *Raw History* additionally receives the historical record unchanged; ReGround reassesses the recorded repair and supplies current execution feedback with relevant source excerpts. Comparisons fix the task, initial checkout, harness, and evaluation criteria; memory-bearing conditions use the same historical record.

Protocol and metrics. All agents run in the mini-SWE-agent harness with a budget of 24 model calls and a fresh sandbox per assignment (Jain et al., 2024; Pan et al., 2024). The frozen task-specific grader evaluates the final workspace when the agent submits or exhausts its budget, checking the designated public witness and protected-test integrity. We evaluate repair effectiveness and interaction efficiency. We measure repair success by acceptance under the designated executable tests, with protected tests left unchanged. Model calls quantify the interaction cost of attempting the repair, including both successful and unsuccessful runs (Bouzenia & Pradel, 2025).

4.2 Repair Performance Across Model Backbones

Table 1 shows that ReGround helps agents solve more repair tasks with fewer model calls than both No Memory and Raw History across all five backbones and both tracks. Figure 2 complements the track-level results with cumulative repair profiles and task-level outcomes across the combined 99 tasks. On Controlled-55, ReGround completes more repairs with 15.5–41.6% fewer calls than No Memory. Grok increases successful repairs from 44 to 52 of 55 while reducing calls from 1,133 to 679 (40.1%). Opus provides an even sharper contrast: despite being a strong backbone, its success drops from 67.3% with No Memory to 20.0% under Raw History, but rises to 89.1% with ReGround, which also uses 15.5% fewer calls than No Memory. Opus also runs closest to the 24-call budget, suggesting that strong model capability alone does not remove the cost of poorly prepared historical context under a constrained interaction budget.

On Natural-44, all five backbones gain 4.5–22.7 percentage points over No Memory using fewer calls. Grok and Qwen each solve ten additional repairs, increasing success from 34.1% to 56.8%. These sizable completion gains coexist with more modest interaction savings. For Grok, ReGround retains all 15 memory-free successes and solves ten additional tasks, while reducing total calls by 4.7%. The newly solved tasks require 20.9 calls on average under the 24-call budget, suggesting that grounding helps the agent complete difficult repairs without removing the substantial investigation and implementation they require. Lower absolute success on Natural therefore does

Table 1: Repair success and interaction cost across five backbones. Success is measured by final patch acceptance under the frozen task grader. Total model calls include both accepted and unsuccessful runs under the same 24-call budget. Bold indicates higher success or fewer calls.

	Controlled ($n = 55$)			**Natural-Provenance** ($n = 44$)		
Backbone	No Memory	Raw History	ReGround	No Memory	Raw History	ReGround
Repair success (%) ↑						
Qwen3.8-27B	83.6%	76.4%	**94.5%**	34.1%	31.8%	**56.8%**
GPT-5.6 Sol	85.5%	67.3%	**100.0%**	56.8%	52.3%	**61.4%**
GPT-5.6 Luna	72.7%	63.6%	**96.4%**	50.0%	43.2%	**61.4%**
Claude Opus 5	67.3%	20.0%	**89.1%**	27.3%	22.7%	**43.2%**
Grok-4.7	80.0%	72.7%	**94.5%**	34.1%	36.4%	**56.8%**
Total model calls ↓						
Qwen3.8-27B	932	1,087	**698**	992	996	**896**
GPT-5.6 Sol	844	918	**583**	780	732	**639**
GPT-5.6 Luna	805	887	**470**	704	630	**615**
Claude Opus 5	1,277	1,320	**1,079**	1,050	1,035	**993**
Grok-4.7	1,133	1,196	**679**	955	954	**910**

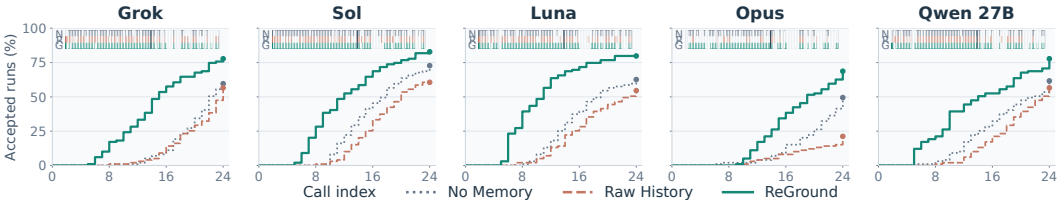

Figure 2: **Repair outcomes.** Each panel combines Controlled-55 and Natural-44. Curves show cumulative accepted runs by model-call index, normalized by all 99 tasks. The strips above the curves show individual task outcomes for each condition. Runs that exhaust the 24-call budget without submitting are plotted at call 24 if their final patches pass evaluation.

not imply smaller gains from grounding: for both Grok and Qwen, the improvement in success rate exceeds that on Controlled. Raw History, by contrast, reduces success in nine of the ten model–track comparisons. Thus, REGROUND's gains extend beyond recovering the aggregate loss from direct historical reuse. Section 5.3 examines the paired outcomes behind these gains.

4.3 COMPARISON WITH HISTORICAL-MEMORY BASELINES

We compare against four literature-inspired strategies for presenting and reusing historical repair experience: **Provenance Annotation** highlights the historical record's source while distinguishing provenance from current applicability, drawing on AuthMem-Bench and EYWA (Zhan et al., 2026; Joshi, 2026); **Temporal Warning** explicitly warns that later repository changes may supersede the record, inspired by memory-lifecycle management (Zhou et al., 2026); **Abstract Repair Trajectory** presents the earlier investigation and repair operation as a reusable procedural hint (Chen et al., 2025a; Xu et al., 2026); and **Episodic + Semantic Memory** separates the concrete past repair from its general repair insight, following ExpeRepair (Mu et al., 2026). These prompt-level adaptations evaluate alternative ways of presenting historical experience without executable replay, rather than reproducing the complete source systems.

Table 2 reports Qwen and GPT-5.6 Luna results on both tracks. REGROUND achieves higher success than every format for both models. On Controlled-55, Qwen completes 52 repairs with 698 calls, versus 39–45 repairs and 1,031–1,155 calls for the adapters. On Natural-44, it completes 25 repairs versus 10–13; the adapters use fewer calls but solve substantially fewer tasks. Luna shows the same overall pattern: on Controlled-55, REGROUND solves 53 tasks with 470 calls, versus 47–50 repairs and 1,278–1,320 calls for the adapters; on Natural-44, it completes 27 repairs with 615 calls, versus 25–26 repairs and 1,035–1,056 calls. Thus, neither restructuring history nor warning against

Table 2: **Alternative historical-context formats.** Within each track, the left pair of columns reports Qwen3.8-27B and the right pair reports GPT-5.6 Luna. Entries are accepted tasks / assigned tasks and total model calls. The four adapters present history without executable replay. Bold marks the best value for each model, track, and metric.

	Controlled (55 tasks)				**Natural** (44 tasks)			
Memory format	Success	Calls	Success	Calls	Success	Calls	Success	Calls
No Memory	46/55	932	40/55	805	15/44	992	22/44	704
Raw History	42/55	1,087	35/55	887	14/44	996	19/44	630
Provenance	43/55	1,095	48/55	1,320	10/44	840	25/44	1,056
Temporal warning	45/55	1,031	47/55	1,278	11/44	829	25/44	1,056
Abstract trajectory	41/55	1,155	48/55	1,320	12/44	839	26/44	1,035
Episodic + semantic	39/55	1,112	50/55	1,320	13/44	**809**	25/44	1,056
REGROUND	**52/55**	**698**	**53/55**	**470**	**25/44**	896	**27/44**	**615**

direct reuse matches REGROUND's repair success in these comparisons. Section 5.2 holds current evidence fixed to examine the additional effect of processing the historical recommendation.

5 WHAT MAKES REPAIR EXPERIENCE REUSABLE?

The benefit of repair memory is not determined by whether its stored patch still works. We examine how the record changes repair outcomes, how its presentation affects the use of fixed current evidence, and where interaction savings arise. Together, these analyses connect the benchmark's action-level evidence to downstream repair and delimit the conclusions supported by a failed replay.

5.1 REPAIR HISTORY OFFERS BOTH GAINS AND LOSSES

Historical context can help some repairs while hindering others within the same model and track. On Natural-44, Grok's increase from 15 repairs without memory to 16 with Raw History combines four history-only successes with three No-Memory-only successes. Conversely, Luna's decline from 22 to 19 repairs combines three gains with six losses. Every backbone exhibits both types of outcome on this track. The same pattern appears on Controlled-55: although Raw History lowers aggregate success for all five backbones, each still solves one to six tasks that No Memory misses.

In particular, the history-only successes on Controlled occur even though the stored patches do not resolve the current witness. The record can therefore help produce a new repair without providing a directly reusable edit. The accompanying losses show why the historical recommendation needs reassessment rather than unconditional reuse. These observations identify the opportunity for ReGround: it reconsiders the historical recommendation while using the record to support the current investigation. Section 5.2 tests this treatment with the accompanying current evidence held fixed.

5.2 REGROUNDING HISTORICAL ADVICE UNDER MATCHED EVIDENCE

Revising historical advice can improve repair even when the accompanying current evidence is held fixed. *History + Evidence* (RF) combines the unchanged historical record with the exact current evidence delivered by REGROUND. Both use the same history-conditioned evidence preparation, backbone, harness, and call budget. The contrast therefore tests assessed and rendered historical advice alongside fixed evidence, not a difference in the source evidence available to the agent.

As shown in Figure 3(c), the two backbones show complementary benefits on Controlled-55. For Grok, accepted repairs increase from 50 to 52 while total calls fall from 853 to 679, a 20.4% reduction. The net gain comprises four ReGround-only successes and two RF-only successes. For Luna, both conditions solve 53/55, but REGROUND uses 470 rather than 668 calls, a 29.6% reduction.

Figures 3(a) and 3(b) complement these aggregate results with Grok's per-call activity under RF and ReGround, showing that identical current evidence need not lead to identical interaction costs: the

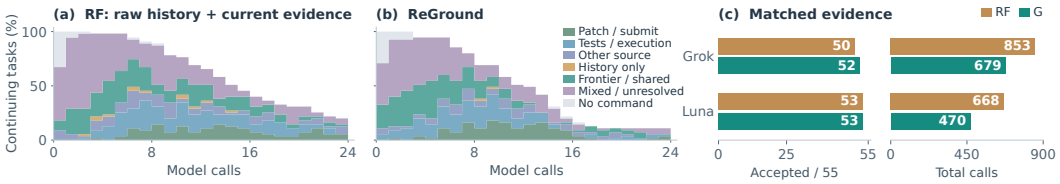

Figure 3: **Repair under matched current evidence.** (a,b) Grok's continuing runs on Controlled-55 at each model-call index, partitioned by command category; source categories are assigned based on the files explicitly referenced by each command. (c) Accepted repairs and total model calls for Grok and Luna under RF and ReGround. Both conditions receive the same current evidence.

Table 3: **Interaction savings on jointly successful tasks.** $n_\cap$ counts tasks accepted under both the comparator and ReGround. Calls report comparator $\rightarrow$ G on that subset. Share is the subset's call reduction divided by the net reduction over all tasks in the same model–track comparison.

Model	Track	N vs. G			R vs. G		
		$n_\cap$	Calls	Share	$n_\cap$	Calls	Share
Qwen	Controlled-55	46	721 $\rightarrow$ 508	91.0%	41	758 $\rightarrow$ 447	79.9%
	Natural-44	15	303 $\rightarrow$ 260	44.8%	13	281 $\rightarrow$ 202	79.0%
Grok	Controlled-55	42	854 $\rightarrow$ 509	76.0%	37	785 $\rightarrow$ 446	65.6%
	Natural-44	15	286 $\rightarrow$ 252	75.6%	16	309 $\rightarrow$ 267	95.5%
Luna	Controlled-55	40	583 $\rightarrow$ 342	71.9%	35	603 $\rightarrow$ 298	73.1%
	Natural-44	21	330 $\rightarrow$ 275	61.8%	17	231 $\rightarrow$ 198	220.0%
Sol	Controlled-55	47	722 $\rightarrow$ 511	80.8%	37	634 $\rightarrow$ 384	74.6%
	Natural-44	22	348 $\rightarrow$ 272	53.9%	20	303 $\rightarrow$ 245	62.4%
Opus	Controlled-55	35	797 $\rightarrow$ 629	84.8%	11	264 $\rightarrow$ 187	32.0%
	Natural-44	11	259 $\rightarrow$ 235	42.1%	9	204 $\rightarrow$ 198	14.3%

treatment of historical advice can improve completion or preserve it with substantially fewer calls. The stacked bands partition active runs by command category, rather than by final repair outcome. The paired analysis below locates the reduction within tasks that both conditions successfully repair.

5.3 Fewer Calls for the Same Successful Repairs

On Controlled-55, most of the net call reduction occurs on tasks solved under both conditions. Across Qwen, Grok, Luna, and Sol, these jointly successful tasks account for 71.9–91.0% of the reduction over No Memory and 65.6–79.9% over Raw History (Table 3). The same pattern holds for Grok under matched evidence: its 48 jointly successful tasks require 715 calls with RF and 590 with ReGround, accounting for 71.8% of the total 174-call saving. The reduction is thus concentrated in runs that remain successful under both conditions. On Natural-44, jointly successful tasks also use fewer calls for all four backbones against both N and R, although their share of the overall saving varies. That share can exceed 100% when additional work on other tasks offsets part of the reduction. For example, Luna saves 33 calls on tasks jointly solved with Raw History but only 15 calls overall, yielding the 220.0% share in Table 3. The savings thus extend to successful repairs, rather than arising solely from shorter unsuccessful runs.

The distinction is especially clear in the archived Qwen Matplotlib evaluation. All 17 tasks across four mechanism groups are solved by both No Memory and ReGround, yet calls fall from 270 to 140, a 48.1% reduction. In the remaining two groups, ReGround raises completion from 16/18 to 18/18 while reducing calls from 255 to 208. These results distinguish additional completions from reduced interaction on tasks already solved by the comparator.

5.4 A Failed Patch Does Not Exclude Its Location

A failed historical patch does not rule out a successful repair at the same code location. Each candidate in the same-locus study undergoes five execution checks. The historical witness must fail before the recorded action a_H and pass afterward. The current witness must fail both without a repair and with a_H, then pass with a different action a'_L at the same consumer location. Fifteen of

20 candidate constructions satisfy this complete pattern, spanning seven repositories, eight carriers, and 11 public witnesses. The checks retain the public function signatures and bodies and use the same parameterized test bodies; skip/xfail results and fixture or collection failures do not count as behavioral evidence, keeping the comparison focused on the effect of the competing edits.

In the Flask example, `SessionInterface.make_null_session` returns a session object wrapped in a singleton tuple in the historical state. The historical repair adds `[0]` to its call inside `RequestContext.push`, recovering the session object. In the current construction, the provider returns a doubly nested tuple. The historical edit still applies, but one unwrapping leaves a tuple rather than the required session object. Replacing it with `[0][0]` at the same expression restores `tests/test_async.py::test_async_route` across its five route parameters, while the public entry point, `Flask.wsgi_app`, remains unchanged. The required operation changes, but the location remains a valid repair site under the changed provider behavior.

6 Related Work

Repair experience and repository memory. Repository memory (Wang et al., 2026) uses commit history to support code localization. ExpeRepair (Mu et al., 2026) combines episodic repair demonstrations with semantic insights; SWE-Exp (Chen et al., 2025a) distills experience from repair trajectories, while MemRepair (Liu et al., 2026) organizes prior fixes and refinement experience for vulnerability repair. Concurrent work explores hierarchical trajectory abstraction (Xu et al., 2026) and adaptive experience selection (Luo et al., 2026). VibeMemBench (Fan et al., 2026) evaluates memory systems on repository tasks selected for a benefit from injected history in a reference setting. RepairMem separates the stored patch's effect from the record's contribution to a new repair.

Memory under changing conditions. Persistent-memory systems (Park et al., 2023; Packer et al., 2023; Chhikara et al., 2025) retain observations and experience across interactions. Experience-based agents also retain feedback or prior trajectories to guide later decisions (Shinn et al., 2023; Zhao et al., 2024). Recent work studies implicit conflict, stale observations, and conflicting memories (Chao et al., 2026; Sun & Zhang, 2026; Chen et al., 2026), as well as revocation and provenance (Zhou et al., 2026; Joshi, 2026). AuthMem-Bench (Zhan et al., 2026) distinguishes a remembered claim from the source authority required for its use. For code-assistant memory, Yadav (2026) studies stale facts derived from GitHub fixes, evaluating retrieval and answer accuracy rather than generated repairs. We examine a distinction: a record can accurately describe a past success even when its patch no longer resolves the current issue.

Execution-guided software repair. SWE-bench (Jimenez et al., 2024) provides executable repository tasks, while SWE-agent (Yang et al., 2024) and OpenHands (Wang et al., 2025) support tool-based repository interaction. Self-Debugging (Chen et al., 2024), Agentless (Xia et al., 2024), and SWE-Search (Antoniades et al., 2025) use execution or validation feedback to improve programs and candidate repairs for the current task. ReGround assesses a patch inherited from an earlier task and combines its current outcome with selected code excerpts to prepare the agent's context before the new repair begins, rather than using execution only after a new patch is proposed.

7 Conclusion

We studied the repair applicability gap: a historical repair can provide useful context without supplying an effective edit for the current issue. RepairMem evaluates this distinction through controlled and naturally sourced tasks, separating the stored patch's current effect from subsequent repair success and reuse cost. ReGround addresses the gap by reassessing the patch with a task-specific executable check, revising the historical advice, and supplying current code for further investigation. Across five backbones and both tracks, this preparation improves repair success over memory-free repair and direct historical reuse, while reducing model calls relative to memory-free repair. Paired analyses locate substantial savings in repairs completed by both conditions, while same-locus constructions show that an unsuccessful edit need not rule out its location. Together, these findings support retaining historical repairs as resources for investigation while using current evidence to reconsider the changes they recommend.

LARGE LANGUAGE MODEL USAGE STATEMENT

We acknowledge the use of large language models (LLMs) as auxiliary tools during the development of this work. LLMs were used to assist with research ideation, benchmark design, literature search and organization, selected implementation and figure-generation code, experimental analysis, and language polishing and manuscript revision. All AI-assisted outputs, including code, references, benchmark-design suggestions, analyses, and manuscript edits, were critically reviewed and verified by the authors before inclusion.

The research questions, benchmark specification, methodological design, experimental protocol, interpretation of results, and scientific conclusions were determined by the authors. All reported empirical results are derived from recorded executions of the evaluated systems, and the authors take full responsibility for the final text, claims, code, and experimental artifacts.

ETHICS STATEMENT

This work studies how coding agents reuse historical software-repair experience. It does not involve human-subject recruitment, user studies, or the collection of private personal data. The experiments use public open-source repositories, publicly available issue and repair histories, and existing software-engineering benchmarks. We use these artifacts to study program behavior and repair processes rather than to evaluate individual developers or infer personal attributes.

The proposed method prepares historical repair information for use by a coding agent and does not modify production systems outside the experimental sandbox. As with automated program-repair systems more broadly, generated patches may be incorrect, incomplete, or introduce regressions or security risks if deployed without validation. We therefore view execution-based testing, repository-specific validation, and human review as important safeguards for practical use. The benchmark and method are intended to study reliable reuse of software-engineering experience rather than autonomous deployment of unreviewed code changes.

REPRODUCIBILITY STATEMENT

The appendix documents how we construct RepairMem, including task composition, controlled and natural-provenance pairing procedures, input provenance, executable checks, scoring criteria, and the information supplied to each evaluation condition. It also specifies the ReGround execution and source-selection procedures, agent configurations, comparison conditions, prompt templates, and resource-accounting methodology.

REFERENCES

Anthropic. Introducing claude opus 5. `https://www.anthropic.com/news/claude-opus-5`, July 2026. Accessed: 2026-09-24.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *International Conference on Learning Representations*, volume 2025, pp. 64485–64515, 2025.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2846–2857. IEEE, 2025.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2188–2200. IEEE, 2025.

Hanxiang Chao, Yihan Bai, Rui Sheng, Tianle Li, and Yushi Sun. Stale: Can llm agents know when their memories are no longer valid? *arXiv preprint arXiv:2605.06527*, 2026.

Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, et al. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025a.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Yixiong Chen, Xinyi Bai, and Alan Yuille. The compliance trap: Diagnosing how ai agents consume conflicting memory. *arXiv preprint arXiv:2607.10608*, 2026.

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, 2025b.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Liyang Fan, Yingcheng Shi, Yongbin Li, Chenghao Sun, Xin Chen, Xander Xu, Hu Wei, Shiwen Ni, Min Yang, and Jieping Ye. Vibemembench: Evaluating memory systems for coding agents on real repository coding tasks. *arXiv preprint arXiv:2609.23570*, 2026.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024.

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.

Resham Joshi. Eywa: Provenance-grounded long-term memory for ai agents. *arXiv preprint arXiv:2605.30771*, 2026.

René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pp. 437–440, 2014.

Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, 2020.

Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pp. 213–224. IEEE, 2016.

Tobias Lindenbauer, Georg Groh, and Hinrich Schütze. From knowledge to noise: Ctim-rover and the pitfalls of episodic memory in software engineering agents. In *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)*, pp. 411–427, 2025.

Simiao Liu, Li Zhang, Fang Liu, Xiaoli Lian, Yang Liu, and Yinghao Zhu. Memrepair: Hierarchical memory for agentic repository-level vulnerability repair. *arXiv preprint arXiv:2605.17444*, 2026.

ZC Luo, JC Guo, WJ He, SY Wang, JC Yu, FM Zhao, Yuan Chen, Te Cao, LQ Liu, Ning Zheng, et al. Adarepair-mem: Adaptive experience orchestration for repository-level program repair. *arXiv preprint arXiv:2609.20130*, 2026.

Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, pp. 468–478. IEEE, 2019.

Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. Experepair: Dual-memory enhanced llm-based repository-level program repair. *Proceedings of the ACM on Software Engineering*, 3(FSE):3954–3976, 2026.

OpenAI. Gpt-5.6 luna. `https://developers.openai.com/api/docs/models/gpt-5.6-luna`, 2026a. Accessed: 2026-09-24.

OpenAI. Gpt-5.6 sol. `https://developers.openai.com/api/docs/models/gpt-5.6-sol`, 2026b. Accessed: 2026-09-24.

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.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Qwen Team. Qwen3.8-27b. `https://huggingface.co/Qwen/Qwen3.8-27B`, 2026. Accessed: 2026-09-24.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Yushi Sun and Yanjie Zhang. When memory lies: An empirical study of spatial memory staleness in vlm agents. *arXiv preprint arXiv:2608.04574*, 2026.

Boshi Wang, Weijian Xu, Yunsheng Li, Xuemei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In *International Conference on Learning Representations*, volume 2026, pp. 111266–111285, 2026.

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, 2025.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 1556–1560, 2020.

xAI. Introducing grok 4.7. `https://x.ai/news/grok-4-7`, September 2026. Accessed: 2026-09-24.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Yisen Xu, Jiayuan Zhou, Ruiqi Pan, and Tse-Hsun Chen. Reusing past repairs through hierarchical trajectory abstraction for coding agents. *arXiv preprint arXiv:2607.29658*, 2026.

N. Yadav. Temporal validity on real software histories: Eliminating stale-fact errors in code-assistant memory over github fixes. *arXiv preprint arXiv:2608.20685*, 2026.

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.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Qiuyang Zhan, Rui Zhang, Sheng Guo, Lepeng Zhao, and Zhuotao Liu. When memory becomes authority: Benchmarking authority collapse at the memory consolidation boundary. *arXiv preprint arXiv:2608.01679*, 2026.

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.

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.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Yan Zhou, Yue Ouyang, Kaiyang Zheng, and Suncheng Xiang. Tepa: Revoking stale memories for conflict-robust language agents. *arXiv preprint arXiv:2608.07429*, 2026.

A REPAIRMEM CONSTRUCTION AND EVALUATION

RepairMem evaluates whether a historical repair remains useful when its stored edit no longer solves the current task. This section describes how executable seeds establish the relationship between historical and current repairs, how fixed records specify the experience available to the agent, and how workspace scoring evaluates the resulting repair. We detail the 99-task composition, the two construction tracks, input provenance, and acceptance criteria. Throughout, $r[a]$ denotes the application of patch a to checkout r, and w is the task's disclosed public witness.

A.1 MOTIVATION

Table 4 contrasts the evaluation targets of memory and software-repair studies. Memory evaluations examine cross-task reuse and changes in validity; repair evaluations emphasize executable outcomes and the work needed to solve an issue. Repair-memory reuse connects these concerns: an evaluation must establish what a retrieved patch does in the current repository and whether its record helps the agent produce a new repair when that patch is insufficient.

RepairMem makes this distinction explicit by fixing the historical record and patch across memory-use strategies and measuring the old action separately from the agent's final repair. Controlled historical/current comparisons and naturally sourced repair pairs provide complementary settings for relating action applicability to repair success and reuse cost.

Table 4: Evaluation coverage in representative memory and repair studies. ✓ denotes explicit coverage, ∼ partial coverage, and × not evaluated. Concurrent work is marked by †.

Study	Cross-task memory	Validity shift	Fixed-history control	Action applicability	Record–action separation	Executable outcomes	Reuse cost
STALE (Chao et al., 2026)	✓	✓	×	×	×	×	✓
SpatialSTALE (Sun & Zhang, 2026)	✓	✓	×	×	×	✓	✓
Temporal Validity† (Yadav, 2026)	✓	✓	×	×	×	×	✓
SWE-bench (Jimenez et al., 2024)	×	×	×	×	×	✓	×
RepoMem (Wang et al., 2026)	✓	×	×	×	×	✓	✓
ExpeRepair (Mu et al., 2026)	✓	×	∼	×	✓	✓	✓
SWE-Exp (Chen et al., 2025a)	✓	×	×	×	✓	✓	✓
MemRepair (Liu et al., 2026)	✓	×	∼	×	∼	✓	✓
STAIR† (Xu et al., 2026)	✓	×	∼	×	∼	✓	✓
AdaRepair-Mem† (Luo et al., 2026)	✓	×	×	×	∼	✓	✓
VibeMemBench† (Fan et al., 2026)	✓	×	✓	∼	×	✓	✓
RepairMem (ours)	✓	✓	✓	✓	✓	✓	✓

A.2 EXECUTABLE SEED SCHEMA

Construction begins with a concrete code relationship or a pair of developer repairs. Each seed binds that relationship to a repository state and public check (Table 5). The construction then instantiates the historical and current tasks, validates their executable properties, and renders the historical experience into a fixed record.

A.3 TASK COMPOSITION AND EXPERIMENTAL UNITS

The benchmark contains 99 tasks: 55 Controlled Hidden-Drift tasks and 44 Natural-Provenance pairs. The controlled track spans four carrier checkouts and 18 instantiated repair relations; the natural track spans ten repositories and 42 historical repair relationships. Table 6 summarizes the two tracks, and Table 7 lists the natural track's repositories.

Experimental units. A *carrier* is a repository checkout together with its execution environment. In the controlled track, an *instantiated repair relation* groups variants of a construction family on a concrete consumer–provider edge. Each task specifies a mutation and witness within that group; multiple variants can share the same carrier and repair relation. Natural relations link earlier developer repairs to current issues. Success rates are task-weighted.

Table 5: Executable seed schema for the two construction tracks. Fields identify the repository artifacts and repair relationship used to construct each historical/current pair.

Seed field	Controlled Hidden-Drift	Natural-Provenance
Repository state	Carrier checkout and execution environment	Recorded base checkout for each real issue.
Public check	Existing test node and its execution command	Disclosed current failure-to-pass tests and test patch.
Code anchor	Public entry and an executed provider–consumer edge	Historical repair function, path, and surviving signature.
Historical experience	Provider transformation and compensating consumer patch	Earlier issue, timestamp, and developer production patch.
Current task	Separate internal mutation beneath the preserved public surface	Later issue with a source/test-supported relation to the earlier repair.
Pairing evidence	Shared carrier, public entry, witness, and concrete code edge	Same-function overlap or a supported cross-function relation.

Table 6: Tasks and repair relations in RepairMem. Multiple tasks can share a repair relation.

Track	Tasks	Relations	Historical support and task relation
Controlled Hidden-Drift	55	18	Executed historical repair with a shared public entry and witness across constructed states.
Natural-Provenance	44	42	Earlier developer repairs paired with later issues through same-function or supported source/test relations.

The controlled track comprises four carrier checkouts and eight public-witness strings across pytest, xarray, and Matplotlib. The ten natural-track repositories cover the project areas listed in Table 7.

Table 7: Repositories and project areas represented in the Natural-Provenance track.

Repository	Project area	Repository	Project area
Astropy	Astronomy/data	Django	Web framework
Matplotlib	Visualization	Pylint	Static analysis
pytest	Testing	scikit-learn	Machine learning
Seaborn	Statistical visualization	Sphinx	Documentation
SymPy	Symbolic mathematics	xarray	Scientific arrays

A.4 Controlled Historical and Current States

The representation-bridge generator modifies provider return values and the executed consumer calls that use them. It implements two transformations:

Family	Historical provider change	Historical consumer repair
Singleton tuple	`return v` → `return (v,)`	`f(...)` → `(f(...))[0]`
Value mapping	`return v` → `return {"value": v}`	`f(...)` → `(f(...))["value"]`

Here, `v` and `f(...)` denote the original return expression and consumer call. The generator requires supported syntax at both sites and produces exact source edits. The two transformation families are instantiated on concrete consumer–provider edges to form the repair relations.

State checks. Let r_* be the passing carrier, b the provider change, a_H the compensating historical edit, and c the separate current mutation. The construction checks the following states:

$$\begin{aligned} w(r_*) = \text{PASS}, \quad w(r_*[b]) = \text{FAIL}, \quad w(r_*[a_H]) = \text{FAIL}, \\ w(r_*[b][a_H]) = \text{PASS}, \quad w(r_*[c]) = \text{FAIL}, \quad w(r_*[c][a_H]) = \text{FAIL}. \end{aligned} \tag{1}$$

A current reference repair is checked separately on $r_*[c]$. The two branches thus establish historical success and test the same repair under a different internal fault. The public entry retains its name,

signature, and body across these constructed branches; they are not a sequence of Git commits. Here, FAIL denotes the recorded command outcome. Its behavioral interpretation depends on the execution stage and coverage, as specified in Appendix A.7.

For example, a pytest seed uses the public entry `PyobjMixin.reportinfo` and a consumer in `getfslineno`. The historical repair reads the `value` entry from the result of `_pytest._code.getfslineno(obj)`, compensating for the mapping-valued provider. The current branch instead changes a guard in plugin-specification handling. The old edit has a demonstrated historical justification but does not resolve this initialization failure. Appendix B.3 shows the corresponding execution observations and delivered source evidence.

Restored-state check. The primary collection additionally removes the current fault and applies the stored edit to the restored passing checkout. Applying the historical edit causes the restored check to fail. Both construction repetitions are available for all 20 primary tasks.

Task selection groups repair relations by carrier, witness, public surface, and concrete consumer–provider edge. Primary task selection and historical-record rendering precede the evaluated agent runs. Current reference edits validate construction but are not supplied as target patches to the agent.

A.5 NATURAL REPAIR PAIRING AND VALIDATION

Natural seeds pair earlier developer repairs with later SWE-bench issues (Jimenez et al., 2024). The same-function procedure indexes production-patch hunks by repository, path, qualified function name, and signature. It retains functions spanning at most 240 lines and requires a shared non-generic test call or changed-code identifier. For each current issue/function, it selects the most recent qualifying earlier repair. The broader procedure allows distinct functions connected by supported source relations, including bounded calls and configured override families, together with shared test/code evidence. Public-surface resolution and removal of import-only links and duplicates precede executable validation.

Pair validation. For each pair, we install the disclosed current test patch, reproduce the failure, and confirm that the current developer patch passes. A reverse-application check identifies an old edit already incorporated into the checkout; patches that apply as new changes are tested on the current witness.

The same-function collection includes all 19 validated records in its primary eligibility class and the first preselected validated record in a secondary class. The broader collection includes all 24 validated pairs in its roster. These two collections form the fixed Natural-44 task set.

Pairing examples. Django issues 11477 and 12184 both concern `RegexPattern.match`. The earlier edit filters absent optional groups from keyword arguments; the later task concerns the choice between positional and named captures. The earlier patch is already present in the current checkout. In the broader Astropy pair, issues 7746 and 7858 address empty-array and scalar-coordinate behavior, respectively, through WCS conversion. Both examples provide a reason to consult the historical record without identifying its edit as the current solution.

A.6 INFORMATION AVAILABLE TO THE AGENT

Each task supplies a current issue and checkout, a disclosed witness, a historical record, and its exact patch. Construction additionally records the current reference repair and pairing metadata. Online REGROUND preparation uses the task inputs and current execution evidence, not the current reference patch. Table 8 identifies the origin and use of each field.

Field provenance. Natural-Provenance evaluates reuse of supplied historical repair pairs. The historical issue and patch originate from the earlier repair. All matched memory conditions use the same supplied record.

Memory templates. The following templates specify how constructed or retrieved experience is delivered to the repair agent. They are agent-input templates, not prompts for generating benchmark tasks; construction uses programmatic rules. Angle-bracketed fields are replaced with instance-specific values, and fixed wording follows the implemented historical-record renderers.

Table 8: Origins and uses of the task inputs.

Field	Origin	Use in evaluation
Current task and witness	Current issue and disclosed repository tests	Shared task input in all conditions.
Historical record and patch	Executed controlled repair or earlier developer repair	Supplied unchanged to Raw History; processed by REGROUND.
Pair relationship	Benchmark-authored source/test analysis, as described above	Explains the supplied record's relation to the current task.
Action assessment	Current patch checks and replay observation	Revises the historical recommendation.
Source frontier	Current source and failure evidence; trace-based selection also uses history/replay signals	Bounded code excerpts for investigation.
Current reference repair	Construction validation	Withheld from REGROUND; used only in the explicitly repair-informed reference condition.

Controlled historical-record template

```
Historical repair memory (advisory evidence):
- Public surface: <path>::<qualified public-entry name>
- Stored repair action: <historical operation description>
- Historical repair locus: <path>::<qualified repair-function name>
- Signature: <historical repair-function signature>
- Action advice: begin with this locus and stored action before
    broadening the investigation.
```

The tuple operation is rendered as "treat the provider result as a one-element tuple and consume element 0"; the mapping operation is rendered as "treat the provider result as a mapping and consume its 'value' entry." The historical function signature is capped at 240 characters. The exact historical patch is associated with the task for replay; this controlled record describes its operation rather than printing the complete diff.

Natural historical-record template

```
Repair-record role: HISTORICAL_REAL_REPAIR
Provenance instance: <historical issue identifier>
Provenance timestamp: <historical issue timestamp>
Issue record:
<historical issue text>
Shared behavioral responsibility:
<benchmark-authored relationship annotation>
Repair locus: <historical repair locus>
Stable signature: <historical function signature>
Successful production repair:
<historical production diff>
Treat this real repair record as advisory evidence and verify it
    against the current checkout.
```

Delivery styles. The natural template includes the production diff and provenance, whereas the controlled template provides a compact action description and location. The instructions also differ: controlled advice prioritizes reuse, while natural advice requests verification. These forms are fixed within each task across matched memory-use conditions.

A.7 EXECUTABLE SCORING

The scorer evaluates the final workspace when the agent submits or reaches its interaction limit. Acceptance depends on the disclosed public check and protected-test integrity, not a match to a

reference diff or an edit at a predetermined location. Table 9 specifies how the controlled and natural graders recognize a passing execution.

Table 9: Final-workspace acceptance criteria. Evaluation uses the disclosed tests and protected-test checks.

Property	Controlled	Natural-Provenance
Public command	Fixed witness node in the carrier environment	Disclosed current failure-to-pass tests, installed with the test patch.
Acceptance	Command exits zero and the protected test path is unchanged	Positive executed-pass evidence and all disclosed test-patch paths unchanged.
Execution evidence	Exit-status predicate	Positive pytest pass count, or positive Django run count with an OK summary; all-skipped runs are rejected.
Reference matching	No reference-diff or required-location match	No reference-diff or required-location match.

Execution outcomes. Controlled replay maps ordinary nonzero command exits to FAIL, including initialization, import, and collection errors. The label records the outcome of the replay attempt; it does not establish that execution reached the modified expression. An unavailable natural replay remains a separate evidence state. These scorers evaluate cooperating repair agents and do not independently detect every production-code change that could bypass the intended behavior.

B REGROUND IMPLEMENTATION AND PROMPT INTERFACES

B.1 EXECUTION AND CONTEXT PREPARATION

Preparation takes the current checkout r, witness w, historical record m, exact patch a_H, and evidence configuration. It returns revised context $\widetilde{m}$, source frontier F, and preparation measurements. This stage uses neither a verifier language model nor a newly generated candidate patch.

Replay execution. The trace-based configuration uses current and historical-replay execution records and traces collected before agent execution. In its primary collection, the historical patches apply but fail the current check. Native preparation runs the current witness in the agent's sandbox, then checks and applies the exact historical diff when possible. After successful application, it runs the same witness once. Exit zero supplies a passing observation; an ordinary nonzero exit supplies a failing one. Failed application or an unavailable command yields no usable replay result. A passing current command does not bypass native replay.

Workspace restoration. Native execution saves the affected files and tracked diff before replay. It restores this state by reverse application, falling back to the saved files, and verifies that the tracked diff is restored before agent editing. Restoration failure terminates the assignment. The check covers tracked state, not arbitrary untracked test products. Temporary replay files remained readable in the Matplotlib evaluation, but the recorded final trajectories did not access them.

Already-present repairs. For natural records, reverse-application checking identifies historical changes already incorporated into the checkout, avoiding replay as a new edit. Table 10 separates the evidence backends and their rendering rules from the patch-state handling used for natural records.

These branches change the initial context without changing the harness's edit permissions. An unsuccessful historical patch therefore does not prevent the agent from proposing a different edit at the same location.

B.2 SELECTING CURRENT SOURCE EVIDENCE

Trace-based selection. The trace-based selector parses source in the current checkout, excluding test directories, `test_` files, and `conftest.py`. It first admits at most four distinct production scopes enclosing extracted failure frames. Runtime nodes are matched by file, first line, and qualified name. Ambiguous matches fall back to an enclosing syntax-tree scope, which may be a class.

Table 10: Evidence backends and historical-context rendering in ReGround. Both backends assess the stored action and provide bounded current-code evidence before agent editing. Failed replay withdraws the recommendation to reuse the edit; the configurations differ in whether its text is retained.

Evidence backend	Execution and source evidence	Failed-replay rendering	Other handling
Trace-based	Precollected current/replay command records and traces	Retains the old action and location as negatively assessed historical context	An empty usable frontier yields empty memory context.
Native	Current command output, native frames, at most one replay, and local source inspection	Omits the old action/location text and delivers current evidence	Supported or unavailable replay retains the record with a receipt.

Natural-record patch states. Presence checking identifies incorporated historical edits, which are not presented as new repairs. These are patch-state handling rules, not a third evidence backend.

Additional candidates are restricted to currently executed scopes. Candidate files comprise failure files, files executed only in the current run, files with new line hits, the historical file, and the two strongest file-level lexical matches scoring at least three. Lexical matching uses the last 6,000 characters of terminal output and a fixed stop list. A shared token contributes three points if it contains an underscore and one point otherwise.

Candidate ranking. Candidates are ranked lexicographically by current-error overlap, current-only execution, file-level line-hit difference, historical-owner affinity, call depth, and non-module status. Path, qualified name, and scope kind break ties. A terminal-error match scoring at least three is admitted before the remaining slots are filled. Once three slots are occupied, candidates without current-only, line-difference, or owner support are skipped.

Excerpt budget. Failure-frame excerpts include five source lines on either side of the evidence line; other excerpts contain the selected scope. Each excerpt is truncated to 3,000 characters. The full context contains at most five scopes and 14,000 characters, including headers. The current reference repair location is used only afterward to assess source coverage.

Native selection. Native extraction starts from repository-local failure frames and their enclosing syntax-tree scopes. It examines supported subscript and call expressions to resolve one-hop provider relationships through keyed registries and module-level containers. A registry match requires a literal key in the error text and receives priority zero. Module-container resolution receives priority one and selects the first implemented method among `__setitem__`, `__getitem__`, `update`, and `get`, in that order. Enclosing failure frames receive priority ten plus their frame order. Path, start line, and name break ties.

Candidates are deduplicated and their source is extracted from the current checkout. The Matplotlib configuration delivers at most two scopes, with limits of 60 lines and 1,800 source bytes per scope and 3,000 source bytes in total. Native selection collects no dynamic line/call graph.

B.3 Worked Example: From Historical Advice to Current Evidence

The excerpts below show the context prepared for pytest task `37f1649f` on carrier `pytest-5221`. The historical patch modifies `getfslineno` in `compat.py`, beneath the unchanged public entry `PyobjMixin.reportinfo`. The public check uses the following file and test node:

```
File: testing/test_collection.py
Node: Test_genitems::test_class_and_functions_discovery_using_glob
```

The historical input uses the mapping-consumption operation in the controlled template (Appendix A.6). The excerpts below preserve the recorded output and show its first delivered source region. The stored patch changes the consumer expression:

```
- fslineno = _pytest._code.getfslineno(obj)
+ fslineno = (_pytest._code.getfslineno(obj))["value"]
```

The historical provider mutation wraps the returned `(fspath, lineno)` pair in a mapping. The current mutation instead reverses a guard in plugin-specification handling. Both the current command and historical replay exit with code 4 and the same message about invalid plugin specifications, taking 4.910841 and 4.503927 seconds, respectively. The failure occurs during initialization, before the selected test body runs.

Selected evidence. The first delivered region is `_get_plugin_specs_as_list` in `src/_pytest/config/__init__.py`. It is selected by terminal-error text matching with lexical score nine, call depth eight, zero current-minus-replay line count, and no historical-owner affinity. The next regions, `Metafunc` and `_prepareconfig`, have lexical scores seven and four; both also have zero replay difference and no owner signal. Selection is therefore anchored in the current error, even though the two command outcomes do not distinguish the historical patch's internal behavior.

First source region: decisive guard excerpt

```
597 | if not (not isinstance(specs, (list, tuple))):
598 |     raise UsageError(
599 |         "Plugin specs must be a ','-separated string or a "
600 |         "list/tuple of strings for plugin names. Given: %r" %
    specs
601 |     )
```

The agent subsequently inspects the supplied parser, edits its guard, and passes the public check.

C Experimental Protocol and Baseline Templates

C.1 Backbones, Harness, and Comparison Conditions

We evaluate Qwen3.8-27B, GPT-5.6 Sol, GPT-5.6 Luna, Claude Opus 5, and Grok-4.7. Qwen uses the local `RedHatAI/Qwen3.8-27B` configuration; the other backbones use the recorded API endpoints. All agents run in mini-SWE-agent with a limit of 24 model calls per assignment. The primary Qwen runs use mini-SWE-agent 2.4.6, greedy decoding, and task seed 4201001. API requests use temperature zero and a 4,096-token response ceiling.

Comparison conditions. No Memory receives the current task and repository without history. Raw History additionally receives the historical record unchanged. REGROUND replaces that record with the prepared context. Within each backbone, the conditions share the task, initial checkout, public witness, harness, and call budget. The main comparison contains $99 \times 5 \times 3 = 1{,}485$ assignments. Diagnostic comparisons and the repair-informed reference are reported separately; infrastructure-invalid attempts are not counted as additional tasks or terminal repair outcomes.

Repair-informed reference. The Qwen study includes a repair-informed reference that supplies current repair intent and location instead of historical advice. Natural configurations may also include the current production diff. This condition uses the same task and agent interface but has privileged current-repair information, unlike the historical-memory conditions.

C.2 Historical-Context Formats

The four literature-inspired strategies in Section 4.3 organize and condense historical repair information in different ways. Provenance Annotation emphasizes the record's source, and Temporal Warning notes that later changes may supersede it. Abstract Repair Trajectory presents the earlier investigation and operation as a procedural hint. Episodic + Semantic Memory separates the past episode from its general repair insight. These are prompt-level adaptations of ideas from the cited memory literature, not reproductions of the complete systems.

Controlled adapters use the historical public surface, action description, location, and signature. Natural adapters provide compact records of provenance, the pairing relationship, repair location, and affected paths, rather than the full historical issue and production diff. None performs executable replay during context preparation.

C.3 Adapter Templates and Shared Task Wrapper

Each adapter replaces only the memory block in a shared task wrapper. The boxes below reproduce the fixed renderer templates; braced names identify task-specific fields.

Shared task wrapper

```
Task:
{task_text}

Memory context:
{adapter_memory}

Controlled suffix:
Work directly in /testbed. Inspect the repository, make one coherent
production-code repair, run relevant tests, and submit when done.
Do not modify tests.

Natural suffix:
Repair production code only. The installed public witness is available
    ;
do not modify tests.
```

Provenance Annotation adapter

```
Controlled renderer:
Provenance-backed historical memory (advisory):
- Public surface: {public_surface}
- Stored repair action: {stored_action}
- Historical repair locus: {historical_locus}
- Signature: {signature}
- Provenance is not proof that the old locus is still authoritative;
  inspect the current checkout before editing.

Natural renderer:
Provenance-backed historical memory (advisory):
- Earlier repair provenance is trusted but does not establish current
    authority.
Repair-record role: HISTORICAL_REAL_REPAIR
Provenance instance: {provenance_instance}
Shared behavioral responsibility: {responsibility}
Repair locus: {historical_locus}
Stable signature: recorded in the historical repair record.
Historical production action touched: {historical_action_paths}
Treat this real repair record as advisory evidence and verify it
    against
the current checkout.
```

Temporal Warning adapter

```
Controlled renderer:
Temporally annotated historical memory (advisory):
- This record predates the current issue and checkout; later code may
    supersede it.
- Public surface: {public_surface}
- Stored repair action: {stored_action}
- Historical repair locus: {historical_locus}
- Signature: {signature}
- Compare this claim with current repository evidence and revise it if
     the
```

```
  current state conflicts.

Natural renderer:
Temporally annotated historical memory (advisory):
- This record predates the current issue and checkout; later evidence
    may supersede it.
Repair-record role: HISTORICAL_REAL_REPAIR
Provenance instance: {provenance_instance}
Shared behavioral responsibility: {responsibility}
Repair locus: {historical_locus}
Stable signature: recorded in the historical repair record.
Historical production action touched: {historical_action_paths}
Treat this real repair record as advisory evidence and verify it
    against
the current checkout.
```

Abstract Repair Trajectory adapter

```
Controlled renderer:
Reusable repair trajectory from an earlier case (abstracted; no patch)
    :
- Earlier public surface: {public_surface}
- Earlier investigation first inspected: {historical_locus}
- Earlier repair operation: {stored_action}
- Earlier target signature: {signature}
- Suggested sequence: inspect the analogous provider/consumer, apply
    the
  operation only if current code supports it, then run the disclosed
    reproducer.
- This is a reusable trajectory hint, not an instruction to copy the
    old location.

Natural renderer:
Reusable repair trajectory from an earlier case (abstracted; no patch)
    :
- Reuse the earlier investigation only where current code supports it;
  do not copy a location blindly.
Repair-record role: HISTORICAL_REAL_REPAIR
Provenance instance: {provenance_instance}
Shared behavioral responsibility: {responsibility}
Repair locus: {historical_locus}
Stable signature: recorded in the historical repair record.
Historical production action touched: {historical_action_paths}
Treat this real repair record as advisory evidence and verify it
    against
the current checkout.
```

Episodic + Semantic Memory adapter

```
Controlled renderer:
Dual-format prior experience (episodic + semantic; advisory):
Episodic record:
- Earlier repair at {historical_locus} for public surface {
    public_surface}.
- Observed target signature: {signature}.
Semantic record:
- The earlier repair used this operation: {stored_action}.
- Preserve the public behavior, but re-check the responsible
    implementation
  in the current checkout.
```

```
Natural renderer:
Dual-format prior experience (episodic + semantic; advisory):
- Re-check both the earlier location and the behavioral operation
    against
  current code.
Repair-record role: HISTORICAL_REAL_REPAIR
Provenance instance: {provenance_instance}
Shared behavioral responsibility: {responsibility}
Repair locus: {historical_locus}
Stable signature: recorded in the historical repair record.
Historical production action touched: {historical_action_paths}
Treat this real repair record as advisory evidence and verify it
    against
the current checkout.
```

D Repair Reuse, Recovery, and Context Accumulation

We examine which history-associated successes are retained and which losses are recovered on Controlled-55 and Natural-44, test the stability of repair gains across decoding seeds, and analyze context accumulation under matched evidence. The retention and context analyses use the recorded main runs; the stability study adds three repetitions on the original 20-task cohort within Controlled-55.

D.1 Preserving Historical Benefits while Recovering Losses

For each backbone and track, the observed No Memory (N) and Raw History (R) outcomes partition tasks into four disjoint groups: both succeed, only R succeeds, only N succeeds, and neither succeeds. We then measure ReGround (G) acceptance within each group. This separates the retention of history-only successes from the recovery of history-associated losses and the completion of tasks missed by both controls.

Table 11: ReGround acceptance within groups defined by the observed N/R outcomes. Each cell reports G-accepted tasks / tasks in the group. R-only captures retention of history-only successes; N-only captures recovery of history-associated losses. The four denominators sum to 55 or 44 in each row.

Backbone	Track	Both N/R	R only	N only	Neither N/R
Grok	Controlled-55	32/34	5/6	10/10	5/5
Sol	Controlled-55	34/34	3/3	13/13	5/5
Luna	Controlled-55	31/31	4/4	9/9	9/11
Opus	Controlled-55	10/10	1/1	25/27	13/17
Qwen	Controlled-55	40/40	1/2	6/6	5/7
Grok	Natural-44	12/12	4/4	3/3	6/25
Sol	Natural-44	19/20	1/3	3/5	4/16
Luna	Natural-44	16/16	1/3	5/6	5/19
Opus	Natural-44	8/9	1/1	3/3	7/31
Qwen	Natural-44	11/11	2/3	4/4	8/26

Recovery and retention. Across the five backbones on Controlled-55, ReGround recovers 63 of the 65 model–task outcomes that succeed without memory but fail with Raw History. It retains 14 of the 16 history-only successes and solves 37 of the 45 cases missed by both controls. The aggregate gains thus combine recovery of losses, retention of historical benefits, and additional completions on the same evaluated task sets.

Natural transfer. The same three types of gain occur on Natural-44. Grok preserves all four history-only successes, recovers all three N-only successes, and solves six tasks missed by both controls. It also retains all 12 shared N/R successes, accounting for its 25 repairs as $12 + 4 + 3 + 6$. Every backbone solves tasks missed by both natural-track controls, with four to eight such completions per backbone.

D.2 REPAIR GAINS PERSIST ACROSS DECODING SEEDS

We revisit the complete original 20-task cohort within Controlled-55, covering pytest, xarray, and Matplotlib. Holding the tasks and memory conditions fixed, we repeat Qwen3.8-27B across three decoding seeds at temperature 0.2. Table 12 reports these repetitions alongside the original temperature-zero evaluation.

Table 12: **Repair success and interaction cost across decoding seeds.** Qwen3.8-27B results on the original 20-task cohort of Controlled-55. Calls and cumulative model-request time include all runs; timing excludes repository execution and preparation. Bold marks the best values in each block.

Condition	Accepted	Success (%)	Model calls	Model-service time (s)
Temperature 0*, seed 4201001 (original evaluation)*				
No Memory	13/20	65.0	407	1,738.3
Raw History	12/20	60.0	416	1,692.1
REGROUND	**17/20**	**85.0**	**350**	**1,638.6**
Temperature 0.2*, seed 4201001*				
No Memory	13/20	65.0	389	1,660.9
Raw History	12/20	60.0	395	1,606.3
REGROUND	**17/20**	**85.0**	**300**	**1,403.6**
Temperature 0.2*, seed 4201002*				
No Memory	14/20	70.0	362	1,548.1
Raw History	11/20	55.0	415	1,739.6
REGROUND	**17/20**	**85.0**	**280**	**1,328.2**
Temperature 0.2*, seed 4201003*				
No Memory	13/20	65.0	391	1,800.3
Raw History	13/20	65.0	401	1,626.4
REGROUND	**16/20**	**80.0**	**300**	**1,420.8**

REGROUND achieves the highest success rate and the fewest model calls in every evaluation. Across the three temperature-0.2 seeds, it succeeds on 50/60 runs (83.3%), compared with 40/60 (66.7%) for No Memory and 36/60 (60.0%) for Raw History. Relative to No Memory and Raw History, respectively, REGROUND reduces the total number of model calls by 22.9% and 27.3%, and cumulative model-service time by 17.1% and 16.5%. Task outcomes differ across these seeds on only 2/20 tasks under REGROUND, compared with 1/20 for No Memory and 5/20 for Raw History. These repetitions show that the repair and efficiency gains persist across decoding seeds, with consistent outcomes on most tasks.

D.3 MATCHED EVIDENCE AND CONTEXT ACCUMULATION

History + Evidence (RF) and G receive identical history-conditioned current source evidence. RF retains the raw historical recommendation, whereas G revises it through action assessment and rendering. We extend the main matched-evidence comparison by examining input and output token volumes on all 55 controlled Grok tasks.

Table 13: Token workload under matched current source evidence for Grok on Controlled-55. Input totals include context repeated across requests; output totals count generated tokens.

Condition	Input tokens (M)	Output tokens (K)	Accepted patches	Empty diffs
History + Evidence	12.808	390.8	50	5
ReGround	7.664	398.1	52	3

G uses 40.2% fewer input tokens while producing two more accepted patches. Output volume remains similar, increasing by 1.9%. The workload reduction is therefore in accumulated input context, not generated output. This complements the call-count comparison: even with the same supplied source evidence, the two historical-context treatments lead to different interaction histories and cumulative input workloads.

E Repair Trajectories and Action Scope

E.1 The Same Patch through Different Investigation Routes

Figures 4 and 5 show two Grok runs on the same Controlled-55 task, with identical supplied current evidence and identical accepted final patches. History + Evidence (RF) uses 24 model calls and 24 executed commands; ReGround uses ten calls and eight commands. The first command containing an edit occurs at index ten under RF and four under G.

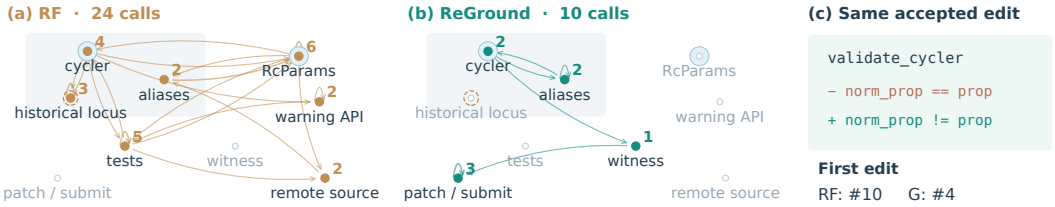

Figure 4: **Same accepted patch, different investigation routes.** This Controlled-55 pair has the largest RF/G call reduction among the Matplotlib construction pairs. Arrows connect consecutive primary command targets; numbers count visits. Blue halos mark delivered source scopes, and the dashed ring marks the historical location. The layout shows an investigation route, not a dependency graph. Both runs make the same comparison-operator edit in `validate_cycler`.

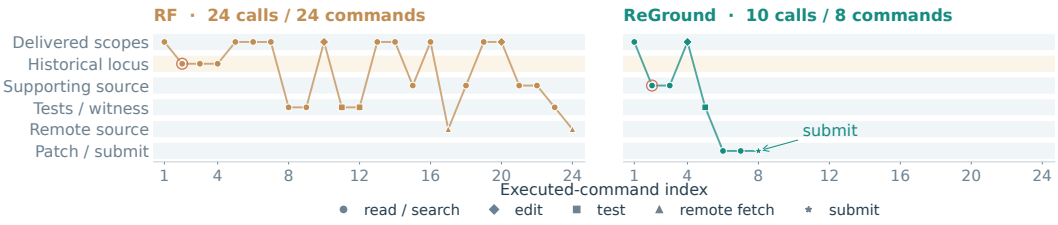

Figure 5: **Command sequences for the same repair pair.** Each point is an executed command, assigned a manually coded primary target; compound commands may also touch other targets. Shapes distinguish edits, tests, remote fetches, and submission, while red rings mark nonzero shell exits. G tests and submits its patch; RF continues investigating until its budget ends, retaining an accepted patch. The shared horizontal scale counts executed commands rather than model calls.

E.2 Observed Recovery and Unresolved Investigations

Table 14 aligns recorded events using one-based command indices. The pytest recovery follows the prepared context shown in Appendix B.3. The paired xarray runs contrast a Raw History success with a ReGround run that reaches the relevant code but does not submit a patch.

Table 14: Recorded inspection, editing, and testing events in selected primary trajectories. Indices refer to executed commands; no discovery time is imputed when an edit is absent.

Run	Investigation	Edit	Test / termination
pytest recovery (G)	Parser read at 4	Guard changed at 6	Public pass at 7; submit at 12.
xarray loss (G)	Alignment reads at 13–15	No submitted patch	Call limit at 24.
xarray paired (R)	Alignment reads at 8–9	Append restored at 12	Eight passes at 13; submit at 17.
pytest collection (G)	Collection probes at 4–9	No submitted patch	No items at 21–22; call limit at 24.

Successful recovery. In pytest task `37f1649f`, commands 1–2 locate and read the public test, command 3 reports the plugin-specification error, and command 4 reads the supplied parser. Command 6

replaces the reversed guard with `not isinstance(specs, (list, tuple))`. The public check passes at command 7, and the agent submits at command 12. Further testing yields 70 passing collection tests. An additional plugin test outside the acceptance suite fails. The frozen task grader accepts the final patch; the N/R/G/reference outcomes are 0/0/1/1.

Unfinished investigation. In xarray task `34bdfd8f`, REGROUND repeatedly runs the rolling reproducer, later inspects alignment code and `deep_align`, and reaches the 24-call limit without a patch. Raw History instead reads alignment code, restores `keys.append(no_key)`, obtains eight passing rolling tests, and submits at call 17. Access to a relevant file therefore does not ensure a successful edit. These sequences describe the difference in outcomes but do not isolate its cause.

Collection difficulty. In pytest task `6f69b8e2`, collection probes and a temporary test module produce no collected items. REGROUND reaches the call limit without a patch; the N/R/G/reference outcomes are 0/0/0/1. Outcomes are determined from test output and the final grader, not solely from shell-pipeline exit status when commands end in `head` or `tail`.

E.3 DIFFERENT REPAIRS AT THE SAME HISTORICAL LOCATION

Let $\mathcal{A}_L$ denote the edits at historical location L. Failed replay of one action $a_H \in \mathcal{A}_L$ does not establish failure of every action in that set. The same-locus study tests this distinction through paired execution outcomes: $w(r[a_H]) = 0$ and $w(r[a'_L]) = 1$ for a different edit a'_L at the same location, together with a validated historical success for a_H.

Nested representations. The constructions instantiate two nested representation changes on real consumer–provider relations. Historically, the provider boxes a value once and the consumer edit unboxes it once. In the current state, the provider boxes it twice. The historical edit still applies, but another unboxing at the same consumer is required. Each verified case has five executions: the historical state without and with a_H, followed by the current state without a repair, with a_H, and with a'_L. Their required outcomes are FAIL/PASS/FAIL/FAIL/PASS.

The public function's signature and body remain unchanged, and the same parameterized test bodies execute in all five states. Skip/xfail results and fixture or collection failures do not satisfy the behavioral checks. The adjusted patch validates the construction; it is not supplied as the agent's repair.

Fifteen candidates establish the full relationship across seven repositories, eight carriers, 11 witnesses, and 15 consumer–provider relations. The remaining five do not establish the historical contract. For two scikit-learn cases, unavailable observations are completed after restoring prebuilt package extensions in a disposable workspace; the previously successful state observations are retained.

Flask example. In Flask, the historical and current mutations change the return value of `make_null_session`, a method of `SessionInterface`. The consumer in `RequestContext.push` requires one additional unwrapping in the current state:

```
Public entry: Flask.wsgi_app
Test: tests/test_async.py::test_async_route
Stored edit: session_interface.make_null_session(self.app)[0]
Adjusted:    session_interface.make_null_session(self.app)[0][0]
```

The public entry and five route parameters remain unchanged throughout. The location remains a valid repair site, but its operation must change to handle the new returned representation.

E.4 DELAYED INTERVENTION DIAGNOSTIC

This diagnostic is separate from the Controlled-55 and Natural-44 main evaluations.

The 24-task diagnostic supplies the same current trace to every condition and permits one delayed replay after a tool event touches the historical location. Replay is triggered on six tasks. N/R/delayed G/reference accept 23/22/22/24 tasks with 282/265/266/187 calls, 91/83/97/77 public-test executions, and 1,256/1,118/1,312/921 aggregate sandbox seconds. Delayed replay leaves Raw History's completion count unchanged while adding execution work.

F TOKEN WORKLOAD AND RESOURCE ACCOUNTING

All four API backbones provide complete input and output usage fields for these comparisons.

F.1 ACCOUNTING UNITS AND RUN SELECTION

For each backbone and condition, we sum provider-reported input and output usage across all assignments in a track, including unsuccessful runs. Input totals include cached tokens when the provider includes them in prompt usage. They measure recorded context volume, not cache-discounted billing. Comparisons are within a backbone and use its tokenizer and usage accounting.

F.2 INPUT ACCUMULATION AND GENERATED OUTPUT

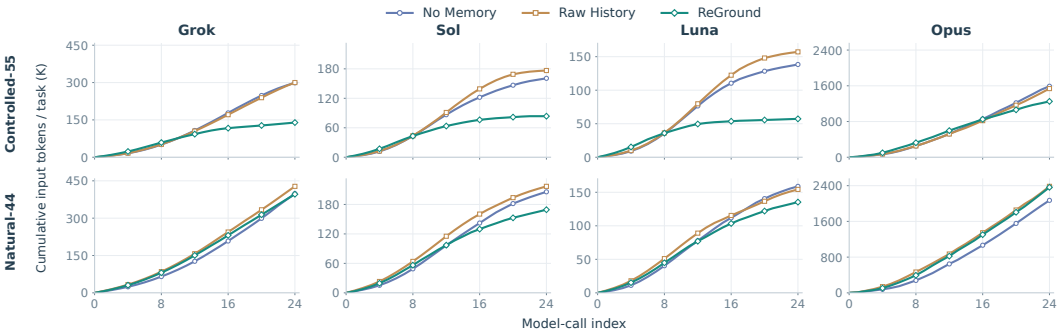

Figure 6: **Context accumulation across the repair budget.** At call index k, each curve reports cumulative input tokens divided by the full track size (55 or 44). Completed runs retain their accumulated total at later indices, keeping the task denominator fixed and including unsuccessful runs. The 1,188 API assignment histories reconcile with Table 15. Vertical scales are shared across tracks within a backbone but differ between backbones; markers occur every four calls.

Figure 6 shows when cumulative input usage separates across conditions. On Controlled-55, the Grok, Sol, and Luna curves diverge most clearly later in the interaction budget: N and R continue accumulating context while G's growth slows. Input accumulation is more sustained on the natural track, where the relationship between workload and additional completions differs from the controlled track.

Table 15: Complete-track token workload for the four API backbones. Each row includes every N/R/G assignment on that track. Input totals are in millions of tokens and output totals in thousands; reported input usage includes cached prompt tokens.

		Input tokens (M)			Output tokens (K)		
Backbone	Track	N	R	G	N	R	G
Grok	Controlled-55	16.44	16.50	7.66	308.8	445.6	398.1
Sol	Controlled-55	8.84	9.71	4.60	399.5	369.7	190.2
Luna	Controlled-55	7.61	8.64	3.15	470.1	394.3	194.8
Opus	Controlled-55	87.45	84.58	68.79	156.4	160.6	158.9
Grok	Natural-44	17.54	18.82	17.43	469.8	441.6	519.1
Sol	Natural-44	9.05	9.54	7.45	316.3	284.2	250.2
Luna	Natural-44	6.99	6.79	5.96	326.9	274.1	269.3
Opus	Natural-44	91.13	104.93	104.00	198.9	220.2	231.0

Controlled workload. On Controlled-55, G reduces total input volume by 21.3–58.6% relative to N across the API backbones. Grok uses 7.66M rather than 16.44M input tokens, and Sol uses 4.60M rather than 8.84M. Grok generates more output under G than under N while consuming less than half as much input context.

Natural workload. Natural-44 shows a different relationship between workload and repair output. Grok's total token volume remains close to that of N while accepted repairs increase from 15 to 25. Sol and Luna reduce both input and output totals. Opus uses more tokens under G than under N but increases accepted repairs from 12 to 19.

F.3 Token Work per Accepted Repair

For condition M, let T_M be the sum of reported input and output tokens across all assignments, and let s_M be the number of accepted repairs. The ratio T_M/s_M measures track-level token workload per accepted repair. Unsuccessful investigations remain in the numerator, so this differs from averaging token usage only over successful runs.

Table 16: Token workload per accepted repair, in thousands of input plus output tokens. Every assignment contributes to the workload numerator. The last two columns report the reduction for G relative to each control.

Backbone	Track	N	R	G	vs. N	vs. R
Grok	Controlled-55	380.7	423.7	155.0	59.3%	63.4%
Sol	Controlled-55	196.6	272.5	87.0	55.7%	68.1%
Luna	Controlled-55	202.0	258.3	63.1	68.8%	75.6%
Opus	Controlled-55	2367.7	7703.3	1407.0	40.6%	81.7%
Grok	Natural-44	1200.6	1204.1	718.1	40.2%	40.4%
Sol	Natural-44	374.7	427.2	285.2	23.9%	33.2%
Luna	Natural-44	332.6	371.7	230.5	30.7%	38.0%
Opus	Natural-44	7610.5	10514.8	5485.7	27.9%	47.8%

G reduces token workload per accepted repair for every API backbone on both tracks. Reductions relative to N are 40.6–68.8% on Controlled-55 and 23.9–40.2% on Natural-44; relative to Raw History, they are 63.4–81.7% and 33.2–47.8%, respectively. On the controlled track, lower input consumption accompanies more completions. On the natural track, higher repair yield also improves this ratio when total workload remains similar or increases.

F.4 Preparation and Execution Costs

The call and token results measure model interaction on all 99 tasks. Preparation adds repository execution work: current-witness execution, patch checks, at most one historical replay, source extraction, and workspace restoration, with no additional model calls. Already-present patches require a presence check rather than a historical replay. Model-service time, sandbox activity, and batch wall time are distinct costs; the reported call and token reductions are not estimates of end-to-end runtime savings.