Stale but Passing: What Task Success Hides About Documentation Debt in Coding Agents

Anonymous Author(s)

***Abstract*—Coding agents are guided by repository instruction files such as AGENTS.md, which are read in every agent session. As any documentation artifact, these files can fall behind the code, e.g., a test folder moves, and the file still points to the old one. In technical debt terms, such a stale instruction is documentation debt, and its interest is the cost it adds to every later agent session (e.g., additional tool calls to correct stale instruction). Evaluations that check only task success can miss this cost and the stale file can persist in the repository. In this study, we ask what leads agents to act on a stale instruction, how it fails them and at what cost, and how they recover. We study three real-world stale instructions from active open-source repositories in depth, each a different kind of guidance: a test command, a code location, and a procedure. Agents fix 22 maintenance tasks under the stale, corrected, and removed version of each instruction, with the code and the task held fixed, and we analyze 732 trajectories of two models at the level of individual tool calls. We find that agents follow stale guidance when it is relevant to their tasks, sometimes even when correct information is available elsewhere. Following stale instructions can lead to wasted tool calls and increased effort without affecting task success. On the stale test command, Claude Sonnet 5.5 needed about two extra tool calls per run, a third more than with the corrected command; in the other cases, we could not detect a difference. Although agents generally recovered from the immediate failures, none repaired the underlying instruction file. Correcting the instructions eliminated operations on obsolete paths, while removing them prevented most. Our findings show that task success alone can hide the costs of documentation debt, highlighting the need to evaluate agent behavior beyond final outcomes.**

***Index Terms*—documentation debt, coding agents, repository instructions, controlled evaluation**

I. Introduction

Developers steer coding agents through instruction files such as `AGENTS.md` or `CLAUDE.md`. These files, usually stored at the root of a repository, tell the agent which commands build and test the project, where its parts live, and which instructions to follow [3], [4]. Agents read these files at the start of their sessions and follow their guidance closely [3]. However, documentation tends to fall behind the code it describes: developers name outdated content among the documentation problems they encounter [5], and practitioners report causes and effects of such documentation debt [6]. Instruction files are no exception. Some are never updated after they are created [7], updates to others mostly add lines rather than remove them [8], and Treude and Baltes flag potentially stale code references in the instruction files of 23.0% of 356 repositories [9]. Consequently, agents may follow outdated guidance, such as commands or paths that are no longer valid; in constructed conflicts between an instruction and the code, agents follow the instruction [10], [11]. We refer to an instruction that remains in the file after becoming outdated as a *stale instruction*.

Fig. 1 illustrates how such instruction can affect an agent's work. The example comes from the static-analysis framework `pyflow` [1], whose `AGENTS.md` lists `pytest tests/ifds` as the command for running its IFDS tests. This command was valid until July 2026, when the tests moved to `tests/analysis/ifds`. However, the instruction remained unchanged through two subsequent edits of the file. To examine its effect, we introduced a small bug by negating a condition, causing one test to fail. We then asked the same agent to fix the bug under three conditions: with the stale instruction, with the instruction corrected, and with it removed. In all three runs, the agent located and fixed the bug using the same sequence of actions: a search, a read, and an edit (see Fig. 1). The difference appeared during verification. With the stale instruction, the agent ran the documented command, but the referenced directory no longer existed, so `pytest` executed no tests. Because the agent piped the output through `tail`, the shell reported a successful exit status. Although the output indicated that no tests had run, the agent needed two further test attempts before successfully verifying the fix. With the corrected instruction, verification took one call, while removing the instruction required two. All three runs fixed the bug and passed the final check on the original tests, making them indistinguishable under an evaluation based only on task success.

By observing this phenomenon through the lens of technical debt [12], the outdated or stale instruction is a documentation debt item, and the example shows both parts of that debt. The principal, the cost of repaying it, is a one-line edit. The interest is what each run pays while the line stays stale: here, two extra tool calls, half again the work of the corrected run, spent on a verification step whose failure looked like success. For one run this interest is small, but it is not paid once. The instruction file is loaded in every agent session, and the agent recovered without repairing the line, so every later task that reaches the line must pay again; if each pays a similar amount, the interest grows with the number of agent runs while the principal stays one line. Nothing in the outcome signals the debt, so it can stay: the bug is fixed, the tests pass, and the maintainer may learn nothing.

Existing research does not measure this kind of cost. Treude and Baltes detect stale references in instruction files and call for controlled experiments on how they affect agents, but they do not run agents themselves [9]. Studies that do run agents compare runs with an instruction file against runs without one,

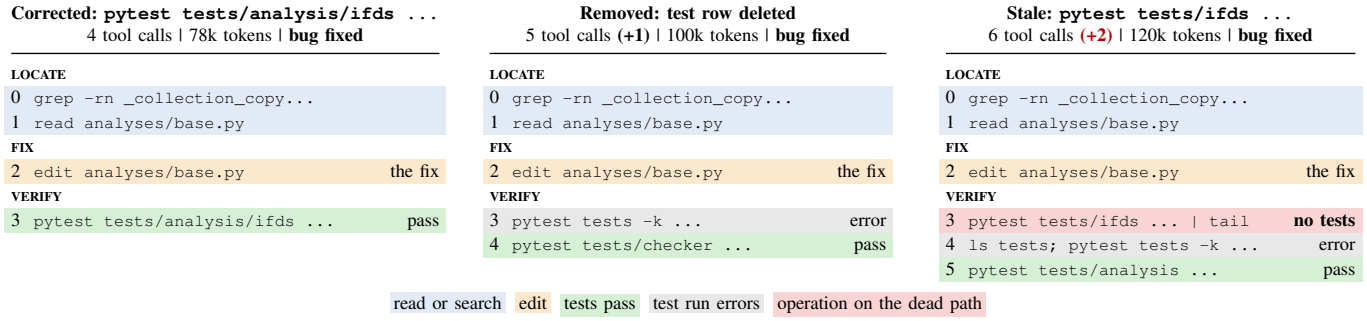


Fig. 1. The stale test command in `pyflow` [1], one of the three stale instructions we study: three runs of Claude Sonnet 5.5 on the same seeded bug (a negated condition), one per version of the line. Every tool call is shown (commands shortened), and (+n) counts calls beyond the corrected run. All three runs find and fix the bug the same way and differ only in checking the fix: one call with the corrected line, two without the line, and three with the stale line, whose documented command runs no tests while the pipe through `tail` reports success. All three pass our final check on the original tests. The two errors come from a test folder that needs a third-party module missing from the environment, unrelated to the bug. We chose the task whose runs varied least across repeats, and the repeat in which each run took its condition's median number of calls; the transcripts are in the replication package [2].

and assume that the file's content is correct [3], [4]. Work on stale instructions builds conflicts over coding conventions, for example a file that demands one style while the existing code uses another, and judge the outcome by the code the agent produces [10], [11]. In that setting, following the stale rule produces worse code, so checking the code reveals the harm. A stale path or command works differently: the repository itself contradicts it, so the agent hits an error, recovers, and still delivers a correct fix.

To investigate the consequences of stale instructions, we address three research questions:

RQ1. What leads an agent to act on a stale instruction?

RQ2. How does a stale instruction fail the agent, and at what cost?

RQ3. How does the agent recover, and does it surface or repair the underlying stale instruction?

To answer these questions, we study three real-world stale instructions in depth, each a different kind of guidance: a test command in `pyflow` [1], a code location in `cog` [13], and a procedure for adding a component in `aghub` [14]. Each was written by the maintainers of an active repository, was correct until a documented change in the code, and survived every later edit of its file (2, 13, and 4 edits; Fig. 2). Agents fix the same bugs with the stale, corrected, and removed version of each instruction, on identical code and under the same setup: 22 tasks, three versions, five to seven repeats, and two models give 732 runs (Section III). We measure task success, operations on the stale path, tool calls, and tokens, and rather than measuring many instructions shallowly, we follow every run tool call by tool call, which lets us explain why agents act as they do: how they came to take the stale path, what happened there, and how they recovered.

Agents took the stale path in two of the three cases: the test command in 78 of its 80 stale runs, the procedure in about two thirds, and the code location never, because no task needed it. The task, not the line alone, decided this: agents used a stale line when their task led them through what it names and gave them no other starting point, and they took the path silently from the loaded file, sometimes even when the bug report named the correct location. At the stale path, the failure often looked like a success: the test command found no tests, but agents piped its output through `tail`, so the call was reported as successful, and in 19 of Sonnet's 40 stale runs the first check of the fix ran no tests. The cost was a few tool calls: about two per run for Claude Sonnet 5.5 on the test command, a third more than with the corrected line; in the other cases, we detected no difference in tool calls or tokens. Task success hid all of this, since agents fixed the bug in nearly every run under every version. Agents worked around the stale path within a few calls, mostly by searching the repository, told the user mainly when the failure hit the check of their own fix, and never repaired the instruction file, even when they noticed other outdated parts of it. Correcting the instruction removed the wasted work. Deleting it removed most of it but not all: agents still guessed the old folder, and on the test command Sonnet needed as many tool calls without the line as with the stale one, because it had to work out how to run the tests itself (Fig. 1).

This paper makes three contributions. First, it compares stale, corrected, and removed versions of real-world stale instructions under controlled conditions and shows that a stale instruction can make agents waste work, including checks that test nothing, while task success stays the same. Second, it shows that this cost depends on whether the task leads the agent through the stale line, and that correcting a stale line works better than deleting it. Third, from the agents' trajectories, it explains why the debt stays hidden: agents meet the line where their workflow needs it, the failed check is reported as a success, and agents that notice a problem work around it without repairing the file. From this, we draw practical lessons: evaluations of instruction files should report what agents do, not only whether they succeed, and harnesses and agents can guard against stale paths and silent test failures.

II. Related Work

This section reviews documentation debt, repository instruction files and how they go stale, studies of whether instruction files help agents and of instructions that conflict with the code, and evaluations that look beyond task success. It then positions our study and closes with our own evidence that stale instructions persist.

Documentation debt. The technical-debt metaphor describes shortcuts that incur recurring costs, or *interest*, until the *principal* is repaid [12], [15]. Documentation that no longer matches the code is a recognized form of such debt [16], [17]. Developers report outdated content among the documentation problems they encounter [5], and Rios et al. survey practitioners about the causes, effects, and management of documentation debt [6]. Researchers have also developed techniques to detect documentation made obsolete by code changes, including comments inconsistent with source code [18], [19] and outdated code-element references in README files and wikis [20]. For human readers, however, the effort associated with such debt can be difficult to observe: a developer may encounter an outdated reference, work around it, and leave no record of the detour. For coding agents, the same detour may appear in recorded tool calls, even when the task is completed successfully.

Instruction files and how they go stale. Repository instruction files, such as `AGENTS.md` and `CLAUDE.md`, provide project-specific instructions to coding agents and are widely used to configure how they work [21]. They commonly specify how to build and test a project, where code is located, and how to carry out routine changes [7], [8], [22]. Previous studies report two maintenance patterns that may contribute to outdated instructions: some files are never updated after creation [7], while updates to others consist mainly of small additions with few deletions [8]. Treude and Baltes describe inconsistencies between agent configuration files and repository code as *context rot* [9]. Applying a checker originally built to find outdated code references in README files [20] to the instruction files of 356 repositories, they flag potentially stale code-element references in 23.0% of them and conclude that such inconsistencies are common. Their study focuses on detecting inconsistencies rather than observing agents. In their research roadmap, they call for controlled experiments to determine which forms of context rot affect coding-agent behavior. A *stale instruction*, as we use the term, is one specific form of context rot: a line that was once correct and that a later code change contradicted. We use the narrower term because our experiment varies exactly one such line. Because agents use instruction files as they work, the consequences of an outdated reference may be visible in their recorded actions.

Do instruction files help agents? Another line of research examines how instruction files affect coding agents, generally without testing whether their contents remain current. Gloaguen et al. compare agents given no instruction file, an LLM-generated file, or a developer-written file [3]. They find no consistent improvement in task success, despite increased inference costs, and observe that agents follow the instructions they receive, for example by invoking tools named in the files. Lulla et al. evaluate an agent on 124 pull requests with and without `AGENTS.md`, reporting lower median runtime and output-token use when the file is present [4]. Shepard and Albrecht take a different approach, refining instruction files through synthetic bug-fix probes and finding that the refined instructions help agents locate relevant code and resolve more tasks [23]. Together, these studies show that instruction files can shape agent behavior, although their effects on success and efficiency vary across settings. They primarily examine whether instructions are provided or how to improve them, rather than isolating the consequences of instructions that were once correct but have become outdated.

Stale and conflicting instructions. Two recent studies investigate constructed conflicts between repository instructions and existing code. Xu and Wu create 40 Python projects in which instruction files prescribe coding conventions that differ from those used in the code [10]. Without intervention, Claude Code and Codex CLI follow the conflicting instructions in 90.1% of scored choices, with Claude Code often acknowledging the conflict. Adding a sentence that directs agents to prioritize existing code reverses this tendency. Mohammadi et al., as part of a broader study of what agents need to know to edit coherently, compare three conditions: a standard that agrees with the code, no standard, and a standard that prescribes a less desirable practice [11]. Agents choose the preferred implementation in 100%, 33%, and 0% of scored decisions, respectively. In that setting, a conflicting standard leads to worse decisions than no standard. Both studies show that agents may follow incorrect rules about *how code should be written*, and both assess the consequences through the resulting implementation decisions or code.

Our setting differs in the kind of outdated information and the consequences we examine. We study test commands, code locations, and development procedures that were once valid but became obsolete when the repository changed. Following such a reference need not damage the final repair: an agent may encounter the obsolete location, recover, and still complete the task correctly. The consequence may instead appear in the operations the agent performs along the way. We examine when agents use these references, what happens when they do, and how they recover.

Looking beyond task success. Coding agents are often evaluated by whether their final patches pass benchmark tests. Several studies question whether passing tests adequately establish patch correctness: 29.6% of plausible SWE-bench patches behave differently from the developers' fixes [24], while strengthened tests reject about one in five previously accepted patches from top agents [25] and test augmentation reduces top agents' resolved rates by 4.2–9.0% [26]. Other studies look beyond the final patch to examine the steps agents take. Analyses of agent trajectories associate context gathering before editing and subsequent validation with successful outcomes [27], [28], while another study finds that failures can begin early and remain undetected until recovery becomes

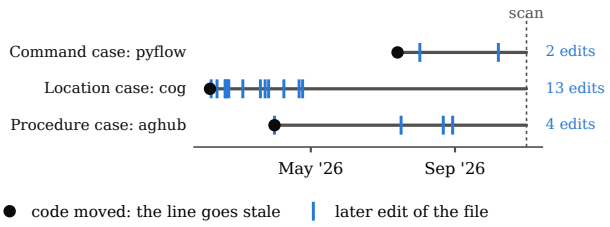


Fig. 2. Stale lines survive maintenance. Each row is the instruction file of one studied case, drawn from the day the code its stale line names moved (dot) to our scan (dotted line); ticks and the count on the right are later edits of the file. Every edit kept the stale line: 2 in the Command case (pyflow [1]), where the tests moved on July 14, 2026, 13 in the Location case (cog [13]), and 4 in the Procedure case (aghub [14]).

difficult [29]. AgentLens identifies "Lucky Passes," in which agents pass benchmark tests despite problematic processes such as blind retries or missing verification [30]. TraceProbe similarly shows that successful runs can differ in the amount of unsuccessful work they perform [31].

These studies highlight two limitations of pass rates. First, passing tests may conceal an incorrect patch. Second, a successful repair may still involve ineffective or unnecessary work. Trajectory analyses can expose such behavior, but the observational studies discussed here do not isolate the effect of a particular outdated instruction. Moreover, a measure that categorizes a test invocation as validation may not distinguish a meaningful test run from one that executes no tests.

Our position. We connect these lines of work. Prior work establishes that repository instructions can become outdated, that agents may follow information conflicting with the code, and that task success alone does not fully describe how agents work. What remains less understood is how a real-world stale instruction enters an agent's workflow, what happens when the agent acts on it, and whether recovery removes the source of the problem. We examine these questions by varying real-world stale references in repository instruction files. For each case, we compare three maintenance decisions: leaving the reference unchanged, correcting it, or removing it, while holding the code, task, model, and harness fixed. As in other controlled studies that vary one factor while keeping others constant [32], [33], differences between conditions can be attributed to the instruction change within the studied tasks. Stale versus corrected evaluates the consequence of leaving an obsolete reference in place; corrected versus removed compares repairing the instruction with deleting it. In two of our three cases, obsolete references changed what agents did, most notably prompting a documented test command that ran no tests while appearing successful, without a detected difference in task success. Whereas constructed convention conflicts produced near-uniform instruction following [10], [11], the frequency of operations on obsolete references varied across our three cases, from nearly every run to none.

III. Study Design

In this section, we describe the methodology used to investigate how coding agents interact with stale repository instructions. We begin by explaining how we selected three real-world stale instructions for in-depth study (Section III-A). We then introduce the selected cases and the kinds of guidance they provide (Section III-B). Next, we describe the maintenance tasks constructed for each case (Section III-C) and the experimental setup used to compare stale, corrected, and removed instructions (Section III-D). Across 22 maintenance tasks and two models, we hold the code and tasks fixed while measuring task success, operations on obsolete paths, tool calls, and token consumption. These measurements allow us to examine when agents follow stale instructions (RQ1), the costs they incur (RQ2), and how they recover, what they report, and whether they repair the underlying instruction files (RQ3).

A. Case Selection

We searched GitHub code search for repositories with a root-level `AGENTS.md` or `CLAUDE.md` file. A case is one stale line in one of these repositories, and to answer our questions it had to meet five requirements. First, the repository is active: it is neither a fork nor archived, has at least five stars, and has at least one commit after the last change to its instruction file, so the code could have moved on since the file was written. Second, the instruction file names a path that existed in the repository until a code change moved it, while the file still presents the old path as current guidance; only real-world stale lines show how agents treat this debt as it occurs in practice. Third, the path moved rather than disappeared, so that a corrected version of the line exists; without one, we could not compare stale, corrected, and removed versions. Fourth, the line asks the agent to act, by running a command, going to a location, or following steps, so that whether the agent follows it can be observed. Fifth, the repository has a runnable test suite, so that tasks can be built from its tests and every run can be graded. The search queries, the sample, and the seed are in the replication package [2].

We kept only active repositories during the search and checked the other requirements in two steps. First, a script extracted every path the sampled files name (inline code and link targets) and checked each against the repository's current state and its commit history, flagging paths that once existed but no longer do; two coding-agent reviews (Claude and Codex) then confirmed whether each flagged line still presents its path as current guidance. Second, we checked the remaining requirements for each confirmed stale line by inspecting its repository and history. Three stale lines met all of them, one of each kind of instruction: a command to run, a location to go to, and steps to follow. After the coding-agent reviews, the first author manually reviewed every flagged line, including the three selected cases. We study all three in depth. In each, the line was correct before a documented change and wrong after it, with no alias keeping the old path working, and it stayed in its file through later edits. The search served only to find cases, not to count them; it covers a small sample of repositories that

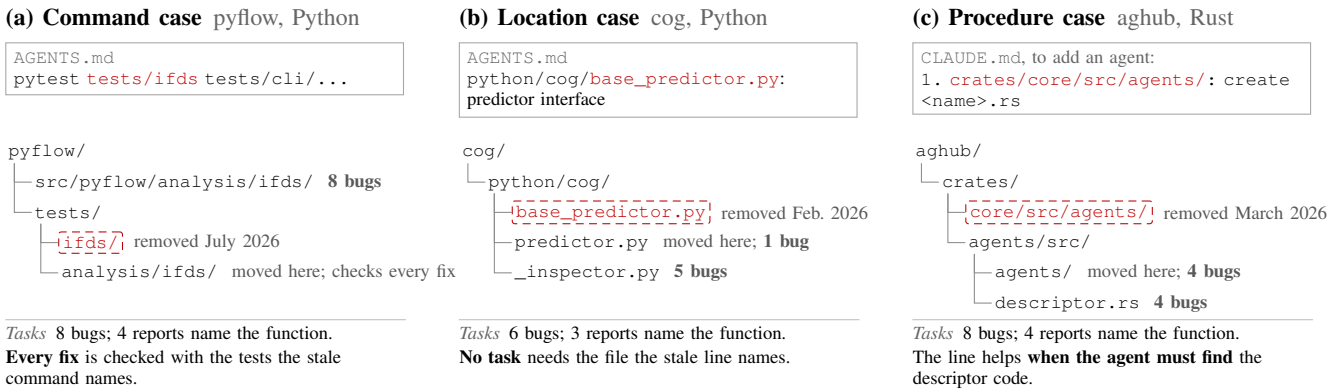


Fig. 3. The three studied cases and their tasks. Each instruction file names a path (red, dashed) that a later commit removed when the code moved; the tree shows where it moved and where the bugs of the case's tasks lie, and the block below summarizes the tasks. Instruction text quoted from the files and shortened; all 22 tasks are listed in the replication package [2].

use instruction files, and stale lines are common in practice [9]. The extraction rules, review labels, and the outcome of each step are in the replication package [2].

B. Studied Cases

We call the three cases by the kind of guidance their stale line gives: the Command case, the Location case, and the Procedure case (Fig. 3). Together they cover the three kinds of guidance that instruction files commonly give [7], [8]: how to test the project, where code is located, and how to carry out a routine change. They also differ in how an agent uses them: a test command is executed as written, a code location is one of several ways to find code, and a procedure is followed step by step. This contrast in cases lets us examine when each kind of stale line enters an agent's work, what it costs, and how agents recover.

- The **Command case** comes from pyflow [1], a Python static-analysis framework. Its instruction file gives the command for running one group of tests. The tests moved to a new folder on July 14, 2026, but the command still names the old one, through two later edits of the file.
- The **Location case** comes from cog [13], a Python tool for packaging machine-learning models. Its instruction files name the file that holds the predictor interface, which has since moved to another file.
- The **Procedure case** comes from aghub [14], a Rust project. Its instruction file lists the steps for adding an agent and places the new code in a folder that has since moved to a separate package; a second instruction file in the repository already gives the new location.

C. Tasks

Every task is a bug in the repository's own code that the repository's own tests catch. In the Command case, every fix is checked with the tests the stale command names; in the Procedure case, the agent needs the line when it must find the descriptor code. The Location case serves as a contrast. Its bugs lie in the same package as the file the stale line names, in the code that checks the predictor interface, but none of its tasks needs that file. The stale line is still in the agent's context in every session, and the agent works right beside the stale path. The case therefore tests whether a stale line pulls an agent off course when the work does not pass through it. Fig. 3 summarizes the tasks of each case; the full list is in the replication package [2].

TABLE I
RUNS PER CASE. EVERY TASK RAN IN ALL THREE CONDITIONS WITH BOTH MODELS.

Case	Tasks	× Conditions	× Repeats	× Models	= Runs
Command case	8	3	5	2	240
Location case	6	3	7	2	252
Procedure case	8	3	5	2	240
Total	22				732

D. Conditions, Runs, and Measures

Fig. 4 shows how each run is built and graded. Each task runs in three conditions (i.e. the three things a maintainer can do with the line): leave it stale, correct it to the location the code moved to, as recorded in the repository's history, or remove it. Each comparison isolates one question. Stale against corrected differs only in whether the path is right, so it measures what the wrong content itself costs, the interest of the debt, and the corrected runs show how the same task proceeds without it. Removed against the other two separates wrong guidance from missing guidance: it shows whether agents need the line at all, whether they return to the old path on their own, and whether deleting the line, the cheapest repair, is enough. Corrected and removed versions change every mention of the stale path in the instruction files; nothing else differs.

We ran every task in each condition with two models, Claude Haiku 5.5 and Claude Sonnet 5.5, so that a finding does not depend on one model's habits; both ran in Claude Code, with about 40 runs per condition, case, and model (Table I).

For each run we record whether the bug was fixed, whether the agent ran, read, or wrote the stale path (our primary measure), and its tool calls and tokens. To see why agents

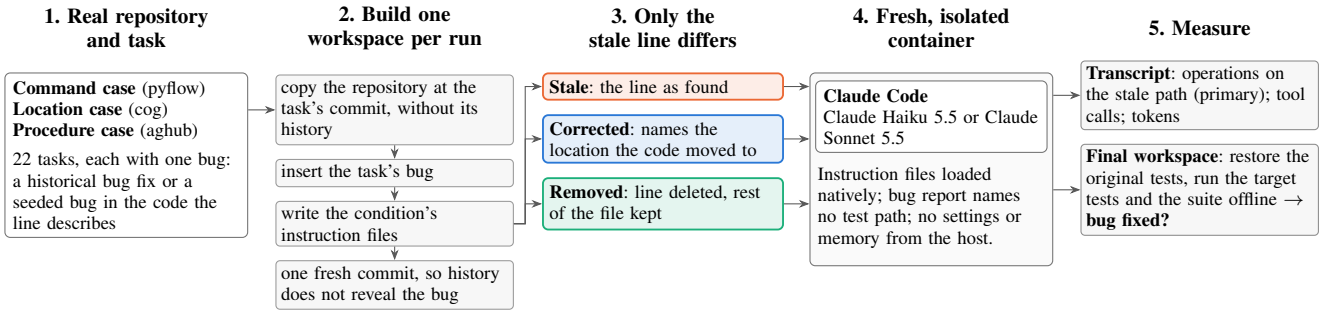


Fig. 4. How each run is built and graded. Every run starts from a real repository at the task's commit with one bug; between conditions, only the stale line of the instruction files differs. Each run is isolated in a fresh container, and outcomes are read from its transcript and from its final workspace, graded on the original tests.

took the stale path and how they recovered, we coded every run that used it from its transcript with fixed rules: what the first stale call was for and what the agent did and said before it, where the call fell relative to the first edit, what it returned and whether the tool marked it as an error, how the agent reached the current location, and whether its messages named the stale path; for the test command, we also checked whether the first test run after the fix ran any tests. Setup details, the coding rules, and the per-run codes are in the replication package [2].

Agent and runs. The agent was Claude Code 2.1.316 at medium effort, with a 30-minute limit and a budget cap per run. Its prompt was the bug report alone, so it received the instruction files only through Claude Code's own loading, and it could run commands without asking for permission. Each block held one run of each condition for one task; the three runs ran together, and blocks ran in shuffled order, so the conditions shared time and machine load. Failed runs were retried; four Haiku retries violated the intended fresh-workspace rule by retaining earlier edits. The replication reconciliation reports these exceptions and the supplied sensitivity analysis. A bug counted as fixed when the target tests and the suite passed offline on a copy of the final workspace, with separate test files restored; Procedure outcomes use sequential regrading after concurrent grading suffered memory failures.

Measures and analysis. We measure token consumption across the entire run, including input, output, and cached tokens, accounting for context reused across model calls. We compare stale-path use and task success separately for each model using exact tests stratified by task, with Holm correction for multiple comparisons. We assess differences in effort using per-task medians and bootstrap confidence intervals. Our analyses of agent trajectories are exploratory and are used to explain how agents encounter stale instructions, respond to failures, and recover.

IV. Results

Table II gives the results for each case and condition. We report them in the order in which an agent meets a stale line: what leads it to take the stale path (RQ1), how the stale path fails it and at what cost (RQ2), and how it recovers and whether it surfaces or repairs the stale line (RQ3).

RQ1: What leads an agent to act on a stale instruction?

Agents took the stale path in two of the three cases. With the stale line, agents used the stale path in 78 of 80 runs in the Command case, 55 of 80 in the Procedure case, and none of the 84 in the Location case (Table II); with the corrected line, no run did. Stale against corrected is significant for each model after Holm correction (Haiku $p = 2 \times 10^{-28}$, Sonnet $p = 2 \times 10^{-33}$).

The task decided whether the line was needed. A stale line mattered only when the task led the agent to what the line describes. Every Command task ends with running the tests, so agents met the stale test command with either kind of bug report. The Procedure line says where the agent descriptors live, and the broad report pointed right there ("some agent descriptor behavior is wrong"), so both models used the stale folder in nearly every such run (Haiku 20, Sonnet 19 of 20); when the report named the function, Haiku used it in 4 of 20 runs and Sonnet in 12 of 20. In the Location case, the agent never had to look anything up: a report that named the function also named its file, for one bug the predictor file under its current name, and a broad report led the agent to list the package or run the tests, which showed the faulty file. With the right file in hand within two calls, no agent turned to the outdated location in the instruction file.

The line filled a missing starting point. With a broad report, Haiku's first call listed the stale folder ("I'll start by looking at the agent descriptors and the registry"), while Sonnet first ran the tests (18 of 19 runs) and then looked for the descriptor code of the failing test in the stale folder. With a named function, Haiku searched for it by name, but Sonnet searched inside the stale folder in 12 of 20 runs, although the report named the function's current module.

Agents adopted the path silently. Before their first stale call, agents cited the instruction file in 2 of the 133 runs that used the stale path and opened it in none; the path came from the file that Claude Code loads into the session. Their messages described the task, not where the path came from, and Sonnet copied the documented test command whole in 35 of its 40 runs in the Command case.

Removing the line prevents most of it. With the line removed, agents still used the stale path in 1 of 80 runs in

TABLE II

RESULTS PER CASE AND CONDITION, FOR EACH MODEL AND FOR BOTH TOGETHER. STALE PATH: RUNS THAT RAN, READ, OR WROTE THE STALE PATH. FIXED: RUNS WHOSE FINAL WORKSPACE PASSED THE ORIGINAL TESTS. CALLS AND TOKENS: PER-TASK MEDIANS AVERAGED OVER THE CASE'S TASKS; FOR BOTH MODELS, THE MEDIANS ARE OVER BOTH MODELS' RUNS OF EACH TASK. IN ONE TEST PER MODEL OVER ALL THREE CASES, STALE RUNS USED THE STALE PATH MORE OFTEN THAN CORRECTED RUNS, SIGNIFICANTLY AFTER HOLM CORRECTION (HAIKU $p = 2 \times 10^{-28}$, SONNET $p = 2 \times 10^{-33}$); THE DIFFERENCE COMES FROM THE COMMAND AND PROCEDURE CASES, SINCE NO RUN IN THE LOCATION CASE USED THE STALE PATH.

		Claude Haiku 5.5				Claude Sonnet 5.5				Both models			
Case	Condition	Stale path	Fixed	Calls	Tokens (k)	Stale path	Fixed	Calls	Tokens (k)	Stale path	Fixed	Calls	Tokens (k)
Command case	Stale	38/40	39/40	16.0	536	40/40	40/40	7.8	195	78/80	79/80	10.2	278
	Corrected	0/40	39/40	18.2	648	0/40	40/40	5.8	152	0/80	79/80	9.6	303
	Removed	0/40	39/40	14.0	454	1/40	40/40	7.8	222	1/80	79/80	10.9	352
Location case	Stale	0/42	42/42	7.5	222	0/42	42/42	5.3	157	0/84	84/84	6.1	175
	Corrected	0/42	42/42	7.3	220	0/42	42/42	5.5	162	0/84	84/84	6.3	186
	Removed	0/42	42/42	7.7	224	0/42	42/42	5.3	157	0/84	84/84	6.4	184
Procedure case	Stale	24/40	39/40	12.4	542	31/40	40/40	7.9	294	55/80	79/80	9.6	391
	Corrected	0/40	38/40	11.2	493	0/40	39/40	6.8	249	0/80	77/80	9.6	395
	Removed	13/40	39/40	10.5	402	7/40	40/40	7.8	283	20/80	79/80	8.9	339

the Command case and 20 of 80 in the Procedure case, all 20 with a broad report: no file named the old folder, but they guessed an `agents` folder inside the core crate from the code's structure. Only the corrected line prevented this.

Answer to RQ1. Agents took a stale path when the task led their strategy through what the line names and gave them no other starting point: always for the test command, mostly for the procedure when the report named only the area, and never for the code location. They took the path silently from the loaded file. Correcting the line prevented this in every run, and removing it prevented most of it.

RQ2: How does a stale instruction fail the agent, and at what cost?

The stale test command failed quietly. Because the old test folder no longer exists, the documented command collected no tests and exited with an error. In 64 of the 80 stale runs in the Command case, at least one call on the stale path ran no tests, and Claude Code flagged none of these calls as an error: every run that ran the command piped its output through `tail` or `head`, so the shell reported success. In all 40 Sonnet runs, the first such call printed that the folder was not found and that no tests ran, yet it counted as a success; Haiku often hid the message itself (`2>/dev/null`, 16 of its 38 runs). As a result, 19 of Sonnet's 40 stale runs first checked their fix with a test run that ran no tests, against none of its 40 corrected runs. The missing folder of the Procedure case was flagged more often, in 20 of 24 Haiku runs and 13 of 31 Sonnet runs.

When the agent met the line. With a broad report, Sonnet met the stale command while reproducing the bug (20 of 20 runs); with a report that names the function, it met it while checking its fix (19 of 20). Haiku explored first whatever the report and met the command before editing in 34 of its 38 runs.

Agents still fixed the bug, but with more work. Agents fixed the bug in nearly every run under every condition ($p = 1.0$ for both models). For tool calls and tokens, we could resolve only one difference, and it is exploratory: on the test command, Sonnet's stale runs used 7.8 tool calls against 5.8 for its corrected runs, with the stale median higher in seven of the eight tasks; the 95% interval excludes zero, though not after correcting for the number of estimates. Its tokens rose as well (195k against 152k), but not significantly. In four of these tasks, all 15 runs across the three conditions produced the same patch, byte for byte, so only the work to reach it differed (Fig. 5).

The cost followed the report. On the procedure, the stale line cost both models about two tool calls per task with a broad report (2.25 for Haiku, 2.0 for Sonnet) and almost none with a named function (0.0 and 0.25), even though Sonnet still entered the stale folder in most of those runs (RQ1). On the test command, Sonnet paid about two calls per task under both kinds of report, while reproducing the bug or while checking the fix. Haiku's runs varied too widely within a task to resolve a difference (on the test command, from 18 fewer to 9 more calls per task). All per-task comparisons are exploratory.

Answer to RQ2. At the stale path, the agent often meets a failure that looks like a success: a test command that runs no tests is reported as passing. The stale line cost a few tool calls each time an agent met it, measurably only for Sonnet on the test command (about two calls per run, a third more than with the corrected line; exploratory). Task success did not change, so an evaluation that checks only the outcome does not see the cost.

RQ3: How does the agent recover, and does it surface or repair the underlying stale instruction?

Most encounters ended in operational recovery. Table III starts from the 133 *encounters*: stale-condition runs that operated on the studied obsolete path, which form the denominator for each row. The *reached target* column records whether a later call actually ran tests off the stale path in the Command case or operated on the current folder in the Procedure case. By this measure, agents recovered in 130 of 133 encounters. The *via search* column identifies the subset of recoveries preceded by a repository search; 98 recovered through search, most often Sonnet in the Procedure case (28 of 31), which searched the repository once the folder was missing, while 32 reached

TABLE III

RECOVERY AND DOCUMENTATION-DEBT RESPONSE AFTER AN AGENT OPERATED ON THE STALE PATH. PERCENTAGES USE THE ENCOUNTERS IN THE SAME ROW AS THE DENOMINATOR. "VIA SEARCH" IS A SUBSET OF RUNS THAT REACHED THE CURRENT TARGET.

			Operational recovery, n (%)		Documentation-debt response, n (%)		
Case	Model	Encounters	Reached target	Via search	Final disclosure	Suggested repair	Edited instruction
Command	Haiku	38	36 (95)	27 (71)	3 (8)	0 (0)	0 (0)
Command	Sonnet	40	40 (100)	27 (68)	17 (43)	0 (0)	0 (0)
Procedure	Haiku	24	23 (96)	16 (67)	1 (4)	0 (0)	0 (0)
Procedure	Sonnet	31	31 (100)	28 (90)	0 (0)	0 (0)	0 (0)

Note: Final disclosure requires the final response to identify the studied mismatch, not another documentation issue. Suggested repair requires an explicit proposal to correct the instruction. No instruction file was edited in any of the 732 runs.

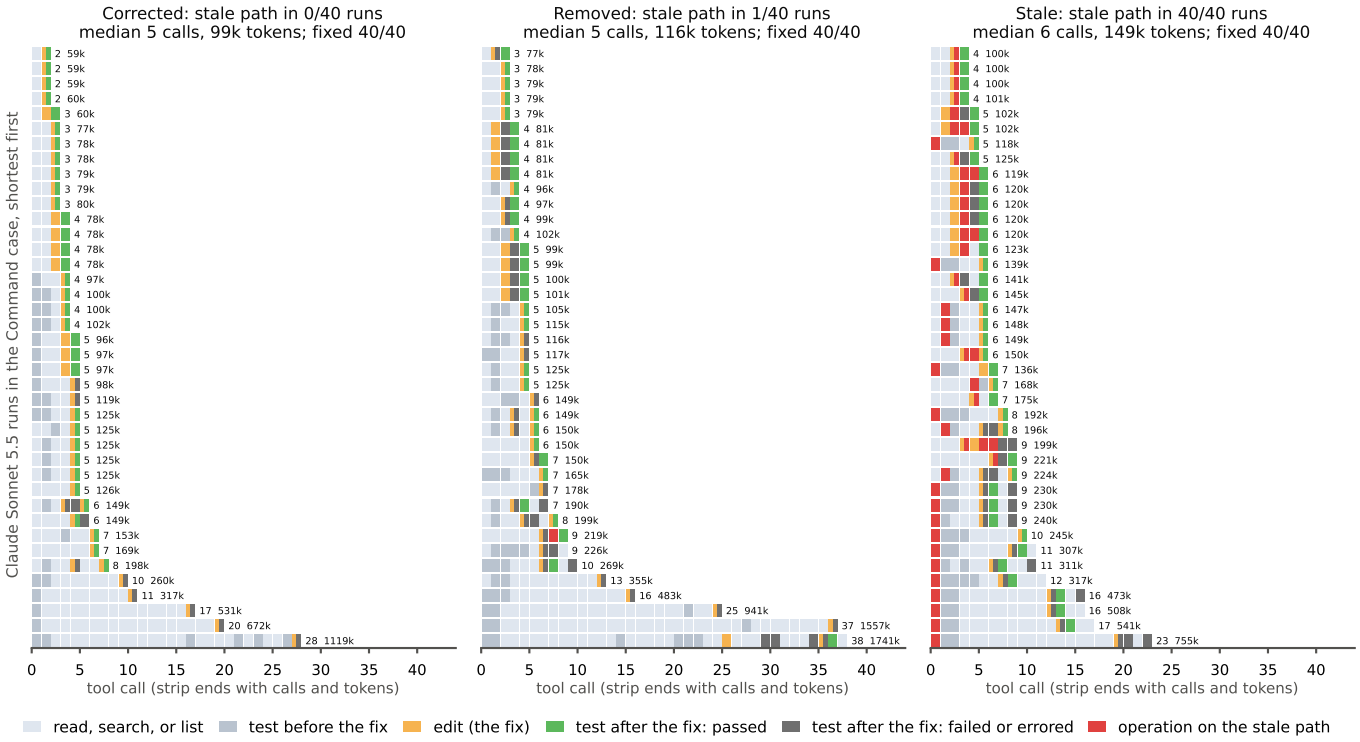


Fig. 5. The stale test command adds calls, mostly around and after the fix. Every run of Claude Sonnet 5.5 in the Command case, 40 per condition, as a strip of tool calls colored by what each call did; a half cell is one shell call that edits and then tests. Each strip ends with the run's tool calls and tokens. All 120 runs fixed the bug. The longest runs in every condition spend their calls reading and searching for the bug. In stale runs, the extra calls are operations on the stale path around the fix, usually the documented command, which runs no tests, followed by more test runs. Tokens grow with calls, but the per-case token difference is not resolved (Section IV).

the target directly. Recovery took a median of two calls after the first stale-path operation; the case–model medians ranged from two calls for both models in the Command case to four for Haiku in the Procedure case. The other three trajectories did not reach the current target before they ended. These post-encounter sequences describe how agents worked around the stale guidance; because they include only runs that took the stale path, they are not causal estimates of its net cost.

Recognition did not reliably reach the user. The *final disclosure* column is stricter than noticing a problem during execution: it counts a run only when the final response identifies the studied mismatch, not another documentation issue. Only 21 of the 133 encounters met this definition. Sonnet named the stale test folder in 17 of its 20 runs whose report named the broken function, where it usually met the command while checking its fix, but in none of its 20 runs with a broad report, where it met the command while reproducing the bug. Haiku named it in 3 of its 38 Command runs. In the Procedure case, only one final response reported the mismatch, describing the old layout without naming the folder, although Haiku often noted during the trajectory that the layout differed from `CLAUDE.md`; several Haiku reports instead discussed other outdated parts of the file. Thus, noticing a mismatch during execution did not imply that the maintainer received it in the

final report. These report-level patterns are descriptive: report type is tied to the task rather than independently randomized.

Disclosure did not become repayment. The last two columns distinguish recommending repayment from performing it. The *suggested repair* column comes from exploratory coding and requires an explicit recommendation to correct the instruction; *edited instruction* requires an actual change to the instruction file in the final workspace. Neither occurred. Final responses that identified the studied mismatch described the missing path or the replacement used to finish the task, but none proposed correcting the instruction, and no run in any condition edited an instruction file. The task asked the agent to fix a code defect, not to maintain documentation, so this is not evidence that editing the file was required for task success. It instead shows how the debt persists during ordinary maintenance: the agent can work around the wrong instruction, deliver a passing patch, and leave the same obstacle for the next session.

Answer to RQ3. Agents usually restored task progress within a few calls, often by searching for the current target, but recovery rarely produced an actionable maintenance signal. Mismatches noticed during a trajectory were often absent from the final report; reports that did name the studied mismatch did not recommend correcting it; and no run repaired the instruction file. Operational recovery therefore removed the immediate blockage without repaying the documentation debt.

V. Discussion

The debt is paid in behavior, not in outcomes. Agents fixed the bug in nearly every run, whatever the condition, but a stale line changed how they got there: they ran a test command that tested nothing, which was even reported as a success, or searched a folder that no longer exists. In technical debt terms, the principal is a one-line edit, and the interest is the extra work an agent does each time it meets the line. How much interest a line charges depends on what one encounter costs and on how often tasks reach the line; we measured only the first. It varied with the line and the model, and it was small when the task offered another way forward: with a report that named the function, even entering the stale folder cost Sonnet almost nothing. The debt can build up because task success, the measure most evaluations use, does not show it, and because later edits of the file left the stale line in place in all three cases (Fig. 2).

Not every stale line carries interest. Agents followed the stale test command almost always, the stale procedure in about two thirds of runs, and the stale code location never. All three lines ask the agent to act; what separated them was whether the task led the agent through what the line names. Every fix ends with running the tests, and agents ran the documented command as written (Sonnet copied it whole in 35 of its 40 runs). The procedure mattered when a report gave no other starting point. No task needed the code location, so that case shows that this line did not matter for these tasks, not that code locations never matter. A stale line can also win over better information in the task: Sonnet searched the stale folder in 12 of its 20 runs even though the report named the function's correct module. Agents almost never said where the path came from (2 of 133 runs), so someone reading a trajectory cannot tell that the instruction file misled the agent. We therefore expect the interest to be largest for lines on a step most tasks share, such as the command that checks every fix, and maintainers with limited time may want to check those commands first. With three cases, the kind of instruction cannot be separated from its repository and tasks, so this remains a hypothesis for larger studies.

Recovery hides the debt from those who could repay it. The agent pays the interest, but only the maintainer can repay the principal, and recovery keeps the two apart. Agents reached the current location in 130 of the 133 runs that took the stale path, mostly by searching the repository (98 runs) rather than by correcting the path, so they finished their task without fixing the file. Whether the maintainer heard about the problem depended on when the agent met it. Sonnet mentioned the stale test folder in its final report in 17 of its 20 runs with a report that named the function, where it usually met the command while checking its fix, but in none of its 20 runs with a broad report, where it met the command while reproducing the bug. In the Procedure case, only one final report described the old layout. Agents also noticed other outdated parts of `CLAUDE.md` and left them: in 22 of Haiku's 120 runs in the Procedure case, under all three conditions, the final report pointed them out and often said the file was left alone because it was not part of the bug. No report proposed correcting the stale line, and no run edited an instruction file. An agent's workaround leaves a record only in a transcript that maintainers rarely read. Our tasks asked for a code fix, not documentation upkeep, so this is what agents do by default, not what they would do if asked.

Models pay the debt differently. Both models followed the stale lines at similar rates, and both went to the stale Procedure folder in nearly every run whose report named only the area. They split when the report named the function: Haiku then searched by name and used the folder in 4 of 20 runs, Sonnet in 12. The cost also showed differently. Sonnet's runs were short and consistent, so its extra tool calls on the test command could be measured; Haiku's runs varied so much that no difference in effort was detectable, although it followed the stale lines just as often. A cost measured with one model therefore says little about another, so evaluations of instruction files should use more than one model and report what agents did, not only how much effort they spent.

For maintainers: correct the line rather than delete it. Deleting a stale line removed most of its effect, but only correcting it removed all of it. With the line deleted, agents used the stale path in 1 of 80 runs in the Command case and 20 of 80 in the Procedure case, against 78 and 55 with the stale line and none with the corrected line; in the Procedure case, they guessed the old folder from the code's structure (Haiku 13, Sonnet 7 of 40). Deleting also kept the effort cost where the line was needed: on the test command, Sonnet used as many tool calls without the line as with the stale one (7.8 per run in both, against 5.8 with the corrected line; Table II).

Stale paths are also cheap to find: a script can check whether a path exists, although deciding whether a missing path is an error still needs judgment. We did not measure the effort of either repair.

For agent and tool builders: the stale command failed silently. Pytest found no tests and exited with an error, but the agents piped its output through `tail` or `head`, so the shell reported success. In 19 of Sonnet's 40 stale runs, the first check of the fix ran no tests, and Haiku often hid the error itself with `2>/dev/null` (16 of its 38 runs). Four safeguards follow, none of which we evaluated. A harness that loads an instruction file can flag paths in it that do not exist. A shell can set `pipefail`, so that a failing command piped into `tail` still fails, although this does not help when the agent hides the error. An agent can treat "no tests ran" as a failure. And an agent that works around a missing path can say so in its final summary and propose the one-line fix. Unlike a stale convention, a stale path can be detected without deciding whether the file or the code is right.

For researchers: measure actions, and compare against both repairs. Pass rates would have scored all three conditions the same. In four Command tasks, all 15 of Sonnet's runs even produced the same patch, byte for byte, so inspecting the patch would not help either. A trajectory measure that counts any test call as validation would score the stale command, which ran no tests, as a verified fix. Studies of instruction files should therefore report what agents did, such as operations on paths that no longer exist, and compare against both a corrected and a removed version of the instruction. In our study, stale references changed what agents did but not whether they fixed the bug, so evaluations that score only the delivered code would miss this effect.

VI. Threats to Validity

We list the main threats to our conclusions and how we addressed them. The replication reconciliation additionally reports infrastructure failures, contaminated retries, missing human detector validation, dependent model reviews, and the distinction between planned detour and exploratory total-effort analyses. These limitations qualify the results throughout the paper. Statements about unchanged task success mean that the study did not detect a condition difference; they do not establish equivalence. References to a cost measurable only for Sonnet concern the exploratory case-specific total-call outcome, not the planned detour outcome. The experiment does not establish a positive cost for every encounter or every future task.

Number of cases. We chose depth over breadth: each case is a real-world stale line with a documented history, studied under three conditions with the code and tasks held fixed. Because instruction type, repository, and tasks vary together, their separate effects cannot be told apart.

Task difficulty. Most tasks are small seeded defects in the code each stale line describes, admitted only if the repository's own tests detect them. Harder maintenance work may show different costs.

Statistical inference. The 732 runs repeat three cases rather than sampling independent instances of debt, and per-task estimates are imprecise. We therefore use exact tests stratified by task, correct the primary tests for multiplicity, treat an undetected difference as unresolved rather than absent, and label analyses added after the protocol was fixed as exploratory. While checking our numbers, we found that the stale-path detector missed relative paths after a change of directory; we corrected it, which changed 19 Sonnet runs in the Procedure case, and the replication package contains the results before and after the fix.

Generalization. This is a focused experiment: two models from one vendor in one harness, three cases of stale context, and 22 tasks. It shows how stale instructions acted in these settings, but other models, harnesses, kinds of stale context, and tasks may behave differently, so our findings should be read as evidence about these cases rather than as general rates.

VII. Conclusion

A stale line in an agent instruction file is documentation debt whose interest is paid in behavior rather than in outcomes. Across three real-world stale instructions, two models, and 732 runs, a stale test command and a stale procedure sent agents to paths that no longer exist, while a stale code location, which no task needed, had no effect. Agents still fixed the bug in nearly every run, so task success hid the debt: the stale test command checked nothing yet appeared to succeed, and agents worked around the stale path without repairing the file. Correcting the line removed this wasted work, and deleting it removed most of it. Future work should test these findings with more stale instructions, repositories, languages, models, and agent harnesses, measure how often agents reach stale lines in ordinary work, and evaluate the safeguards we propose. Until then, evaluations of instruction files should report what agents do, not only whether they succeed.

AI Assistance Disclosure

OpenAI Codex and Anthropic Claude assisted with manuscript drafting and revision, reference and history searches, data-collection and validation scripts, task generation, the experiment harness, analyses, and figures. Claude and Codex also produced the case-search review labels; those reviews were not independent and were not human validation. Claude Code with the two reported Claude models was the experimental system under study. The authors are responsible for the study decisions and for verifying the text, code, and interpretation. The replication reconciliation describes the limits of validation and historical provenance.

Replication and protocol reconciliation

The dated frozen protocol and its amendments are included in the package; no external preregistration is asserted. The planned effort measure was a detour interval, analyzed on $\log(1 + \text{detour})$. Its stale-versus-corrected results are:

Haiku: coefficient 0.3067, SE 0.0740, Holm-adjusted $p = 6.84002e - 05$.

Sonnet: coefficient 0.1228, SE 0.0697, Holm-adjusted $p = 0.155787$.

For stale, corrected, and removed conditions respectively, the numbers of runs with a defined detour are Haiku: 82, 83, 77; Sonnet: 71, 88, 62. Detour conditions on observed endpoints and is not total execution effort. The case-specific total-call analysis emphasized in the main text is exploratory; it is a different outcome, not a replacement for the planned result. The package records this reporting reconciliation retrospectively and does not claim a contemporaneous rationale for the change of emphasis.

Attempt accounting. In stale, corrected, and removed order, failed attempts were Haiku: 2, 3, 2; Sonnet: 162, 159, 159. Haiku failures were seven timeouts. Sonnet failures included 480 rate-limit rejections across 160 scheduled run IDs, three rejected attempts each, followed by later completion. This exceeded the protocol's three-attempt total limit; the preserved ledger records the deviation. Failed attempts are outside the reported completed-run effort totals. Six smoke runs are also excluded.

Four Haiku retries retained earlier workspace edits, violating the fresh-restart rule. They remain in the headline analysis. The package identifies these runs and reports sensitivity excluding them. Rust linker failures affected both grading and agent execution; all Procedure outcomes use sequential regrades, which cannot remove earlier interference from the trajectories. The package reports signal-9 exposure by condition.

Validation and provenance. The planned blind two-person detector validation was not performed (A4). The later working-directory correction (A5) is a post-outcome measurement correction. Case-search reviews were generated by Claude and Codex; Codex had seen Claudeś labels, so the two model reviews were not independent of each other. After the model reviews, the first author manually reviewed every flagged path, including the three selected cases. Native instruction loading is inferred from the harness and trajectories; it is not a direct observation of the modelś internal use of context.

Reproduction boundary. The package regenerates analyses from recorded transcripts and evaluator outputs. It does not rerun the historical hosted models or independently execute every saved patch. Historical image digests and exact hosted model snapshots were not recorded; current Docker builds are rerun recipes, not verified reconstructions of the original environment. The frozen protocol, failure ledger, detector-version evidence, and claim index remain available as supporting material.

REFERENCES

[1] ZJU-PL, "pyflow," GitHub repository, 2026, commit 0128cfd. [Online]. Available: https://github.com/ZJU-PL/pyflow

[2] Anonymous, "Replication package for "Stale but Passing: What Task Success Hides About Documentation Debt in Coding Agents"," 2026, local replication package supplied alongside this manuscript; archival publication pending.

[3] T. Gloaguen, N. Mündler-Sasahara, M. N. Müller, V. Raychev, and M. Vechev, "Evaluating AGENTS.md: Are repository-level context files helpful for coding agents?" arXiv:2602.11988v3, 2026. [Online]. Available: https://arxiv.org/abs/2602.11988v3

[4] J. L. Lulla, S. Mohsenimofidi, M. Galster, J. M. Zhang, S. Baltes, and C. Treude, "On the impact of AGENTS.md files on the efficiency of AI coding agents," arXiv:2601.20404, 2026. [Online]. Available: https://arxiv.org/abs/2601.20404

[5] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 1199–1210.

[6] N. Rios, L. Mendes, C. Cerdeiral, A. P. F. Magalhães, B. Perez, D. Correal, H. Astudillo, C. Seaman, C. Izurieta, G. Santos, and R. O. Spínola, "Hearing the voice of software practitioners on causes, effects, and practices to deal with documentation debt," in *Requirements Engineering: Foundation for Software Quality (REFSQ)*, ser. Lecture Notes in Computer Science, 2020, pp. 55–70.

[7] S. Mohsenimofidi, M. Galster, C. Treude, and S. Baltes, "Context engineering for AI agents in open-source software," in *Proceedings of the 23rd IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2026, arXiv:2510.21413.

[8] W. Chatlatanagulchai, H. Li, Y. Kashiwa, B. Reid, K. Thonglek, P. Leelaprute, A. Rungsawang, B. Manaskasemsak, B. Adams, A. E. Hassan, and H. Iida, "Agent READMEs: An empirical study of context files for agentic coding," arXiv:2511.12884, 2025. [Online]. Available: https://arxiv.org/abs/2511.12884

[9] C. Treude and S. Baltes, "Context rot in AI-assisted software development: Repurposing documentation consistency for AI configuration artifacts," arXiv:2606.09090v1, 2026. [Online]. Available: https://arxiv.org/abs/2606.09090v1

[10] X. Xu and W. Wu, "Stale instruction files override the code in coding agents: A one-sentence fix," arXiv:2610.30571v1, 2026. [Online]. Available: https://arxiv.org/abs/2610.30571v1

[11] B. Mohammadi, L. Klein, A. Chadha, A. Arora, and L. Bindschaedler, "The working set of a coding agent: Coherence debt in repository-scale tasks," arXiv:2608.16630v1, 2026. [Online]. Available: https://arxiv.org/abs/2608.16630v1

[12] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[13] Replicate, "Cog," GitHub repository, 2026, commit ee8ae12. [Online]. Available: https://github.com/replicate/cog

[14] AkaraChen, "aghub," GitHub repository, 2026, commit 2a846f2. [Online]. Available: https://github.com/AkaraChen/aghub

[15] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, 1992, pp. 29–30.

[16] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[17] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016.

[18] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 427–435.

[19] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li, "Just-in-time obsolete comment detection and update," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 1–23, 2023.

[20] W. S. Tan, M. Wagner, and C. Treude, "Detecting outdated code element references in software repository documentation," *Empirical Software Engineering*, vol. 29, 2024.

[21] M. Galster, S. Mohsenimofidi, J. L. Lulla, M. A. Abubakar, C. Treude, and S. Baltes, "Harness engineering for agentic AI coding tools: An exploratory study," in *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware)*, 2026, arXiv:2602.14690.

[22] W. Chatlatanagulchai, K. Thonglek, B. Reid, Y. Kashiwa, P. Leelaprute, A. Rungsawang, B. Manaskasemsak, and H. Iida, "On the use of agentic coding manifests: An empirical study of Claude Code," in *Proceedings of the 26th International Conference on Product-Focused Software Process Improvement (PROFES)*, 2025.

[23] A. Shepard and J. Albrecht, "Probe-and-refine tuning of repository guidance for coding agents," arXiv:2606.20512v2, 2026. [Online]. Available: https://arxiv.org/abs/2606.20512v2

[24] Y. Wang, M. Pradel, and Z. Liu, "Are "solved issues" in SWE-bench really solved correctly? an empirical study," in *Proceedings of the 48th*

IEEE/ACM International Conference on Software Engineering (ICSE), 2026, pp. 169–181.

[25] B. Yu, Y. Cao, Y. Zhang, L. Lin, J. Xu, Z. Zhong, Q. Xu, G. Wang, J. Cao, S.-C. Cheung, P. He, and L. Briand, "SWE-ABS: Adversarial benchmark strengthening exposes inflated success rates on test-based benchmark," arXiv:2603.00520, 2026. [Online]. Available: https://arxiv.org/abs/2603.00520

[26] C. Li, Y. Xu, Z. Wang, S. H. Tan, and T.-H. P. Chen, "Probe to generate: Program variant-guided test augmentation for repository-level repair benchmarks," in *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026, arXiv:2604.01518.

[27] I. Bouzenia and M. Pradel, "Understanding software engineering agents: A study of thought-action-result trajectories," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, arXiv:2506.18824.

[28] T. Mehtiyev and W. Assunção, "Beyond resolution rates: Behavioral drivers of coding agent success and failure," arXiv:2604.02547, 2026. [Online]. Available: https://arxiv.org/abs/2604.02547

[29] X. Zhao, H. Li, S. Li, T. Zhao, E. T. Barr, F. Sarro, and H. Ye, "Failure as a process: An anatomy of CLI coding agent trajectories," 2026.

[30] P. Sahoo, G. Mittal, X. Li, S. Ma, B. Steenhoek, P. Lin, and Y. Hu, "Agentlens: Revealing the lucky pass problem in swe-agent evaluation," 2026. [Online]. Available: https://arxiv.org/abs/2605.12925

[31] R. Shu, C. Y. Chong, X. Zhou, Y. Peng, Z. Wu, X. Han, Z. Zhuang, G. Yuan, and Y. Wang, "What resolve rate hides: Trajectory structure diagnostics for coding agents," 2026. [Online]. Available: https://arxiv.org/abs/2607.06184

[32] Z. L. Liu, S. Pandit, X. Ye, E. Choi, and G. Durrett, "CodeUpdateArena: Benchmarking knowledge editing on API updates," arXiv:2407.06249v3, 2025, first posted in 2024. [Online]. Available: https://arxiv.org/abs/2407.06249v3

[33] J. Shi, Y. Chen, J. He, Y. Liu, and D. Lo, "Aletheia: Permission-minimality testing for coding-agent rules," arXiv:2609.39678v2, 2026. [Online]. Available: https://arxiv.org/abs/2609.39678v2