Coding Agents Don't Know When to Act

Anonymous Authors[1]

Abstract

Coding agents are increasingly deployed to autonomously maintain software, including to resolve user-reported issues: a bug report comes in and the agent creates a patch to address it. However, in any real-world deployment, they will encounter stale bug reports about issues that have already been resolved. Agents should recognize this and abstain from modifying the code to avoid accumulating technical debt. To systematically evaluate whether current agents do so, we introduce FIXEDBENCH, a code benchmark with 200 human-verified coding tasks in which no code changes are required, testing five recent models across four agent harnesses. We find that even state-of-the-art models fail, proposing undesirable changes (excluding tests and documentation) in 35 to 65% of cases. Explicit instructions to reproduce the issue before patching partially address this issue but introduce a new failure mode: when an issue is partially fixed, they abstain even though a patch would still be needed. More broadly, our results indicate that LLMs fall prey to an action bias: they choose to act even if inaction would be appropriate. To break this pattern, inaction needs to be explicitly framed as a path to success, which highlights an overreliance on human guidance implicit in current training objectives.

1. Introduction

Coding agents are increasingly deployed for autonomous software maintenance. In production settings, teams now routinely delegate issue resolution, code review, security scanning, and code editing to agents that operate with minimal supervision (Anthropic, 2025; OpenAI, 2025; Google, 2025; Wang et al., 2025), and sometimes under full automation (Anysphere, Inc., 2026). A typical maintenance workflow involves an agent that receives a bug report, investigates the issue, writes a patch, and then submits it under full automation without any human review. As this level of delegation transitions from individual tasks to continuous, event-driven pipelines, the quality of agent judgment becomes critical for long-term software maintenance.

A key aspect of this judgment involves knowing when *not* to act. In any real codebase, agents routinely encounter outdated bug reports, issues already fixed in a parallel pull request, or tickets that reference behavior that has since been patched. Such duplicate reports can make up to 49% of all bug reports (Cavalcanti et al., 2013). A competent agent should recognize such situations, confirm that no code change is required, and move on without introducing technical debt. This ability to abstain from undesirable changes is a prerequisite for autonomous maintenance, yet it has never been evaluated systematically.

Assessing Model Abstention Rates. This work directly measures whether current coding agents can recognize when the code is already correct and abstain from making any meaningful changes. To this end, we introduce FIXEDBENCH, a random subset of 200 SWE-Bench Verified instances (Jimenez et al., 2024) in which the changes resolving the issue are already applied. The expected behavior is simple: submit an empty patch, editing at most tests and documentation. We evaluate five recent models across four agent harnesses and conduct thorough ablations to investigate failure modes and remediation strategies.

Models Fail to Abstain. Our results show that even frontier agents and models fall prey to an action bias, incorrectly applying undesirable changes in $35 - 65\%$ of instances. Manual trace analysis reveals two key factors: (i) models often never attempt to reproduce the reported bug before patching it, and (ii) even after realizing the issue is already fixed, the agent feels compelled to modify the code.

Prompting Bias. System prompts have a strong effect on this failure mode. Specifically, instructing the agent to edit the codebase exacerbates the problem, for example, almost halving the abstention rate of GPT-5.4 MINI from 60.5% to 36.5%. In contrast, instructing it to first reproduce the

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

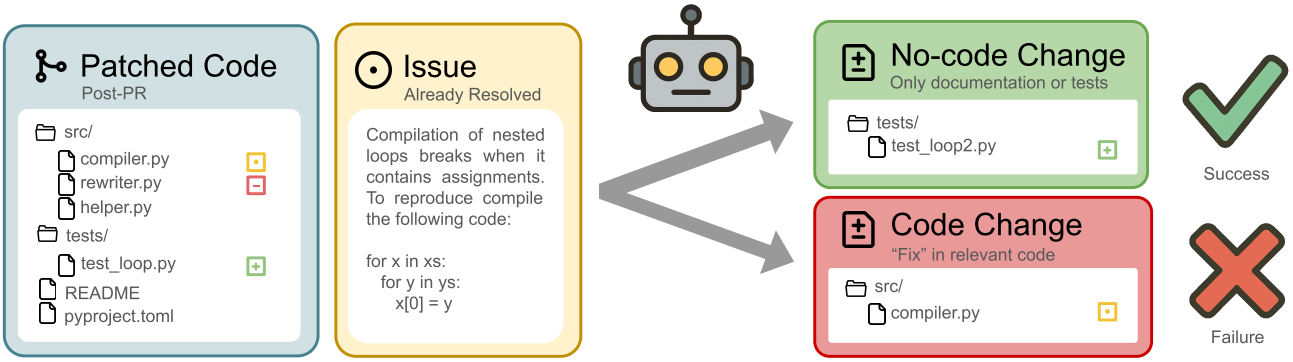

Figure 1. Overview of the main FIXEDBENCH task (RESOLVED): The agent is tasked to resolve the user issue after it has already been resolved. Any (unnecessary) change to executable code is counted as failure.

problem and to abstain if it is already resolved alleviates the issue, for example, increasing the abstention rate for GPT-5.4 MINI to 88.5%. Interestingly, prompting the model only to reproduce the problem, without explicitly framing abstention as a successful outcome, has no effect or even an adverse effect, for example, reducing the abstention rate for GPT-5.4 MINI to 47.5%. However, this prompting strategy is brittle: when the issue is only partially or incorrectly resolved, the same prompt causes models to over-abstain and, as a result, fail to fix code that genuinely needs repair.

Action Bias. More broadly, our findings point to a gap between how agents are trained and what autonomous maintenance may require. Current models are trained to act, e.g., , to produce patches, rather than to decide whether action is required. This behavior likely stems from the reinforcement learning post-training procedure of models, which is dominated by tasks that require a change (Badertdinov et al., 2025; Mohamadi et al., 2025; Cambronero et al., 2025). However, we argue that if agents that process thousands of changes per week systematically fail to recognize when code is already correct, the result is a steady accumulation of undesirable modifications to the codebase and increased technical debt.

Contributions. The core contributions of our paper are as follows:

- A framework for generating benchmarks evaluating whether coding agents recognize that code is already correct, applied to SWE-BENCH VERIFIED.
- A thorough evaluation of multiple models and agent harnesses showing that none reliably abstain from making undesirable changes across a range of settings.
- An analysis of prompt-based remediation, showing that a verify-then-abstain prompt substantially improves abstention rates but introduces a complementary failure mode of over-abstention in the presence of incorrect prior patches.
- A large-scale trace analysis, identifying that agents feel compelled to edit the code even after realizing the issue is already resolved, suggesting that training objectives should reward verification, not just patch generation.

2. Benchmark

In this section, we outline the structure of our benchmark FIXEDBENCH and how we leverage it to assess the capabilities of coding agents to determine when a given issue is already resolved versus when it needs further work.

2.1. Notation and Definitions

We first introduce the notation to describe codebases, their test suites, and changes to these codebases in the form of patches. A codebase R is a set of files f. A patch X is a set of tuples (f, c) of a file and a changed, inserted, or removed line of content. $|X|$ counts the number of lines touched by a patch. A patch can be applied to a codebase. We denote a codebase R after applying patch X as $R \circ X$. Several patches can be applied sequentially, i.e. $R \circ X \circ Y$ is the codebase R after applying a first patch X and then a second one Y. A set of tests T can be used to check the correctness of a given codebase.

A single test s can either pass (P) or fail (F) after we execute it within the context of codebase R. If an error is thrown during a test's execution (either due to unexpected failure or explicit failed assertions), we consider it to have failed. Conversely, if the test executes without raising an error, we consider it to have passed. We define this process as an execution function: $\text{exec}(s, R) \in \{P, F\}$.

2.2. Benchmark Overview

Our evaluation is based on human-validated, real-world GitHub issues, in the style of SWE-Bench (Jimenez et al.,

2024). For ease of overview, we briefly outline the construction process of SWE-BENCH VERIFIED (Chowdhury et al., 2024). The construction consists of four key steps.

1. Scrape pull requests (PRs) from open-source Python repositories on GitHub.
2. Filter PRs to only include those that were merged, resolved a GitHub issue, and made changes to at least one test file.
3. Filter PRs to feature at least one test that fails in the pre-PR state R and passes in the merged state $R \circ X^*$, removing PRs that result in installation/runtime errors.
4. Filter instances with underspecified issues and overly restrictive test cases using manual annotation by human software engineers.

This results in task instances consisting of a GitHub issue I, a pre-patch repository state R, a golden patch X^* fixing the issue, and a set of golden reference tests T^*.

2.3. Tasks

We consider three tasks that each represent a stage in software development: NOVEL: resolve a novel issue, PARTIAL: resolve an issue that was already partially or incorrectly addressed, and RESOLVED: process a stale issue that was already resolved.

Concretely, we initialize the coding agent in the following repository states for each task, given the pre-patch repository state R, a partial patch X^P, and the golden patch X^*.

- $R_{\text{NOVEL}} = R$: The standard pre-patch repository state is used. This corresponds to the setup of standard SWE-Bench-like benchmarks. The golden test suite fails.
- $R_{\text{PARTIAL}} = R \circ X^P$: An incorrect or insufficient patch generated by a weaker agent is applied to the pre-patch codebase. The golden test suite fails.
- $R_{\text{RESOLVED}} = R \circ X^*$: In the RESOLVED setting, the golden patch is applied to the pre-patch codebase. The golden test suite passes.

2.4. Scenarios

For all tasks, we consider two settings: a best-case scenario (BEST), and a worst-case scenario (WORST):

BEST In the best-case scenario, the Git history is available with the last commit resolving the issue, and the agent operates in a fully set up execution environment (e.g., environment variables, packages). There is no friction in running the repository test suite, and there are clear indications whether the bug is already fixed. The BEST scenario represents a realistic setting where the agent is part of a deployment pipeline, for example, being automatically provided with submitted GitHub issues. In Appendix C we also consider a best-case scenario in which the repository already contains a correct test file for the fixed issue.

WORST In the WORST scenario, the `.git` folder is removed and the environment is not set up, i.e., the agent needs to install the required packages on its own if it wants to run tests. It allows us to assess the agent's performance under adverse conditions.

2.5. Metrics

In order to successfully maintain code over long horizons, coding agents should produce patches $\hat{X}$ that are both minimal and correct. We introduce two metrics to assess this:

Resolution Rate measures whether the produced patch resolves the issue. Concretely, we apply the patch $\hat{X}$ to the base repository of the task R_X and execute the golden reference tests T^*. The task resolution metric PASS is 1 exactly when all golden reference tests pass, i.e., $\text{PASS} = \text{exec}(T^*, R_X \circ \hat{X}) = P$. The *Resolution Rate* is its average across instances.

Abstention measures whether a submitted patch makes no meaningful changes to the codebase. We ignore changes that modify comments, documentation, or test cases and provide more detail on our filtering in Appendix A. The metric ABSTENTION is 1 exactly when the predicted patch (net comments and tests) is empty, i.e., $|\hat{X}| = 0$. We then compute the *Abstention Rate* as its average. Depending on the task, we clarify the desired behavior by denoting it the *Correct* (RESOLVED) or *Incorrect* (PARTIAL, NOVEL) abstention rate.

3. Experimental Evaluation

3.1. Setup

Datasets While our benchmark generation approach can be applied to arbitrary SWE-Bench-like tasks, we construct FIXEDBENCH from a random subset of 200 SWE-BENCH VERIFIED (Chowdhury et al., 2024) instances. Unless otherwise mentioned, we evaluate the RESOLVED setting, where we automatically commit the golden patch from the PR such that the issue described in the problem statement is resolved.

Metrics We measure the *Resolution Rate*, the percentage of correctly resolved issues, and the *Abstention Rate*, the percentage of patches without undesirable edits, for all instances, as described in §2.5. All reported error bars show

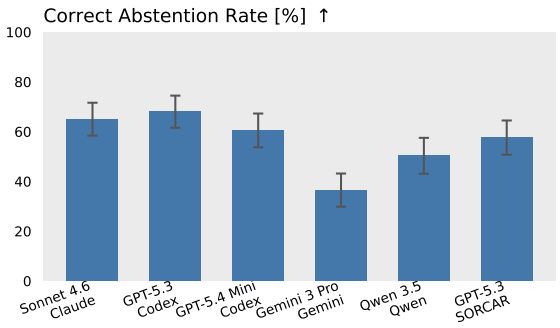

Figure 2. Correct abstention rate in the BEST scenario for the ISSUE prompt on the RESOLVED task.

95% confidence intervals. For statistical tests, we compute p-values with McNemar's test on the Abstention Rate.

Agent Instructions We prompt the agent with four different instructions:

1. ISSUE: The instance description without additional information. This serves as the default baseline prompt.
2. EDIT: A prompt tasking the agent to edit the codebase to address the issue. This is an adverse condition reinforcing the action bias.
3. REPRODUCE: First, reproduce the issue and then address it.
4. ABSTAIN OR FIX: First, reproduce the issue, and then either address the issue or abstain if there is nothing to fix.

For the exact prompt templates, see Appendix E.

Models For our main results, we use 4 closed models and 1 open-source model, SONNET-4.6, GPT-5.3 CODEX (with xhigh thinking), GPT-5.4 MINI (with xhigh thinking), GEMINI-3 PRO, and QWEN3.5-122B, with their corresponding agentic harnesses (i.e., Claude-code for SONNET-4.6, Codex for GPT-5.4 MINI and GPT-5.3 CODEX, Gemini-CLI for GEMINI-3 PRO, and Qwen-Code for QWEN3.5-122B). We also use the SORCAR harness with GPT-5.3 CODEX to evaluate open-source agentic frameworks. For our ablations (e.g., system prompt, scenario, and PARTIAL), we use SONNET-4.6 and GPT-5.4 MINI only for cost reasons. We defer details to Appendix B.

3.2. Main Results

Agents Fall Victim to Action Bias Even under favorable conditions, i.e., with access to the full git history, a working execution environment, and the ISSUE prompt, coding agents struggle to recognize when code is already correct. Across models and harnesses, agents apply unnecessary edits to already-fixed code in 35 to 65% of cases (see Figure 2 and Table 1). Even the best-performing models SONNET-4.6, GPT-5.4 MINI and GPT-5.3 CODEX do not achieve correct abstention rates above 67%, meaning that at least one in three stale issue reports results in spurious edits to working code.

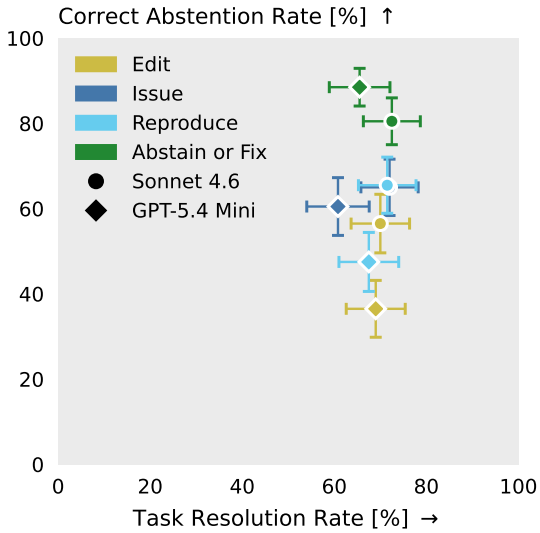

Figure 3. Correlation of correct-abstention rate and SWE-Bench task resolution across models and prompts.

Better Instructions Help We consider four different prompts to assess the impact of the agent's instructions on this failure mode and find a strong effect, visualized in Figure 3 and detailed in Table 1.

Explicitly instructing the agent to edit the codebase to fix the issue (EDIT) significantly exacerbates the action bias, reducing the (correct) abstention rate to 56.5% and 36.5% for SONNET-4.6 and GPT-5.4 MINI, respectively, down from 65.0% and 60.5%. In contrast, prompting the agents to first verify the issue is still present and abstain if it is not (ABSTAIN OR FIX) improves correct-abstention rates significantly to 80.5% and 88.5%, an uplift of 15.5 ($p = 9.5 \times 10^{-3}$) and 28.0 ($p = 2.3 \times 10^{-8}$) percentage points over their respective ISSUE baselines. The SORCAR run with GPT-5.3 CODEX shows a similar improvement, from $57.6\%^{\pm 6.9\%}$ under ISSUE to $83.5\%^{\pm 5.1\%}$ under ABSTAIN OR FIX.

Interestingly, prompting for reproduction *without* the explicit option to abstain is insufficient. The REPRODUCE variant yields only a negligible change for SONNET-4.6 (65.0% → 65.5%, $p = 1.0$) and even a notable degradation for GPT-5.4 MINI (60.5% → 47.5%, $p = 6.5 \times 10^{-5}$),

Table 1. The abstention rate (Abs.) and SWE-bench resolution rate (Res.) across tasks, scenarios, and prompts for selected models. The accuracy columns use the same raw SWE-bench accuracy for the corresponding prompt. All values are means ± 95% binomial CIs.

Task	Scenario	Prompt	SONNET-4.6		GPT-5.4 MINI	
			Abs. [%]	Res. [%]	Abs. [%]	Res. [%]
RESOLVED	BEST	ISSUE	$65.0^{\pm 6.6}$	$72.0^{\pm 6.2}$	$60.5^{\pm 6.8}$	$60.8^{\pm 6.8}$
		EDIT	$56.5^{\pm 6.9}$	$70.0^{\pm 6.4}$	$36.5^{\pm 6.7}$	$69.0^{\pm 6.4}$
		REPRODUCE	$65.5^{\pm 6.6}$	$71.5^{\pm 6.3}$	$47.5^{\pm 6.9}$	$67.5^{\pm 6.5}$
		ABSTAIN OR FIX	$80.5^{\pm 5.5}$	$72.5^{\pm 6.2}$	$88.5^{\pm 4.4}$	$65.5^{\pm 6.6}$
	WORST	ISSUE	$50.0^{\pm 7.0}$	$72.0^{\pm 6.2}$	$52.5^{\pm 6.9}$	$60.8^{\pm 6.8}$
		EDIT	$42.5^{\pm 7.3}$	$70.0^{\pm 6.4}$	$30.5^{\pm 6.4}$	$69.0^{\pm 6.4}$
		ABSTAIN OR FIX	$72.9^{\pm 6.2}$	$72.5^{\pm 6.2}$	$85.0^{\pm 4.9}$	$65.5^{\pm 6.6}$
PARTIAL	BEST	ISSUE	$70.7^{\pm 7.3}$	$8.7^{\pm 4.5}$	$77.3^{\pm 6.7}$	$5.3^{\pm 3.6}$
		ABSTAIN OR FIX	$81.3^{\pm 6.2}$	$6.0^{\pm 3.8}$	$93.6^{\pm 4.3}$	$2.9^{\pm 2.8}$

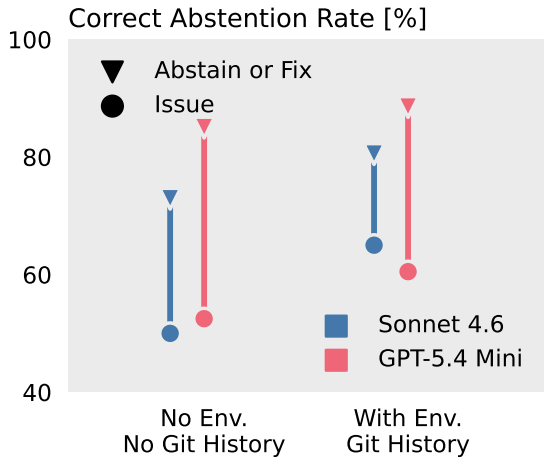

Figure 4. Correct abstention rate under ISSUE and ABSTAIN OR FIX prompts across the BEST and WORST scenarios.

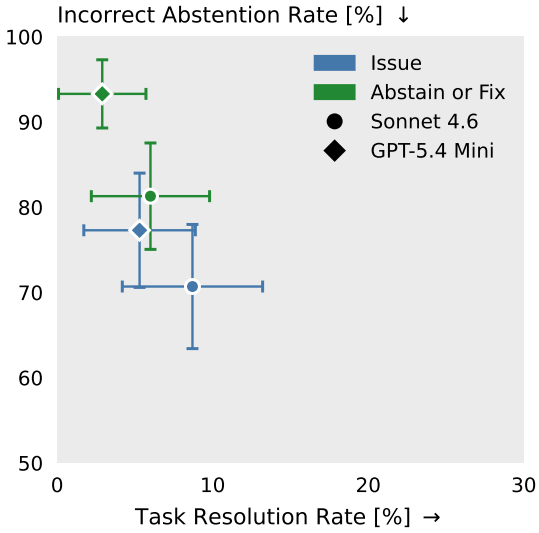

Figure 5. Incorrect abstention rate (higher is worse) vs. Resolution Rate on the PARTIAL task, where a partial fix is applied and the agent should fix rather than abstain.

suggesting that reproduction steps alone do not help agents decide to hold back. Framing abstention as a successful outcome is what drives the improvement.

Importantly, these prompts do not compromise the agents' ability to resolve genuine bugs. Across all four variants, SWE-Bench Verified resolution rates for both models remain within each other's confidence intervals, even improving performance in some cases (Figure 3).

Adverse conditions. So far we have considered a best-case scenario, where agents had access to the git history, with the most recent commit addressing the issue at hand, and an already set up environment. We now evaluate the effect of removing both the git history and the environment, visualizing results in Figure 4 with details in Table 1. We observe that without an easy way to confirm task resolution, the correct abstention rate decreases from 65.0% to 50.0% and from 60.5% to 52.5% for SONNET-4.6 and GPT-5.4 MINI, respectively. However, the ABSTAIN OR FIX prompt remains effective, improving the non-edit rate from 50.0% to 72.9% ($p = 1.1 \times 10^{-8}$) and from 52.5% to 85.0% ($p = 5.5 \times 10^{-25}$) for SONNET-4.6 and GPT-5.4 MINI, respectively. This almost matches the abstention rate in the BEST scenario, indicating that agents are capable of performing the necessary verification even under unfavorable conditions, provided they are explicitly instructed to do so.

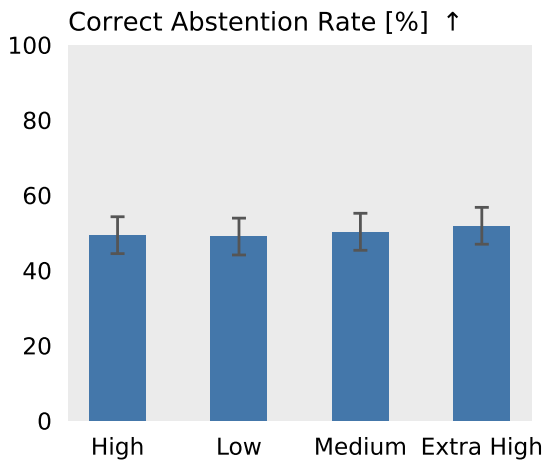

Figure 6. Abstention rate of GPT-5.4 MINI across thinking efforts.

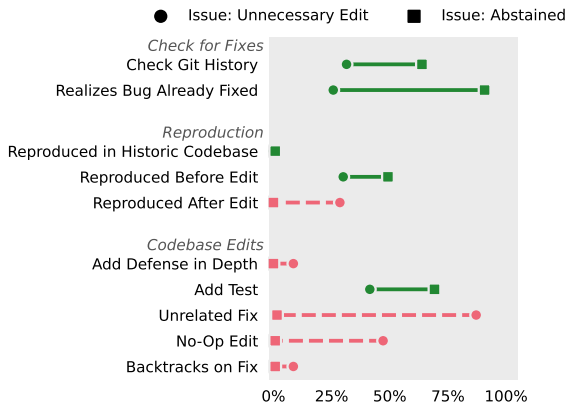

Figure 7. Comparison between traces leading to abstention vs. unnecessary edits in the best-case scenario. Green solid lines indicate higher frequency in the successful group; red dashed lines the reverse.

The Problem with Instruction Changes. While our ABSTAIN OR FIX instruction reduces unnecessary edits on already-fixed code without reducing the success rate on unmodified code, it introduces a symmetric failure mode: over-abstention on incorrect code that genuinely requires fixing. We measure this with the PARTIAL task on 150 instances where an incorrect patch (generated with GPT-5 NANO) is applied and the agent *should* modify the codebase (see Figure 5 with details in Table 1). We report the *incorrect abstention rate*, the fraction of instances where the agent incorrectly abstains from making changes, as the primary metric; higher values indicate worse performance.

Under the baseline ISSUE prompt, agents already exhibit a troubling degree of incorrect abstention: SONNET-4.6 and GPT-5.4 MINI only make any meaningful edit in $29.3\%^{\pm 7.3\%}$ and $23.7\%^{\pm 6.7\%}$ of cases, resolving the issue only in $8.7\%^{\pm 4.5\%}$ and $5.3\%^{\pm 3.6\%}$ of cases, respectively. This shows a concerning lack of ability to differentiate working from broken code, as resolution rates on the unmodified versions of the same subset of instances are much higher at $27.3\%^{\pm 7.9\%}$ and $24.4\%^{\pm 6.4\%}$, respectively. Critically, the ABSTAIN OR FIX prompt, which most effectively suppresses unnecessary edits on fixed code, *increases* this incorrect abstention rate even further such that edits are only made in $18.7\%^{\pm 6.2\%}$ and $6.4\%^{\pm 4.3\%}$ of cases for SONNET-4.6 and GPT-5.4 MINI.

Prompt engineering thus trades one failure mode for another: it can suppress action bias on fixed code at the cost of inducing passivity on broken code.

The (Lacking) Effect of Reasoning Effort. To investigate whether more reasoning helps agents avoid unnecessary edits, we vary the thinking effort of GPT-5.4 MINI across four levels under the ISSUE prompt, finding no significant effect (see Figure 6). While correct-abstention rates increase from 61.5% to 63.3%, 64.3%, and 65.8% for Low, Medium, High, and Extra High effort, respectively, this change (Spearman rank correlation of $r = 1.0$ at $p = 0.083$) is notably smaller than the confidence intervals of $\sim \pm 7.0\%$ and the uplift resulting from a change of prompt. This suggests that action bias is not a failure of deliberation that more compute can overcome, but rather a failure of task framing: agents need to be told that abstaining is a valid outcome and not just told to think harder. This also suggests that they have not been trained to discern cases where the instructions (here fixing a non-existent bug) are wrong.

3.3. Case Studies

In this subsection, we analyze the traces of SONNET-4.6 across various settings.

Experimental Design We manually inspected 50 traces to define 10 (non-exclusive) behavioral categories, each corresponding to a relevant action of the agent (e.g., looking at the git history or implementing a fix). See Appendix D for detailed descriptions of all categories. We then use GPT-5.4 as a judge to identify (potentially multiple) behaviors in each trace. These traces consist of the provided tool calls and outputs, but not the thinking traces, as we found these to mislead the judge in a manual inspection. We find high agreement between the human and LLM judge at 80.5% precision and 75.0% recall. For a detailed analysis, see Appendix C.

Why Agents (Don't) Abstain We examine what separates traces leading to correct abstention from those leading

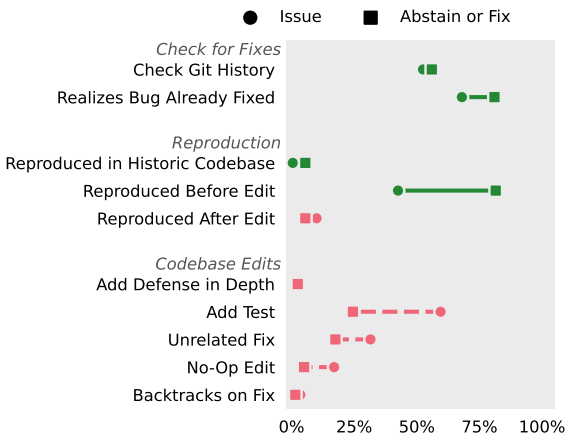

Figure 8. Comparison between the ISSUE and the ABSTAIN OR FIX prompt. Green solid lines indicate behaviors more frequent in the ABSTAIN OR FIX group; red dashed lines indicate the reverse.

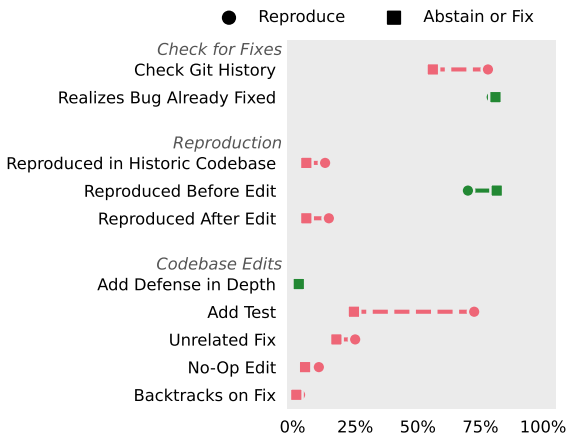

Figure 9. Comparison between the REPRODUCE and the ABSTAIN OR FIX prompt. Green solid lines indicate behaviors more frequent in the ABSTAIN OR FIX group; red dashed lines indicate the reverse.

to unnecessary edits under the ISSUE prompt in the BEST scenario in Figure 7. We observe the two strongest predictors of correct abstention to be checking the git history (63.8% vs. 31.4%) and attempting to reproduce the issue before making any edits (49.2% vs. 30.0%): agents that pause to gather evidence are far more likely to recognize that the code is already correct (90.8 % vs 25.7%).

Interestingly, agents that correctly abstained more frequently *added a test* (69.2% vs. 41.4%), suggesting that they channel their action bias into verification rather than modification. In contrast, unsuccessful traces are overwhelmingly characterized by agents modifying code unrelated to the issue they were supposed to resolve (87.1%) or applying changes that do not meaningfully change behavior (47.1%).

Why Prompting Helps Visualizing the behavioral differences between the ISSUE, REPRODUCE, and ABSTAIN OR FIX prompts in Figures 8 and 9, we observe the most significant difference to be whether the agent attempts to reproduce the issue before making any edits (81.5% (ABSTAIN OR FIX) and 70.0% (REPRODUCE) vs 42.5% ISSUE), leading the agent to realize the issue is already fixed far more frequently (81.0% and 79.5% vs 68.0%). However, realizing a bug is fixed alone does not prevent the agent from making undesirable edits, e.g., ABSTAIN OR FIX and REPRODUCE realize this at almost the same rate of 81.0% and 79.5%, respectively, while the correct abstention rate varies significantly (80.5% vs 65.5%). Instead, the framing of abstention as successful task resolution drives a significant reduction in all forms of edits, both targeted fixes and broader refactors. This suggests the key issue is not the agent's ability to reproduce an issue and, as a result, realize it is already resolved, but what it believes its success criteria are.

4. Related Work

Our work connects several active research areas: software engineering benchmarks, LLM-based coding agents, and LLM topics like sycophancy and hallucinations.

Code generation benchmarks Early code generation benchmarks such as HumanEval (Chen et al., 2021) and MBPP (Odena et al., 2021) focus on isolated function synthesis. SWE-BENCH VERIFIED (Jimenez et al., 2024) moved to software engineering tasks that edit an existing codebase, drawing tasks from real GitHub issues and pull requests. More recent benchmarks (Badertdinov et al., 2025; Vergopoulos et al., 2025; Gloaguen et al., 2026) create tasks from more diverse, niche repositories where data contamination is less likely. Our work directly builds on SWE-BENCH VERIFIED, but *inverts* the standard evaluation setup: rather than presenting a codebase with an unresolved bug, we present one where a fix is already applied and measure whether agents recognize that the fix is correct and abstain.

Code generation benchmarks are fragile along multiple axes such as language and repository distribution, data contamination, and test adequacy. Riddell et al. (2024) found substantial overlap between HumanEval/MBPP and open pretraining corpora. These concerns motivated temporally-aware benchmarks: LiveCodeBench (Jain et al., 2024) collects post-training-cutoff problems, while SWE-rebench (Badertdinov et al., 2025) provides direct evidence of contamination, with a nearly 20 percentage point performance gap between SWE-BENCH VERIFIED

and temporally-filtered tasks. Our work still focuses only on Python, but completely sidesteps data contamination: even if a model has memorized the correct patch, the correct response in our setup is to produce *no patch*, since the fix is already applied.

Automated program repair and test quality Bug-fixing software engineering tasks were previously known in the literature as automated program repair (APR) (Zhang et al., 2023; Jiang et al., 2026), and were often defined as fixing buggy code given a failing test suite. However, even outside of LLM-based approaches, a central concern in APR is *patch overfitting*, i.e. generating patches that pass the tests but are semantically incorrect. Wang et al. (2021) found that most APR-generated patches overfit on some benchmarks, and Mündler et al. (2024) showed that agent-generated fail-to-pass tests can substantially increase patch precision by serving as effective filters. SWE-bench+ (Aleithan et al., 2024) revealed a related issue: many SWE-bench instances were incorrectly marked passing due to weak test cases. Our benchmark requires LLMs to perform reasoning on whether a repair is correct and sufficient, a task strongly related to validating program repairs.

Abstention and sycophancy Sharma et al. (2023) demonstrated that RLHF-trained assistants systematically exhibit sycophantic behavior, preferring responses that match user beliefs over truthful ones. Code agents exhibit what might be termed *task-level sycophancy*: they comply with the implicit premise of the bug report rather than questioning whether that premise holds in the codebase. The abstention literature provides frameworks (Feng et al., 2024) for deciding whether to act based on the knowledge ingested by the model. As part of such a broad benchmark, Kirichenko et al. (2025) found that strategic prompting can enhance abstention ability, but model reasoning does not improve abstention rates. These broader benchmarks mirror our findings, but we quantify these effects in the context of code reasoning and bug-fixing tasks.

5. Limitations and Future Work

We highlight three limitations of our work that present opportunities for future work: First, our evaluation focuses on Python code from popular repositories. Future work may investigate repositories of different popularity (Gloaguen et al., 2026) and languages (Yang et al., 2025). Second, our benchmark does not capture the complexity of necessary patches. Third, it remains open how to address the action bias failure mode without exacerbating the over-abstention failure mode in the presence of partial fixes.

6. Conclusion

We introduced FIXEDBENCH, a benchmark measuring whether coding agents recognize when an issue is already resolved and does not require any edits. Across recent models and agent harnesses, we find this capability to be lacking: even strong agents frequently modify already-correct code, revealing an action bias that is a critical failure mode in autonomous software maintenance, yet not captured by current coding benchmarks. While we find that agent instructions that explicitly highlight the possibility to abstain can mitigate this failure mode, this comes at the cost of over-abstention on partially fixed code that still requires repair.

More generally, we argue that current agents are overreliant on the exact task framing and do not apply "common sense" by default, a gap that is critical for higher-autonomy applications. We hope that FIXEDBENCH raises awareness of this issue and helps shift evaluation and training toward this more realistic setting.

Reproducibility Statement

To ensure reproducibility of our results, we detail all our hyperparameters and additional configuration details in §3.1 and Appendix B. We will release the code used to conduct the presented experiments. In the code, we pin requirements and evaluated LLM versions and coding agent harnesses. We report all used prompts in Appendix E. To ensure replicability, we focus on significant results and report all outcomes with a confidence interval at 95%.

References

Aleithan, R., Xue, H., Mohajer, M. M., Nnorom, E., Uddin, G., and Wang, S. Swe-bench+: Enhanced coding benchmark for llms, 2024. URL https://arxiv.org/abs/2410.06992.

Anthropic. Claude Code overview, 2025. URL https://code.claude.com/docs/en/overview.

Anysphere, Inc. Cursor Automations, 2026. URL https://cursor.com/blog/automations.

Badertdinov, I., Golubev, A., Nekrashevich, M., Shevtsov, A., Karasik, S., Andriushchenko, A., Trofimova, M., Litvintseva, D., and Yangel, B. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2505.20411.

Cambronero, J., Tufano, M., Shi, S., Wei, R., Uy, G., Cheng, R., Liu, C.-J., Pan, S., Chandra, S., and Rondon, P. Abstain and validate: A dual-llm policy for re-

ducing noise in agentic program repair. *arXiv preprint arXiv:2510.03217*, 2025.

Cavalcanti, Y. C., da Mota Silveira Neto, P. A., Lucrédio, D., Vale, T., de Almeida, E. S., and de Lemos Meira, S. R. The bug report duplication problem: an exploratory study. *Softw. Qual. J.*, (1), 2013.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. 2021.

Chowdhury, N., Aung, J., Shern, C. J., Jaffe, O., Sherburn, D., Starace, G., Mays, E., Dias, R., Aljubeh, M., Glaese, M., Jimenez, C. E., Yang, J., Ho, L., Patwardhan, T., Liu, K., and Madry, A. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Feng, S., Shi, W., Wang, Y., Ding, W., Balachandran, V., and Tsvetkov, Y. Don't hallucinate, abstain: Identifying llm knowledge gaps via multi-llm collaboration, 2024. URL https://arxiv.org/abs/2402.00367.

Gloaguen, T., Mündler, N., Müller, M. N., Raychev, V., and Vechev, M. T. Evaluating agents.md: Are repository-level context files helpful for coding agents? *CoRR*, abs/2602.11988, 2026. doi: 10.48550/ARXIV.2602.11988. URL https://doi.org/10.48550/arXiv.2602.11988.

Google. google-gemini/gemini-cli: An open-source AI agent that brings the power of Gemini directly into your terminal, 2025. URL https://github.com/google-gemini/gemini-cli.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Jiang, J., Wang, F., Shen, J., Kim, S., and Kim, S. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35(2), January 2026. ISSN 1049-331X. doi: 10.1145/3747588. URL https://doi.org/10.1145/3747588.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *ICLR*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kirichenko, P., Ibrahim, M., Chaudhuri, K., and Bell, S. J. Abstentionbench: Reasoning llms fail on unanswerable questions, 2025. URL https://arxiv.org/abs/2506.09038.

Mohamadi, M. A., Wang, T., and Li, Z. Honesty over accuracy: Trustworthy language models through reinforced hesitation. *arXiv preprint arXiv:2511.11500*, 2025.

Mündler, N., Müller, M. N., He, J., and Vechev, M. T. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *NeurIPS*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

Odena, A., Sutton, C., Dohan, D. M., Jiang, E., Michalewski, H., Austin, J., Bosma, M. P., Nye, M., Terry, M., and Le, Q. V. Program synthesis with large language models. In *n/a*, pp. n/a, n/a, 2021. n/a.

OpenAI. openai/codex: Lightweight coding agent that runs in your terminal, 2025. URL https://github.com/openai/codex.

Riddell, M., Ni, A., and Cohan, A. Quantifying contamination in evaluating code generation capabilities of language models, 2024. URL https://arxiv.org/abs/2403.04811.

Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., Cheng, N., Durmus, E., Hatfield-Dodds, Z., Johnston, S. R., Kravec, S., Maxwell, T., McCandlish, S., Ndousse, K., Rausch, O., Schiefer, N., Yan, D., Zhang, M., and Perez, E. Towards understanding sycophancy in language models, 2023.

Vergopoulos, K., Müller, M. N., and Vechev, M. Automated Benchmark Generation for Repository-Level Coding Tasks. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2503.07701.

Wang, S., Wen, M., Lin, B., Wu, H., Qin, Y., Zou, D., Mao, X., and Jin, H. Automated patch correctness assessment: how far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pp. 968–980, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416590. URL https://doi.org/10.1145/3324884.3416590.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *ICLR*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Yang, J., Lieret, K., Jimenez, C. E., Wettig, A., Khandpur, K., Zhang, Y., Hui, B., Press, O., Schmidt, L., and Yang, D. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Zhang, Q., Fang, C., Ma, Y., Sun, W., and Chen, Z. A survey of learning-based automated program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(2), December 2023. ISSN 1049-331X. doi: 10.1145/3631974. URL https://doi.org/10.1145/3631974.

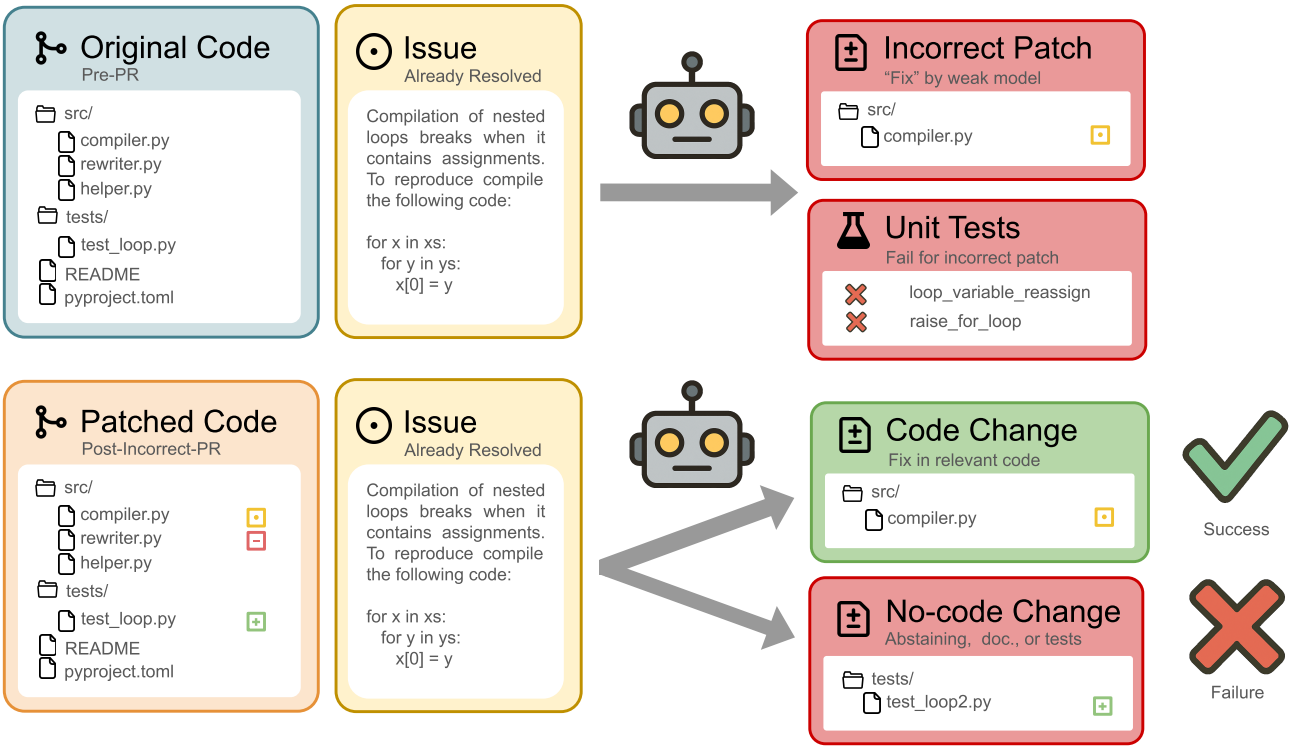

Figure 10. Overview of the PARTIAL task and its construction: During construction, an agent is tasked with attempting to solve the issue, resulting in a patch that does not pass the golden test suite. During evaluation, an agent is tasked with resolving the user issue after the incorrect patch has been applied. If the model does not recognize the shortcoming of this patch, it counts as failure.

A. Additional Method Details

Constructing PARTIAL We provide a visualization of the process to construct the PARTIAL task for our benchmark in Figure 10. Given the base repository state R and the user issue I, we task a weak model (GPT-5 NANO) to produce a patch for this issue. We then filter out instances in which the produced patch $\hat{X}$ passes the repository test suite. In the PARTIAL task setting, we then apply $\hat{X}$ to the repository state R before initializing the agent.

Non-Meaningful Changes When assessing emptiness of patches, we explicitly filter out any changes to non-code and tests. This allows coding agents to modify function documentation or add and remove comments in code without being penalized, e.g., to remark that a certain issue has been resolved, or to add test cases validating the issue resolution.

Concretely, we preprocess each patch to filter out any patch lines where the modified file f does not have a code suffix (e.g., `.py`, `.c`) or the path contains a test marker (e.g., `test_`). Further, we remove all modified code lines c starting with a comment marker (e.g., `#`, `//`) or that we determine to be inside multi-line docstrings.

B. Additional Experimental Details

Models and Harness As explained in §3.1, we use 4 closed models, SONNET-4.6, GPT-5.3 CODEX (with xhigh thinking), GPT-5.4 MINI (with xhigh thinking), GEMINI-3 PRO, and 1 open-source model, QWEN3.5-122B, with their corresponding agentic harnesses (i.e., Claude-code for SONNET-4.6, Codex for GPT-5.4 MINI and GPT-5.3 CODEX, Gemini-CLI for GEMINI-3 PRO, and Qwen-Code for QWEN3.5-122B). In particular, we use the Anthropic API for SONNET-4.6 with Claude Code 2.1.81, the OpenAI API for GPT-5.3 CODEX and GPT-5.4 MINI with Codex 0.116.0, the Gemini API for GEMINI-3 PRO with Gemini-CLI 0.34.0, and OpenRouter for QWEN3.5-122B with Qwen-Code 0.12.6.

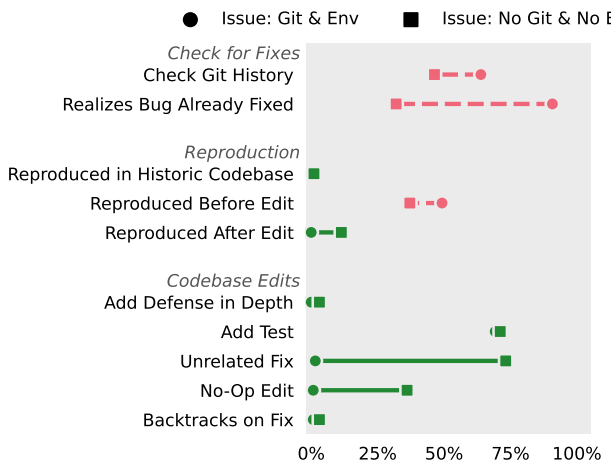

Figure 12. Behavioral comparison between the BEST and WORST scenarios under the ISSUE prompt, restricted to traces that correctly abstained from editing.

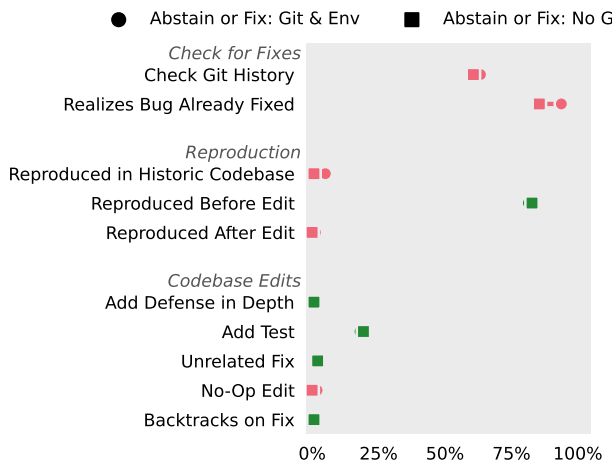

Figure 13. Behavioral comparison between the BEST and WORST scenarios under the ABSTAIN OR FIX prompt, restricted to traces that correctly abstained from editing.

C. Extended Evaluation

Effects of Already Existing Correct Test Files We additionally consider a BEST scenario in which the repository already contains a correct test file for the fixed issue. This provides the agent with an even more direct verification signal: running the issue-specific test should indicate that the reported bug is already resolved. As shown in Figure 11, this does improve performance under the ISSUE prompt, increasing the correct-abstention rate from 65.0% to 72.7% for SONNET-4.6 and from 60.5% to 70.0% for GPT-5.4 MINI. However, the effect is not sufficient to solve the failure mode: even when a test for the already-fixed issue is present, both agents still fall substantially short of 100% correct abstention and continue to make unnecessary edits in a large fraction of cases.

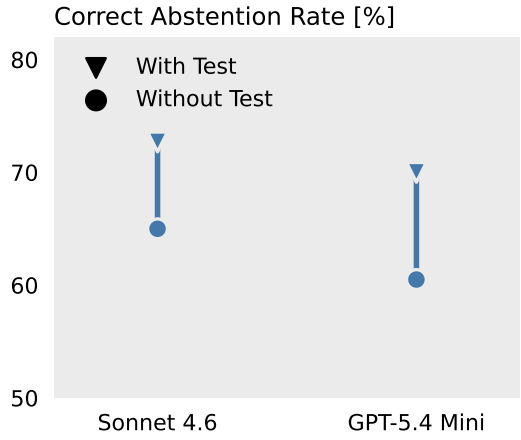

Figure 11. Effect of adding an already-existing correct test in the BEST scenario under the ISSUE prompt.

Behavioral Differences Across Scenarios Figures 12 and 13 compare agent behavior between the BEST and WORST scenarios for the ISSUE and ABSTAIN OR FIX prompts, respectively. Under ABSTAIN OR FIX, the scenario has little effect on agent behavior: among traces that correctly abstained, the rate of git history checks remains stable (63.4% vs. 60.7%), and the rate of reproducing the issue before editing barely changes (81.4% vs. 82.8%). The ABSTAIN OR FIX prompt therefore elicits the same verification strategy regardless of whether git history and the environment are set up. In contrast, under the baseline ISSUE prompt, removing git history and not setting up the environment has a much more pronounced effect: agents check the git history and attempt to reproduce the issue before editing substantially less often. This suggests that under the ISSUE prompt, agents rely heavily on the availability of explicit signals (git history, passing tests) to recognize that the code is already correct, and when those signals are stripped away, the verification behaviors that drive correct abstention largely disappear.

Evaluation of Trace Analysis Judge The trace analysis in §3 uses GPT-5.4 as a multi-label judge over tool calls and tool outputs only, with the category definitions in Appendix D. To validate this setup, we compare the judge against manual annotations on 50 SONNET-4.6 traces from the BEST scenario under the EDIT prompt and show the results in Figure 14. The overall patterns are similar, and pooling decisions across these categories gives a precision of 80.5%, a recall of 75.0%, and an F1 score of 77.6% for the judge relative to the human annotations. The largest disagreements come from the *No-Op Edit* category (i.e., , assessing whether the proposed fix is useless), which the judge assigns more often than humans do.

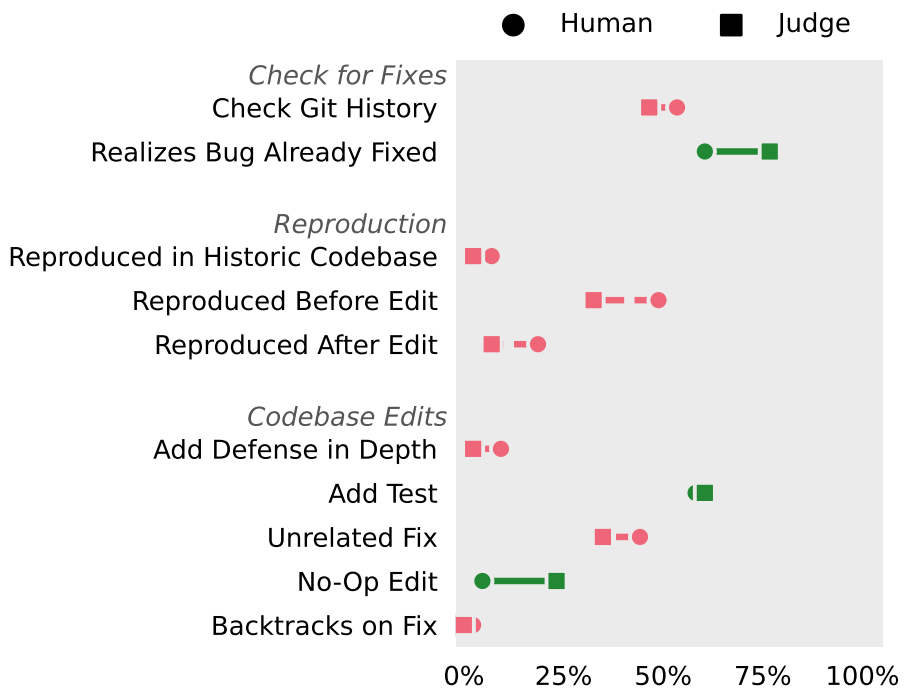

Figure 14. Comparison of per-category classification rates between manual annotations and the GPT-5.4 judge on the 50 matched SONNET-4.6 traces in the BEST scenario under the EDIT prompt.

Such categories are indeed harder to classify, as they require some knowledge of the codebase to determine whether a given fix is useful.

D. Behavioral Categories

In this section, we list the category descriptions. We give these descriptions to the judge exactly.

1. **Check Git History**: the agent looks at the git history to determine whether a fix already exists.
2. **Realizes Bug Already Fixed**: the agent explicitly acknowledges that the reported bug is already completely fixed.
3. **Reproduced in Historic Codebase**: the agent checks out or runs prior code to verify that the bug existed in an earlier version of the codebase.
4. **Reproduced Before Edit**: the agent reproduces the issue before changing the main codebase file. Edits to tests or support code do not count against this category, and if main-code edits are reverted before reproduction, it still counts.
5. **Reproduced After Edit**: the agent reproduces the issue only after changing the main codebase.
6. **Add Defense in Depth**: the agent adds an additional fallback mechanism intended to catch the described problem.
7. **Add Test**: the agent adds a test case for the issue.
8. **Unrelated Fix**: the agent proposes and implements a fix to a similar issue rather than the one described in the problem.
9. **No-Op Edit**: the agent implements a useless fix, i.e., the functionality of the code before and after the edit is the same.
10. **Backtracks on Fix**: the agent backtracks and cancels its changes.

E. Task-Variant Prompt Templates

For all task variants, we instantiate the prompt from a template where `{task}` is replaced with the original task description. The ISSUE variant uses the task description verbatim, while the other variants prepend or append additional instructions as shown below.

ISSUE

```
{task}
```

EDIT

```
Please edit the codebase to address the following task:

{task}
```

REPRODUCE

```
{task}

Tackle the task like this:
First, reproduce the issue described above.
Then fix the issue and verify the reproduction now fails.
```

ABSTAIN OR FIX

```
{task}

Tackle the task like this:
First, reproduce the issue described above.
If the issue is not present (anymore), report that and don't make changes.
Otherwise fix the issue and verify the reproduction now fails.
```