ACTIVE-SWE: BENCHMARKING CODING AGENTS FOR PROACTIVE BUG FIXING WITHOUT ISSUE REPORTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents powered by large language models (LLMs) are increasingly adopted in software engineering (SWE) scenarios, capable of fixing a specific bug in large-scale codebase. However, existing SWE benchmarks typically assume that high-quality issue reports with detailed information are always available, which is easily violated in practice due to the complexity of report acquisition and curation. To address this, we introduce **Active-SWE**, a benchmark for evaluating coding agents on proactively discovering and fixing multiple bugs without report guidance, covering 1,663 tasks across six bug categories and eight languages. Beyond shifting the focus from existing reactive bug fixing to proactive bug fixing, Active-SWE enables a more in-depth evaluation by expanding the scope from fixing a specific recorded bug to multiple-bug fixing and potential bug discovery scenarios. To construct Active-SWE, we propose a novel difficulty-aware task formulation pipeline with a dual-track evaluation framework, facilitating comprehensive evaluation of proactive bug fixing capability. Extensive experiments reveal that most state-of-the-art coding agents struggle with proactive bug fixing tasks, demonstrating limited performance in locating and resolving recorded bugs, handling multiple bug fixing scenarios, and discovering valid potential bugs.

1 INTRODUCTION

Coding agents (Yang et al., 2024a; Zhang et al., 2024) driven by Large Language Models (LLMs) (Chang et al., 2024; Zhao et al., 2026) have been increasingly applied to Software Engineering (SWE) scenarios (Fan et al., 2023; Jin et al., 2024), with the research community expanding from simple programming tasks to real-world software engineering tasks such as fixing a specific bug in large codebase. To comprehensively evaluate coding agents in repository-level SWE scenarios, representative benchmarks construct instances from GitHub issues and their corresponding pull requests (PRs), *e.g.*, SWE-bench Verified (Jimenez et al., 2024) and SWE-bench Pro (Deng et al., 2025), where coding agents are typically tasked with generating a code patch for the specific bug described in the human-written issue report.

However, existing SWE benchmarks (Zhang et al., 2026; Zan et al., 2026) evaluate coding agents under the ideal assumption that human-authored issue reports are available and contain desirable debugging information, *e.g.*, fault locations, error messages, and expected behavior, which is daunting to satisfy as illustrated in Fig. 1(a). On the one hand, the acquisition of issue reports is inherently post-hoc and prohibitively expensive, *i.e.*, bugs are typically identified after escaping into production, leading to severe consequences such as resource waste, service outages, and security vulnerabilities. For instance, the 2024 CrowdStrike outage (CrowdStrike, 2024) was caused by an out-of-bounds read bug in the Content Interpreter, crashing millions of Windows devices and incurring billions of dollars in losses (Microsoft, 2024; Parametrix, 2024). On the other hand, due to the lack of expert knowledge and standardized practices, human-authored descriptions in issue reports are often ambiguous and incomplete, with insufficient information to resolve potential problems. In particular, recent analyses (OpenAI, 2026a;b) from OpenAI reveal that even carefully curated benchmarks such as SWE-bench Verified and SWE-Bench Pro contain non-negligible low-quality issue reports, resulting in tasks that are infeasible to resolve. Taken together, existing benchmarks focus

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

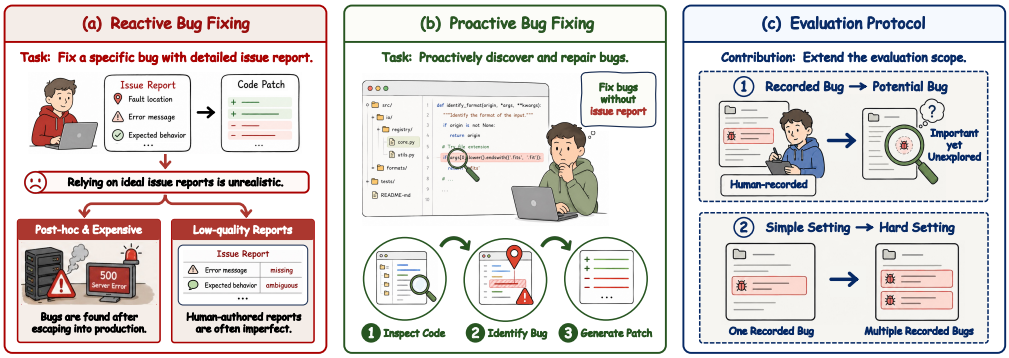

Figure 1: (a) **Reactive Bug Fixing:** existing benchmarks aim to fix a specific bug with detailed issue reports, largely overlooking that such desirable reports are often unavailable due to costly bug identification and unreliable issue descriptions. (b) **Proactive Bug Fixing:** we introduce Active-SWE to evaluate coding agents on proactive bug discovery and repair without issue reports, posing greater demands on code inspection, bug identification, and repair capabilities. (c) **Comprehensive Evaluation:** two novel evaluation dimensions are designed for the comprehensive assessment of proactive bug fixing capability: i) validating both recorded bug fixing and potential bug discovery; ii) supporting bug fixing tasks that involve multiple bugs rather than a specific fix.

on evaluating the reactive bug fixing capability of coding agents, which relies on human involvement and high-quality issue reports.

In light of the limitation, we pose a more general and challenging question for SWE scenarios:

> *Can coding agents proactively discover and resolve bugs before they are reported or even hinted at by developers?*

As shown in Fig. 1(b), this capability involves inspecting the code repository, identifying potentially multiple bugs, and generating corresponding fixes without issue reports, which we refer to as *proactive bug fixing*. To answer this question, we introduce Active-SWE, which might be the first benchmark to comprehensively evaluate the proactive bug fixing capability of coding agents. Different from existing benchmarks, Active-SWE sheds light on the capacity of coding agents to act as active bug fixers, thus facilitating a more in-depth evaluation from two perspectives: i) **multiple bug scenarios**: existing benchmarks focus on resolving one specific bug, while Active-SWE further validates the capability to discover and resolve multiple bugs in proactive bug fixing; ii) **potential bug scenarios**: beyond the evaluations on recorded bugs, Active-SWE extends the evaluation scope to reveal important yet under-explored bugs in real-world repositories.

In this paper, we introduce a novel pipeline that systematically constructs Active-SWE from real-world SWE issues and evaluates it in a rigorous and reproducible manner. First, we adopt a taxonomy-driven data curation module to derive executable repository snapshots with discoverable bugs from GitHub PRs, which supports reliable benchmark construction and evaluation. Second, we propose a novel difficulty-aware task formulation module that reformulates raw PRs as proactive bug fixing tasks and then derives tasks involving multiple bug fixes. Third, we design a dual-track evaluation framework to comprehensively evaluate the bug fixing capability of coding agents from both the recorded bug repair and potential bug discovery perspectives. In summary, the major contributions and novelties of this work are given as follows.

- We propose Active-SWE, which could be one of the first benchmarks that shifts the focus from reactive to proactive bug fixing. Different from existing benchmarks, Active-SWE evaluates the capability of coding agents to proactively discover and fix bugs without issue reports, covering 1,663 tasks across diverse bug categories and programming languages.
- For comprehensive evaluations of proactive bug fixing capability, we propose a novel benchmark construction pipeline with a dual-track evaluation framework, extending evaluation scope to both multiple bug fixing and potential bug discovery scenarios.

- Extensive experiments on the Active-SWE demonstrate that even state-of-the-art coding agents still struggle with proactive bug fixing tasks, especially in locating and resolving recorded bugs, handling hard tasks with multiple bugs, and discovering valid potential bugs.

2 Related Work

Real-world software engineering requires nuanced reasoning over large repositories and performing complex code modifications under diverse tasks (Yang et al., 2024b; Raghavendra et al., 2026). According to the characteristics of tasks, existing SWE benchmarks could be broadly grouped into three categories: i) issue-solving benchmarks (Zhang et al., 2026), which focus on evaluating whether LLMs could resolve bugs or other software problems described in the issue; ii) test-generation benchmarks (Liu et al., 2026; Mündler et al., 2024), where LLMs are tasked to derive executable tests that capture expected behaviors from issue descriptions or curated documents; iii) feature-development benchmarks (Zhou et al., 2026; Raghavendra et al., 2026), which require LLMs to translate high-level development documents into function-level or repository-level implementations.

In this work, we systematically evaluate a highly-practical yet less-explored task, *i.e.*, proactive bug fixing, which involves discovering and repairing bugs in the repository without issue reports. In other words, existing SWE benchmarks typically rely on carefully-curated issues or documents that would specify predefined objectives and provide detailed information. In contrast, the bug fixing task aims to first identify fixing objectives and then implement them without human involvement, which poses higher demands on code exploration, objective identification, and code implementation capabilities of coding agents.

3 Active-SWE

Active-SWE is a benchmark for evaluating the proactive bug fixing capability of coding agents, which comprises 1,663 tasks spanning six bug categories and eight programming languages. In this section, we will introduce the task formulation, data curation pipeline and the evaluation framework of Active-SWE.

3.1 Task Formulation

Let r_i denote the repository snapshot of the i-th bug fixing instance, $\mathcal{F}_i$ indicate the corresponding set of files to be inspected, and $\hat{\mathbf{b}}_i$ represent the recorded bugs contained in $\mathcal{F}_i$. For a given repository snapshot r_i, the goal of the bug fixing task is to generate a code patch c_i for resolving the discovered bugs within the review scope $\mathcal{F}_i$, *i.e.*,

$$c_i = \mathcal{L}\left(r_i, \mathcal{F}_i, \mathcal{T}_{\text{R}}\right), \tag{1}$$

where $\mathcal{T}_{\text{R}}$ denotes the unified bug fixing task template and $\mathcal{L}$ denotes the evaluated LLM. After that, we adopt the following test-driven evaluation protocol to verify the effectiveness of the predicted patch c_i, *i.e.*,

$$\mathcal{P}(c_i \oplus r_i, \mathbf{t}_i) \in [0, 1], \tag{2}$$

where $\mathbf{t}_i$ indicates the test set, $\oplus$ denotes the patch application operation, $\mathcal{P}(\cdot) \in [0, 1]$ denotes the test pass rate, and $\mathcal{P}(c_i \oplus r_i, \mathbf{t}_i) = 1$ *i.f.f.* all tests in $\mathbf{t}_i$ are passed successfully after applying c_i to r_i. Different from the existing bug fixing benchmarks that rely on the costly, post-hoc issue reports, the bug fixing task shifts the focus towards proactively discovering and repairing bugs with a unified bug fixing template, which is more general and practical in real-world SWE scenarios. See more discussions with real-world applications in Appendix C. In the following, we will elaborate on the construction pipeline of Active-SWE and the evaluation framework for validating the bug fixing capabilities of coding agents.

3.2 Taxonomy-driven Data Curation

To collect data for constructing Active-SWE, it is essential to curate executable repository snapshots with realistic and discoverable bugs. Accordingly, we propose a taxonomy-driven data curation

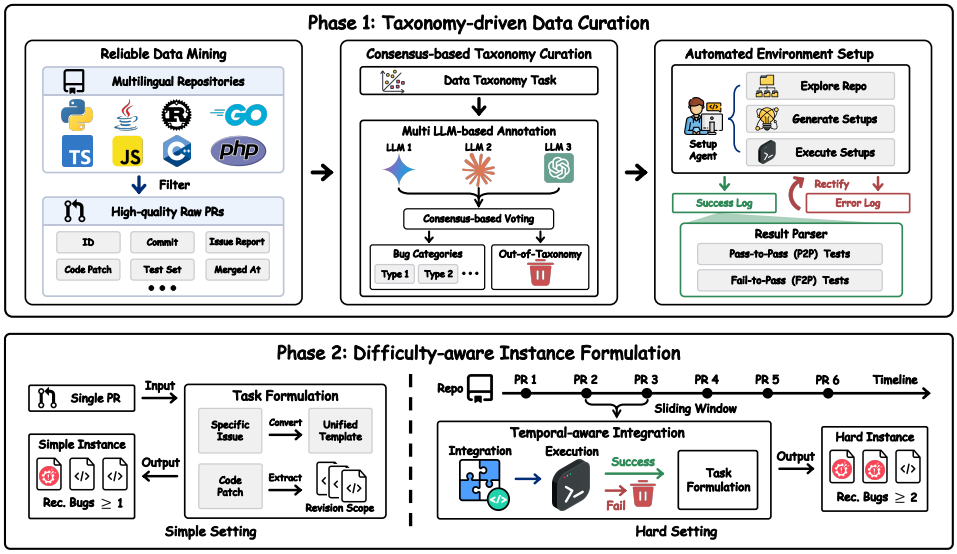

Figure 2: **Construction pipeline of Active-SWE.** The pipeline consists of two phases: i) we crawl high-quality multilingual PRs from GitHub, identify discoverable bug fixing PRs, and build executable environments with a setup agent; ii) we reformulate mined PRs into proactive bug fixing tasks, *i.e.*, each simple instance is derived from a single PR, while each hard instance is constructed by integrating temporally adjacent PRs with a sliding window mechanism.

pipeline consisting of reliable data mining, consensus-based taxonomy curation, and automated environment setup, as illustrated in Fig. 2.

Reliable Data Mining. GitHub provides a rich source of real-world bug fixing histories, where PRs record how developers identify and repair bugs in evolving codebases. Since the raw PRs are usually noisy and poorly documented, we first mine high-quality PRs from open-source repositories. Specifically, we select PRs from **87** popular open-source repositories across **eight** programming languages, each with at least **1,000** GitHub stars. For each repository, we crawl its historical PRs and select those linked to GitHub issues with reports, reference code patch, reference test set, merged time, and so on.

Consensus-based Taxonomy Curation. Although the mined PRs are of high quality, not all of them correspond to bug fixing activities, *e.g.*, some involve feature implementations or API extensions. To remedy this, we propose a consensus-based curation mechanism for filtering out PRs without discoverable bugs. Specifically, for a given PR with issue report s_i, reference code patch $\hat{c}_i$, and reference test set $\hat{\mathbf{t}}_i$, we employ multiple LLM-based annotators to assign taxonomy labels as follows,

$$y_i^k = \mathcal{A}^k\left(s_i, \hat{c}_i, \hat{\mathbf{t}}_i, \mathcal{T}_{\mathrm{C}}\right), \tag{3}$$

where y_i^k denotes the taxonomy label predicted by the k-th annotator $\mathcal{A}^k$, $\mathcal{T}_{\mathrm{C}}$ denotes the taxonomy task template. After that, we derive the final taxonomy label through consensus-based voting strategy, *i.e.*,

$$y_i = \mathcal{V}\left(y_i^1, \ldots, y_i^K\right), \tag{4}$$

where K denotes the number of annotators and $\mathcal{V}(\cdot)$ denotes the voting operation. In the implementation, the taxonomy is defined with software engineering experts, consisting of **6** major bug categories and an additional *Out-of-Taxonomy* category. PRs labeled as *Out-of-Taxonomy* are discarded, while the remaining PRs are retained as valid ones.

Automated Environment Setup. To facilitate faithful execution and evaluation of bug fixing tasks, we employ an LLM-based setup agent to establish docker-based environments and test scripts for retained PRs. Specifically, the setup agent adopts the following ReAct-style (Yao et al., 2022) process: i) **explore** r_i to identify build commands and test commands, then generate the dockerfile and test script; ii) **execute** the generated setups by building the docker image and running test script; iii) **rectify** dockerfile and test script based on execution feedback until success or the turn limit is reached.

To facilitate the subsequent benchmark construction and evaluation, the setup agent is required to generate the test script that reports test-level outcomes. According to the test-level outcomes, the reference test set $\hat{\mathbf{t}}_i$ could be decomposed into fail-to-pass tests $\hat{\mathbf{f}}_i$ and pass-to-pass tests $\hat{\mathbf{p}}_i$.

3.3 Difficulty-aware Task Formulation

In this section, we elaborate on how to reformulate raw PRs into proactive bug fixing tasks and construct tasks with different difficulty levels.

Simple Setting. For the simple setting, each bug fixing instance is derived from a single valid PR, whose files pending review contain at least one recorded bug, *i.e.*, $|\hat{\mathbf{b}}_i| \geq 1$. To be specific, for a given PR, we discard issue report s_i and instead treat a unified task template $\mathcal{T}_{\mathrm{R}}$ as instruction, which prompts coding agents to proactively fix bugs. After that, we extract the files pending review $\mathcal{F}_i$ from reference code patch $\hat{c}_i$, thus guaranteeing the review scope covers the recorded bug,

$$\mathcal{F}_i = g\left(\hat{c}_i\right), \tag{5}$$

where $g(\cdot)$ denotes the file extraction operation. The remaining artifacts are directly inherited from the PR to construct the simple instance $(r_i, \hat{\mathbf{t}}_i, \mathcal{F}_i, \mathcal{T}_{\mathrm{R}})$.

Hard Setting. For the hard setting, we adopt the temporal-aware integration mechanism to formulate tasks with multiple recorded bugs, where the review scope involves at least M bugs, *i.e.*, $|\hat{\mathbf{b}}_i| \geq M$. The key insight is that an earlier repository snapshot may contain multiple bugs later addressed by subsequent PRs, enabling us to aggregate them for deriving hard instance. Specifically, for each repository, we sort valid PRs by merge time and slide a temporal window over the ordered sequence, where $\mathbf{w}_i$ denotes the indices of PRs in the i-th window. After that, we derive the integrated artifacts of the corresponding hard instance as follows,

$$\hat{c}_i = \bigcup_{j \in \mathbf{w}_i} \hat{c}_j, \quad \hat{\mathbf{t}}_i = \bigcup_{j \in \mathbf{w}_i} \hat{\mathbf{t}}_j, \tag{6}$$

where $\bigcup$ denotes the integration operation. To verify the integrated artifacts $\hat{c}_i$ and $\hat{\mathbf{t}}_i$ are valid, we preserve the high-quality hard instance only if the following condition is satisfied, *i.e.*, $\hat{\mathbf{f}}_i = \bigcup_{j \in \mathbf{w}_i} \hat{\mathbf{f}}_j$. In other words, the fail-to-pass tests of the hard instance are expected to be the union of those from the individual PRs. Then, we extract the integrated review scope $\mathcal{F}_i$ according to Eq. 5, while the other artifacts are inherited from the earliest PR in $\mathbf{w}_i$.

3.4 Dual-Track Evaluation Framework

As discussed in Introduction, the key to evaluating the proactive bug fixing capability of coding agents lies in two dimensions: i) whether agents could discover and repair the recorded bugs; ii) whether agents could reveal valid yet under-explored bugs. To this end, we propose a novel dual-track framework to facilitate the comprehensive evaluation of coding agents.

3.4.1 Recorded Bug Evaluation

Existing issue-driven benchmarks rely on detailed debugging cues provided by issue reports, which significantly simplify the bug localization and repair process. Since reports are unavailable in proactive bug fixing tasks, it is necessary to diagnose the limitations of coding agents in terms of the two aspects. To this end, we propose decoupled metrics for evaluating recorded bug fixing capabilities of coding agents, which treat the reference code patch $\hat{c}_i$ and the reference test set $\hat{\mathbf{t}}_i$ as the oracle for bug localization and code repair, respectively.

Localization Recall (LR) and Precision (LP): we quantify localization capability with two fine-grained edit-matching metrics, *i.e.*,

$$\mathrm{LR} = \frac{|\hat{\mathcal{H}}_i^{\mathrm{hit}}|}{|\mathcal{H}(\hat{c}_i)|}, \quad \mathrm{LP} = \frac{|\mathcal{H}_i^{\mathrm{hit}}|}{|\mathcal{H}(c_i)|}, \tag{7}$$

where c_i denotes the generated code patch in Eq. 1, $\mathcal{H}(\cdot)$ extracts the edit hunks from the patch, and $\hat{\mathcal{H}}_i^{\mathrm{hit}} = \{\hat{h}_j \mid \hat{h}_j \in \mathcal{H}(\hat{c}_i), \exists h_k \in \mathcal{H}(c_i), \hat{h}_j \sim h_k\}$ and $\mathcal{H}_i^{\mathrm{hit}} = \{h_k \mid h_k \in \mathcal{H}(c_i), \exists \hat{h}_j \in$

$\mathcal{H}(\hat{c}_i), h_k \sim \hat{h}_j\}$ with $\sim$ indicating that two hunks are matched, Such behavior not only treats the reference code patch as oracle for reliable localization capability estimation, but also decomposes the code patch into fine-grained units and thus enables scalable analysis in multi-bug scenarios.

Resolved: we employ the test-driven evaluation protocol for estimating the bug repair capability, *i.e.*,

$$\text{Resolved} = \mathbb{I}\left[\mathcal{P}(c_i \oplus r_i, \hat{\mathbf{t}}_i) = 1\right], \tag{8}$$

where $\mathbb{I}[\cdot]$ is an indicator function evaluating to 1 *i.f.f.* the condition is satisfied, and Resolved indicates whether the recorded bugs are fully addressed.

3.4.2 Potential Bug Evaluation

Beyond recorded bugs, Active-SWE further evaluates whether the potential bugs revealed in the generated code patch are valid. However, evaluating potential bugs is quite challenging as no reference oracle is available. Instead of exhaustingly collecting human-curated oracles, we reformulate the patch validation as the test generation problem, where revealed bugs are credible only if reproduced by valid test evidence. To this end, we propose a trustworthy test-driven evaluation mechanism, which tasks agents with generating tests for reproducing the revealed bugs. Specifically, given a predicted code patch c_i, the evaluated LLM is instructed to generate a set of tests $\mathbf{t}_i$ for reproducing the bugs in c_i, *i.e.*,

$$\mathbf{t}_i = \mathcal{L}\left(r_i, c_i, \mathcal{F}_i, \mathcal{T}_\text{S}\right), \tag{9}$$

where $\mathcal{T}_\text{S}$ denotes the test generation task template. For a trustworthy evaluation, we further adopt an LLM-based judge agent to derive the semantic association between the revealed bugs and the generated tests, *i.e.*,

$$\mathbf{M}_i = \mathcal{J}\left(r_i, c_i, \mathbf{t}_i, \mathcal{T}_\text{J}\right), \quad \mathbf{M}_i \in \{0,1\}^{|\mathbf{b}_i| \times |\mathbf{t}_i|}, \tag{10}$$

where $\mathcal{J}$ denotes the judge agent, $\mathcal{T}_\text{J}$ denotes the judge task template, $\mathbf{b}_i$ denotes the set of bugs revealed in c_i, and $\mathbf{M}_i[j,k] = 1$ indicates that the j-th bug in $\mathbf{b}_i$ is reproduced by the k-th test in $\mathbf{t}_i$. Although the generated tests provide post-hoc evidence, they might still suffer from the following failure modes: i) some tests might not exhibit fail-to-pass behavior and thus are invalid for reproducing bugs; ii) even if all tests are valid, they might only cover a subset of the revealed bugs. To capture these failure modes, we introduce the following metrics for more in-depth evaluation.

Test Validity (TV): we verify whether all generated tests exhibit fail-to-pass behavior, *i.e.*,

$$\text{TV} = \mathbb{I}\left[|\mathbf{t}_i| = |\mathbf{f}_i|\right], \tag{11}$$

where $\mathbf{f}_i$ denotes the subset of generated tests that exhibit fail-to-pass behavior.

Revealed: we regard the revealed bugs as valid only if each of them is supported by test evidence, *i.e.*,

$$\text{Revealed} = \mathbb{I}\left[\forall j \in \{1,\ldots,|\mathbf{b}_i|\},\ \exists k \in \{1,\ldots,|\mathbf{f}_i|\} : \mathbf{M}_i[j,k] = 1\right]. \tag{12}$$

In the implementation, the Revealed metric is built upon $\text{TV} = 1$, thus ultimately indicating whether the agent reveals potential bugs with corresponding valid test evidence.

4 Experiments

In this section, we conduct extensive experiments on the proposed Active-SWE to evaluate the bug fixing capability of state-of-the-art (SOTA) LLMs.

4.1 Experiment Configurations

Data Statistics. For comprehensive evaluation, we construct 1,663 bug fixing tasks with 1,411 simple instances and 252 hard instances, covering six major bug categories, *i.e.*, Logic & Computation, Reference & Data Flow, Data Processing & Type, Domain-specific Workflow, Exception Safety, and State & Lifecycle and eight mainstream programming languages, *i.e.*, Python, Go, Rust, PHP, Ruby, JavaScript/TypeScript (JS/TS), Java, and C/C++. For high-quality and efficient evaluation, human experts curate Active-SWE with 400 tasks, including 300 simple and 100 hard instances. To support broader-scale evaluation, we provide Active-SWE-Extend with all constructed tasks.

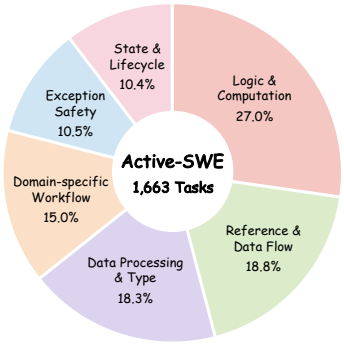

Figure 3: Bug category distribution of the Active-SWE.

Table 1: Median values of different attributes in the Active-SWE under the simple and hard settings.

Category	Attribute	Simple	Hard
Codebase	# Lines	193k	353k
	# Files	910	926
Gold Patch	# Lines edited	12	19
	# Files edited	1	2
	# Hunks edited	2	3
Tests	# Fail to Pass	1	2
	# Pass to Pass	16	104

Table 2: Performance comparisons of different state-of-the-art LLMs.

Model	Recorded			Potential		
	LR	LP	Resolved	Count	TV	Revealed
Closed-Source Models						
Claude Opus 4.8	29.1	35.0	20.0	1.2	81.0	75.0
Claude Sonnet 4.6	16.5	18.0	8.3	1.6	78.3	53.8
GPT-5.5	28.8	24.1	18.5	3.0	87.5	65.5
GPT-5.4	17.2	17.6	8.8	1.3	52.3	32.8
Gemini-3.1-Pro	26.8	18.8	15.8	2.8	77.8	57.0
Qwen3.7-Max	18.6	19.4	11.8	1.7	81.0	59.8
Seed2.1	21.1	20.3	11.0	1.8	64.5	52.8
Open-Source Models						
DeepSeek-V4-Pro	17.5	18.9	9.5	2.0	64.3	53.3
GLM-5.2	13.9	17.0	13.0	1.7	79.0	69.0
Kimi-K2.7-Code	18.7	16.1	14.8	2.6	65.0	47.8
Kimi-K2.6	17.2	14.7	14.0	2.1	67.3	51.8
MiniMax-M2.7	9.9	13.4	7.8	1.6	45.0	38.5
Ring-2.6-1T	13.3	14.9	11.0	1.9	65.5	56.3
Hy3	12.3	15.8	9.8	1.1	77.3	58.5
Qwen3.5-397B	12.4	13.9	6.8	1.6	56.8	42.0
Qwen3.5-122B	11.2	13.8	5.8	1.4	46.8	40.0
Qwen3.5-35B	10.0	11.2	5.0	1.3	32.5	27.5

Baselines. To evaluate the bug fixing capability of existing LLM-based coding agents, we select SOTA closed-source and open-source models with strong code reasoning capabilities, including Claude Opus 4.8 (Anthropic, 2026b) and Claude Sonnet 4.6 (Anthropic, 2026a), GPT-5.5 (Singh et al., 2025) and GPT-5.4, Gemini-3.1-Pro (Google DeepMind, 2026), Qwen3.7-Max (Qwen Team, 2026), Seed2.1 (Seed, 2026), DeepSeek-V4-Pro (DeepSeek-AI, 2026), GLM-5.2 (Zeng et al., 2026), Kimi-K2.7-Code (Team et al., 2026) and Kimi-K2.6, MiniMax-M2.7 (Chen et al., 2026), Ring-2.6-1T Li et al. (2026), Hy3 Tencent Hy Team (2026), Qwen3.5 series (Team, 2026). For fairness, we adopt Claude Code as the unified scaffold for evaluation, as it is one of the most widely used agentic coding systems.

Implementation Details. We treat three closed-source models as LLM-based annotators for consensus-based taxonomy curation, including Gemini-3.1-Pro, GPT-5.4, and Claude Opus 4.8. For potential bug evaluation, we adopt one of the most popular open-source models, *i.e.*, Qwen3.5-

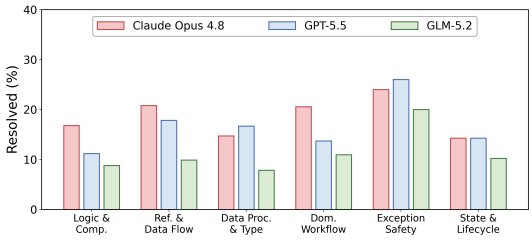

Figure 4: Performance comparison across various bug categories.

Table 3: Performance comparison between reactive and proactive bug fixing on a subset of Active-SWE.

Model	Resolved	
	Reactive	Proactive
Claude Opus 4.8	59.0	26.0
GPT-5.5	57.0	22.0
GLM-5.2	58.0	17.0

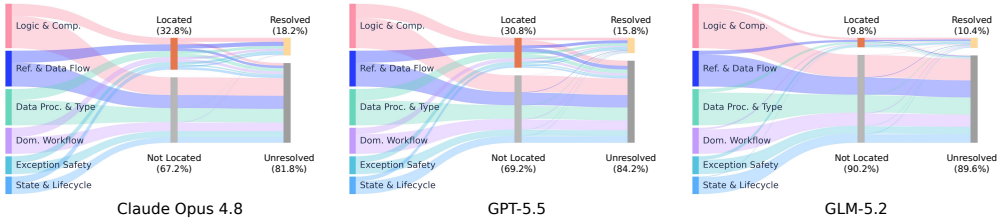

Figure 5: Relationship between bug localization and resolution. The recorded bugs are considered as *Located* if LR ≥ 0.5.

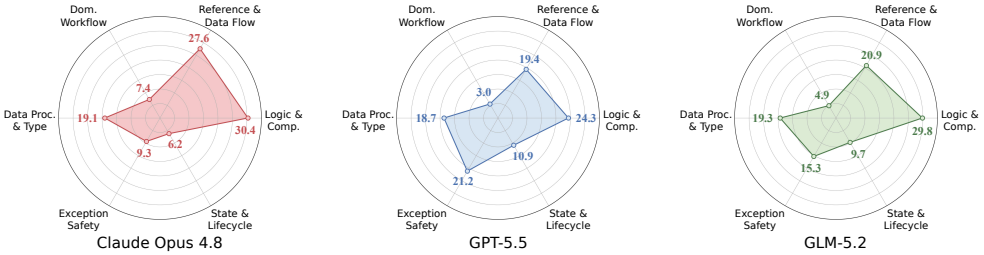

Figure 6: Distribution of revealed potential bugs.

397B, as the LLM-based judge agent to facilitate reproducible and accessible evaluation. As for the hard setting, due to the high failure rate during temporal-aware integration, we set the window size to 2, *i.e.*, each hard instance contains at least two recorded bugs.

4.2 Main Results

We compare SOTA LLMs on the Active-SWE benchmark in Table 2, where "Count" refers to the number of generated tests. From the results, one could have the following observations. For recorded bug evaluation, existing LLMs achieve limited localization scores and resolved rates, with the best resolved rate reaching only 20.0%, indicating that proactively discovering and repairing recorded bugs remains highly challenging. Moreover, LLMs with stronger bug localization capability tend to achieve higher resolved rates, suggesting that localization quality is generally associated with repair success. For potential bug evaluation, the "Count" and "TV" metrics are not always aligned, indicating that generating more bug-reproducing tests does not necessarily imply that these tests are valid. Across various LLMs, Claude Opus 4.8 and GLM-5.2 achieve the strongest overall performance on both recorded bug fixing and potential bug discovery. Besides, within the Qwen3.5 family, larger models with stronger coding capabilities tend to achieve better proactive bug fixing performance.

4.3 Analytic Experiments

Proactive bug fixing capability varies across bug categories. As shown in Fig. 4, LLMs achieve better resolved rates on Exception Safety and struggle with bugs of State & Lifecycle, suggesting that LLMs are better at discovering and fixing certain categories of bugs.

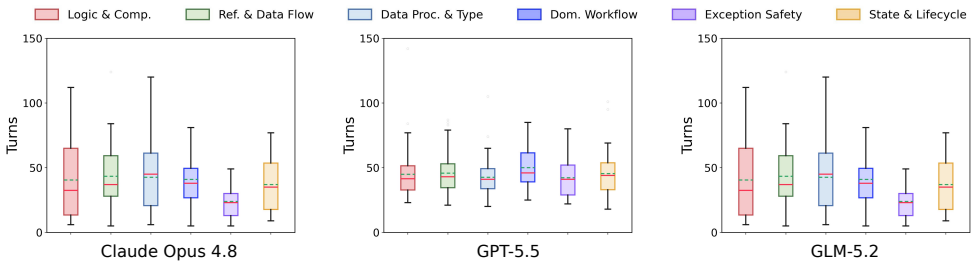

Figure 7: Reasoning turns across various recorded bug categories.

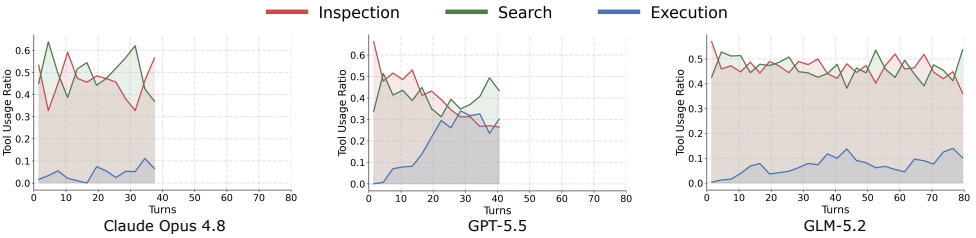

Figure 8: Tool-usage patterns and turn dynamics in proactive bug fixing.

Proactive bug fixing is more challenging than reactive bug fixing. As demonstrated in Table 3, LLMs suffer performance degradation in the proactive setting compared with the reactive setting, suggesting that even SOTA LLMs still struggle to resolve bugs without issue reports.

Right bug localization is halfway to successful bug fixing. From the results in Fig. 5, correctly located bugs are more likely to be resolved, suggesting that accurate localization is positively associated with successful repair. It is worth noting that some successful repairs do not match the reference oracle, suggesting that LLMs may discover alternative repair paths beyond the human-recorded one.

LLMs tend to reveal certain potential bugs. We visualize the category distribution of revealed potential bugs in Fig. 6. The revealed bugs are concentrated in Logic & Computation, Reference & Data Flow, with fewer cases in Domain-specific Workflow and State & Lifecycle. Such a phenomenon suggests that LLMs are more sensitive to discovering local correctness and flow-related issues than to domain- or state-dependent bugs.

LLMs exhibit similar reasoning depth across bug categories. As demonstrated in Fig. 7, reasoning turns are broadly comparable across bug categories within each model, indicating that LLMs exhibit similar reasoning depth and interaction costs regardless of the target bug categories.

Different LLMs share common reasoning patterns but differ in reasoning depth. From the results in Fig. 8, LLM-based coding agents invest more early-stage effort in code inspection in the proactive scenario and then shift from inspection to search and execution, highlighting a progressive process from repository exploration to bug localization and repair. Besides, different LLMs exhibit distinct reasoning depths, *e.g.*, GLM-5.2 allocates more turns for in-depth analysis, while Claude Opus 4.8 resolves bugs with shorter and more efficient trajectories.

5 Conclusion

In this paper, we propose Active-SWE, a novel benchmark for comprehensively evaluating LLM-based coding agents in proactive bug fixing capability. With the proposed data curation pipeline, Active-SWE curates 1,663 high-quality bug fixing tasks spanning six major bug categories and eight programming languages. To facilitate more in-depth validation, we design a novel task formulation strategy and a dual-track evaluation framework, which expand the evaluation scope from both difficulty and coverage perspectives. Extensive experiments reveal the limitations of existing SOTA LLMs in proactive bug fixing and provide insights for subsequent research. In the future, we plan to explore broader downstream scenarios of proactive bug fixing, such as vulnerability discovery and security-oriented code auditing.

AI USE STATEMENT

In this work, we used generative AI tools implementing methods and supporting dataset curation and evaluation. Specifically, the use of LLMs for data annotation, data filtering, environment setup, and automated evaluation is described in the main text. We have not used generative AI tools for helping develop theoretical models or conceptual frameworks, formulating mathematical claims, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, assisting with translation, or interpreting results, and providing critical ingredients for proving mathematical claims and assisting in the writing of proofs are not applicable to this work. Additionally, we used generative AI tools for editing a research paper to improve readability. We have reviewed all AI-assisted work. Specifically, we check AI-assisted code for correctness and AI-assisted text for technical accuracy and consistency with our intended meaning. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Anthropic. System card: Claude sonnet 4.6. https://www-cdn.anthropic.com/78073f739564e986ff3e28522761a7a0b4484f84.pdf, 2026a.

Anthropic. System card: Claude opus 4.8. https://www-cdn.anthropic.com/0b4915911bb0d19eca5b5ee635c80fef830a37ea.pdf, 2026b.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology*, 2024.

Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, Boji Dan, Changqing Yu, Chao Wang, Cheng Ma, Cheng Zhong, et al. The minimax-m2 series: Mini activations unleashing max real-world intelligence. *arXiv preprint arXiv:2605.26494*, 2026.

CrowdStrike. External technical root cause analysis — channel file 291. https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf, 2024.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf, 2026.

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

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *ICSE-FoSE*, 2023.

Google DeepMind. Gemini 3.1 pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, 2026.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. In *ACL*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Ang Li, Ben Liu, Bin Han, Bin Hu, Bin Jing, Binbin Hu, Bing Li, Cai Chen, Caizhi Tang, Changxin Tian, et al. Ling and ring 2.6 technical report: Efficient and instant agentic intelligence at trillion-parameter scale. *arXiv preprint arXiv:2606.15079*, 2026.

Steven Liu, Jane Luo, Xin Zhang, Aofan Liu, Hao Liu, Jie Wu, Ziyang Huang, Yangyu Huang, Yu Kang, and Scarlett Li. Testexplora: Benchmarking llms for proactive bug discovery via repository-level test generation. *arXiv preprint arXiv:2602.10471*, 2026.

Microsoft. Helping our customers through the crowdstrike outage. https://blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/, 2024.

Niels Mündler, Mark N Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. In *NeurIPS*, 2024.

OpenAI. Why swe-bench verified no longer measures frontier coding capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/, 2026a.

OpenAI. Separating signal from noise in coding evaluations. https://openai.com/index/separating-signal-from-noise-coding-evaluations/, 2026b.

Parametrix. Crowdstrike to cost fortune 500 $5.4b; insured loss range of $0.54b - $1.08b. https://www.parametrixinsurance.com/in-the-news/crowdstrike-to-cost-fortune-500-5-4-billion-insured-loss-range-of-540-million-to-1-08-billion, 2024.

Qwen Team. Qwen3.7: The agent frontier, May 2026. URL https://qwen.ai/blog?id=qwen3.7.

Mohit Raghavendra, Soham Dan, Miguel Romero Calvo, Yannis Yiming He, Johannes Baptist Mols, Gautam Anand, Cole McCollum, Edgar Arakelyan, Vijay Bharadwaj, Andrew Park, et al. Swe atlas: Benchmarking coding agents beyond issue resolution. *arXiv preprint arXiv:2605.08366*, 2026.

Bytedance Seed. Seed2. 0 model card: Towards intelligence frontier for real-world complexity. *arXiv preprint arXiv:2607.00248*, 2026.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents. https://qwen.ai/blog?id=qwen3.5, 2026.

Tencent Hy Team. Hy3. https://huggingface.co/tencent/Hy3, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *ICLR*, 2025.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

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

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024b.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. In *NeurIPS*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. In *NeurIPS*, 2026.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *ACL*, 2024.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! In *NeurIPS*, 2026.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Zican Dong, Yupeng Hou, Beichen Zhang, Yingqian Min, Junjie Zhang, Peiyu Liu, et al. A survey of large language models. *Frontiers of Computer Science*, 2026.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, et al. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026.

APPENDIX

A More Data Statistics **14**

B Related Work of Coding Agents **15**

C Discussions on Real-world SWE Application **15**

D More Experimental Results **16**

D.1 Experimental Results under Simple and Hard Settings 16

D.2 Experimental Results across Programming Languages 17

D.3 Experimental Results under Various Scaffolds 17

D.4 Experimental Results on Cost Efficiency 18

D.5 Analytic Study on Edit Complexity 19

D.6 Analytic Study on Performance across Time Periods 19

D.7 Analytic Study on Reactive Bug Fixing 20

D.8 Analytic Study on Broader Review Scope 20

D.9 Analytic Study on Different Judge Agents 20

D.10 Analytic Study on the Overlap Between Potential and Recorded Bugs 21

E Task Templates **22**

F Case Study **26**

A MORE DATA STATISTICS

In this section, we present additional data statistics of our benchmark in Tables 4 and 5, which report the feature distributions of Active-SWE-Extend and the curated Active-SWE.

Active-SWE-Extend exhibits variation in patch complexity across languages. Specifically, Rust and Go involve more fragmented patches, with 12.2 and 9.5 hunks on average, respectively, while Go and Java touch the largest number of files. In contrast, the code patches of PHP and Ruby instances are generally more localized, involving fewer files and hunks per patch. Besides, the test statistics also vary notably, *e.g.*, Python and Ruby have the largest average number of pass-to-pass tests, while JS/TS and Python have the highest average number of fail-to-pass tests.

Active-SWE preserves the multilingual and heterogeneous nature of Active-SWE-Extend and provides a compact subset for main experiments, which includes 400 instances across the same eight languages. Compared with Active-SWE-Extend, Active-SWE retains diverse patch and test characteristics, including localized edits in Ruby and PHP, more fragmented patches in Go and Rust, and non-trivial regression coverage in Python and Ruby, and so on. These statistics show that the subset remains representative enough for evaluating realistic bug fixing capabilities while reducing evaluation cost.

Table 4: Feature distribution of Active-SWE-Extend instances across programming languages.

	Instance	Code Patch		Unit Test	
Language	# Num	# Files	# Hunks	# F2P	# P2P
Python	505	1.8	3.3	2.2	114.2
Go	327	2.2	9.5	1.3	26.8
Rust	153	1.7	12.2	1.5	28.9
PHP	250	1.7	3.0	1.3	40.5
Ruby	98	1.7	3.0	1.8	67.9
JS/TS	181	2.1	4.5	3.2	28.7
Java	76	2.2	4.2	1.1	0.2
C/C++	73	1.7	7.7	1.6	42.8

Table 5: Feature distribution of Active-SWE instances across programming languages.

	Instance	Code Patch		Unit Test	
Language	#Num	# Files	# Hunks	# F2P	# P2P
Python	103	2.1	3.2	3.5	118.9
Go	59	2.1	6.0	1.4	8.8
Rust	45	2.3	5.5	1.6	9.8
PHP	42	1.7	3.1	1.5	13.2
Ruby	29	1.4	2.1	1.8	62.9
JS/TS	65	2.2	4.4	3.3	26.6
Java	27	2.1	4.3	1.0	0.0
C/C++	30	1.9	4.4	1.9	32.6

B Related Work of Coding Agents

Over the last few years, LLM-based coding agents (Huang et al., 2023; Zhang et al., 2024; Islam et al., 2024) have emerged as the dominant paradigm in real-world software engineering, where LLMs are equipped with tools to interact with the development environments over multiple turns. In particular, recent advances on coding agents have demonstrated substantial performance gains over tool-free LLMs across various challenging tasks, including repository-level code repair and generation. With the rapid development of agent harnesses (Yang et al., 2024a; Wang et al., 2025), coding agents are able to leverage command-line tools to search and inspect repositories, edit source code, and execute tests, thus supporting various complex SWE tasks (Yang et al., 2026; Xia et al., 2025). In this work, we adopt diverse instructions and inputs to endow coding agents with the ability to construct the benchmark and evaluate the proactive bug fixing capability.

C Discussions on Real-world SWE Application

As discussed in the manuscript, Active-SWE focus on evaluating proactive bug fixing capabilities within files pending review. Such a setting corresponds to a widely-used development workflow, where developers review recently modified files before opening a pull request. It is worth noting that the design of review scope could also serve as a subtask of proactive bug fixing for the whole repository. In other words, our setting naturally supports this broader scenario by decomposing the repository-level task into successive file-level tasks that collectively cover the codebase.

To further verify this claim, we conduct additional experiments on two stable and mature repositories, *i.e.*, the pflag and Click repositories, each containing ten recorded bugs as verifiable evaluation targets. Specifically, we compare two settings using SOTA coding agent, *i.e.*, GLM-5.2, to verify the rationale for our design: i) *global review*, which instructs the coding agent to proactively fix bugs across the entire repository in a single run; ii) *iterative local review*, which requires the coding agent to review a few source files, *e.g.*, two files, at a time until the entire repository is covered.

From the results in Table 6, simply performing global review across the entire repository barely identifies historically recorded bugs, while our design, *i.e.*, iterative local review, substantially improves bug localization and resolved rates. Such results suggest that existing coding agents struggle to find and resolve recorded bugs when directly asked to review the entire repository. In contrast, the task formulation in Section 3.1 evaluate proactive bug fixing capability within specified review scope and could be extended to the scope of the whole repository by decomposing global review into successive local review tasks.

Table 6: Comparison of global review and iterative local review on recorded bug fixing.

Repository	Setting	LR	LP	Resolved
pflag	Global review	0.0	0.0	0.0
	Iterative local review	10.6	23.8	20.0
Click	Global review	0.0	0.0	0.0
	Iterative local review	4.7	33.3	10.0

D MORE EXPERIMENTAL RESULTS

D.1 EXPERIMENTAL RESULTS UNDER SIMPLE AND HARD SETTINGS

In the manuscript, we have conducted comprehensive experiments on curated Active-SWE, which contains 400 proactive bug fixing tasks. Here, we provide more experimental results on the 1,663 tasks in Active-SWE-Extend. Specifically, for reproducible and accessible evaluation, we evaluate several open-source models on Active-SWE-Extend in Table 7-8.

From the results, one could have the following observations and conclusions: i) for recorded bug fixing, LLMs still achieve limited localization scores and resolved rates under the simple and hard settings, suggesting that LLMs struggle with both needle-in-a-haystack bug localization and subsequent repair generation; ii) for potential bug discovery, although the review scope contains a few recorded bugs, LLMs are able to reveal additional potential bugs; iii) compared with the simple setting that involves one recorded bug, LLMs rarely resolve multiple recorded bugs simultaneously under the hard setting, further highlighting the difficulty of comprehensive proactive bug fixing; iv) under the hard setting, even when LLMs fail to resolve all recorded bugs, they may fix a subset of them, suggesting that LLMs tend to perform incomplete bug fixing.

Table 7: Performance Comparisons of different SOTA models under the simple setting.

Model	Recorded			Potential		
	LR	LP	Resolved	Count	TV	Revealed
DeepSeek-V4-Pro	15.2	16.0	9.4	1.7	77.0	61.3
Kimi-K2.6	17.8	14.6	13.8	2.1	73.8	56.5
MiniMax-M2.7	10.5	13.6	5.1	1.5	42.9	36.2
Qwen3.5-397B	11.5	14.6	6.2	1.5	55.7	44.6
Qwen3.5-122B	11.4	14.0	6.0	1.5	48.1	38.5
Qwen3.5-35B	8.6	11.0	5.2	1.4	28.8	22.7

Table 8: Performance Comparisons of different SOTA models under the Hard setting.

Model	Recorded				Potential		
	LR	LP	Res.	Sub Res.	Count	TV	Revealed
DeepSeek-V4-Pro	7.9	9.5	0.4	5.2	2.2	82.1	62.3
Kimi-K2.6	10.4	8.1	1.2	6.5	2.9	69.0	56.7
MiniMax-M2.7	3.7	6.8	0.4	2.0	1.9	55.2	45.6
Qwen3.5-397B	5.4	7.6	0.8	3.2	1.5	60.3	46.8
Qwen3.5-122B	3.5	5.7	0.4	2.2	2.0	48.0	36.5
Qwen3.5-35B	5.1	7.2	0.8	2.6	2.2	29.8	17.1

D.2 Experimental Results across Programming Languages

We have carried out experiments across various bug categories in the manuscript. Here, we present more experimental results of proactive bug fixing across programming languages. As shown in Fig. 9, resolved rates vary substantially across programming languages, with stronger performance on PHP, Java for at least some models, while Python and C/C++ remain more challenging. For potential bug discovery, the revealed rates are generally higher across most languages, but Java and C/C++ still show weaker results for several models. Taken together, these conclusions suggest that recorded bug repair and potential bug discovery exhibit different language-specific patterns.

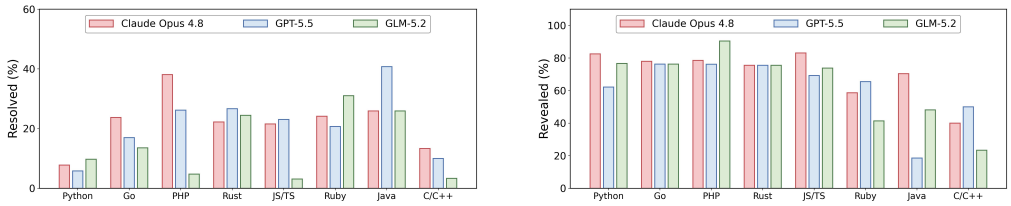

Figure 9: Proactive bug fixing performance across programming languages.

D.3 Experimental Results under Various Scaffolds

We conduct more experiments under different coding-agent scaffolds to examine the influence of hardness on proactive bug fixing performance. As shown in Table 9, different scaffolds lead to substantial performance variations, with Claude Code scaffold achieving better proactive bug fixing performance. Such results suggest that, beyond the underlying capability of LLMs, scaffold design could significantly affect proactive bug fixing performance.

Table 9: Performance comparisons under various scaffolds on a subset of Active-SWE.

Model	**Claude Code**		**SWE-agent**		**mini-swe-agent**	
	Resolved	**Revealed**	**Resolved**	**Revealed**	**Resolved**	**Revealed**
Claude Opus 4.8	21.0	73.0	11.0	25.0	12.0	58.0
GPT-5.5	19.0	63.0	15.0	32.0	18.0	46.0
GLM-5.2	12.0	64.0	10.0	28.0	11.0	44.0

D.4 Experimental Results on Cost Efficiency

We provide additional experimental results on cost efficiency, where cache tokens are excluded for a fair comparison. As shown in Table 10, proactive bug fixing exhibits a trade-off between performance and inference cost across various LLMs. For closed-source models, Claude Opus 4.8 achieves the strongest performance but incurs the highest cost, while Qwen3.7-Max provides competitive performance at a substantially lower cost. For open-source models, GLM-5.2 achieves the best overall performance with a relatively medium inference cost, offering a strong cost-performance trade-off. Overall, stronger proactive bug fixing performance often comes with higher inference cost, suggesting that effective bug discovery and repair require more extensive token usage for in-depth reasoning.

Table 10: Cost efficiency comparisons of different SOTA models. "Cost" denotes the average cost per instance.

Model	Cost Efficiency		
	Resolved	Revealed	Cost
Closed-Source Models			
Claude Opus 4.8	20.0	75.0	$32.33
Claude Sonnet 4.6	8.3	53.8	$11.95
GPT-5.5	18.5	65.5	$22.41
GPT-5.4	8.8	32.8	$5.23
Gemini-3.1-Pro	15.8	57.0	$7.74
Qwen3.7-Max	11.8	59.8	$12.92
Seed2.1	11.0	52.8	$11.22
Open-Source Models			
DeepSeek-V4-Pro	9.5	53.3	$2.41
GLM-5.2	13.0	69.0	$8.66
Kimi-K2.7-Code	14.8	47.8	$13.54
Kimi-K2.6	14.0	51.8	$7.53
MiniMax-M2.7	7.8	38.5	$2.14
Ring-2.6-1T	11.0	56.3	$4.18
Hy3	9.8	58.5	$0.47
Qwen3.5-397B	6.8	42.0	$1.14
Qwen3.5-122B	5.8	40.0	$0.74
Qwen3.5-35B	5.0	27.5	$0.54

D.5 Analytic Study on Edit Complexity

We conduct additional analysis to investigate the impact of edit complexity on proactive bug fixing. As shown in Fig. 10, the resolved rate for recorded bugs generally decreases as the generated patch spans more edits and longer patch lengths, indicating that more modifications do not necessarily lead to correct repairs for recorded bugs. As for potential bug discovery, the revealed rate shows a weaker and less monotonic relationship with edit complexity.

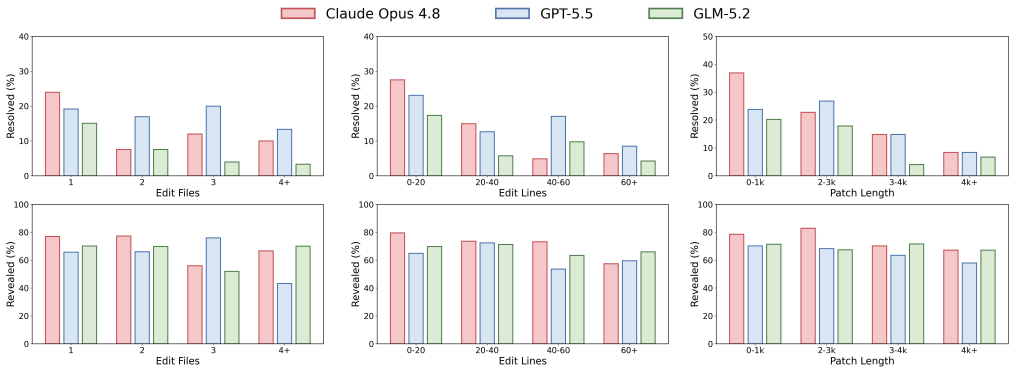

Figure 10: **Impact of edit complexity on proactive bug fixing performance.**

D.6 Analytic Study on Performance across Time Periods

We carry out more analysis on the performance comparison across different time periods based on the instance-wise merge time. From the results in Fig. 11, resolved and revealed rates fluctuate across years without a consistent temporal trend, suggesting that performance is not strongly correlated with time periods.

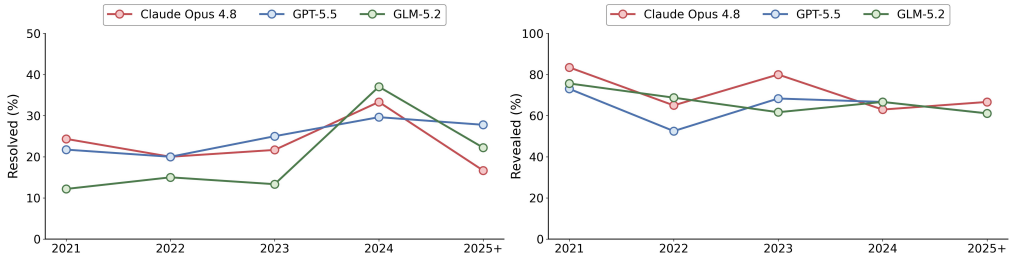

Figure 11: Proactive bug fixing performance across various time periods.

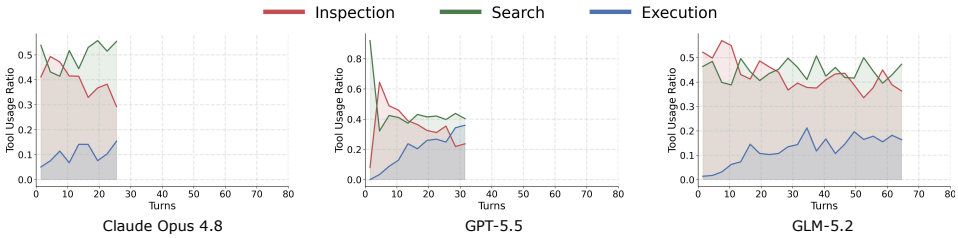

Figure 12: Tool-usage patterns and turn dynamics in reactive bug fixing.

D.7 Analytic Study on Reactive Bug Fixing

In the manuscript, we have conducted analytic study on the reasoning patterns of proactive bug fixing in Fig. 8. Here, we provide more analysis on the reasoning patterns of reactive bug fixing for comparison. According to Fig. 12, LLMs tend to perform search in the early stages and transition faster toward execution under the reactive setting, suggesting that issue reports provide debugging cues that narrow the search space and accelerate the fixing process. Compared with the proactive setting in Fig. 8, LLMs use fewer reasoning turns when issue reports are available, indicating that additional efforts are required for in-depth exploration and analysis without issue guidance.

D.8 Analytic Study on Broader Review Scope

In the manuscript, we define the review scope $\mathcal{F}_i$ as the files modified by the reference patch, ensuring that $\mathcal{F}_i$ always contains the recorded bugs. Here, we investigate how broader review scopes affect proactive bug fixing performance using the open-source SOTA model, *i.e.*, GLM-5.2. Specifically, we augment the original scope $\mathcal{F}_i$ with 2, 5, 10, or 20 additional source files, and further extend it to include all source files in the repository. As shown in Table 11, more files pending review generally degrade performance in both recorded bug fixing and potential bug discovery, highlighting the difficulty of proactive bug fixing across broader review scopes. In other words, even for a SOTA coding agent, proactive bug fixing becomes more challenging as the review scope grows.

Table 11: Analytic study on broader review scopes using GLM-5.2.

Scope	LR	LP	Resolved	Count	TV	Revealed
Original	18.7	21.8	15.0	1.7	85.0	70.0
+2 Files	19.9	20.5	15.0	1.5	85.0	72.5
+5 Files	17.3	20.4	15.0	1.9	85.0	65.0
+10 Files	16.5	13.4	12.5	1.5	77.5	65.0
+20 Files	9.8	7.2	10.0	1.8	80.0	60.0
All Files	4.0	1.4	2.5	3.7	47.5	15.0

D.9 Analytic Study on Different Judge Agents

To assess the sensitivity of potential bug evaluation to judge selection, we compare the evaluation results of two open-source SOTA models, Qwen3.5-397B and GLM-5.2, with those of human experts. From the results in Table 12, the judge agents driven by various LLMs and human experts yield similar evaluation results, suggesting limited sensitivity to the selection of judge agents.

Table 12: Potential bug evaluation with different judges.

Judge	Count	Revealed
Qwen3.5-397B	1.7	69.0
GLM-5.2	1.7	70.0
Human Expert	1.7	68.0

D.10 Analytic Study on the Overlap Between Potential and Recorded Bugs

To investigate whether revealed potential bugs extend beyond the recorded set, we ask human experts to examine the overlap between potential and recorded bugs. As demonstrated in Table 13, for all the coding agents, the revealed bugs consistently go beyond the recorded ones, suggesting that coding agents are able to discover potentially unrecorded bugs.

Table 13: Comparison between the validated potential bugs and the recorded bug. "Coverage" denotes whether at least one potential bug is validated. "Overlap" indicates the ratio that valid potential bugs matching recorded bugs.

Model	**Valid Bugs**	**Coverage**	**Overlap**
GLM-5.2	1.7	90.0	15.6
Claude Opus 4.8	1.2	85.0	48.0
GPT-5.5	3.0	90.0	9.6

E TASK TEMPLATES

In this section, we present more details on the task templates used throughout Active-SWE construction and evaluation, including the taxonomy curation template $\mathcal{T}_C$, the bug fixing template $\mathcal{T}_R$, the test generation template $\mathcal{T}_S$, and the judge template $\mathcal{T}_J$.

Taxonomy Curation Template

Role
You are a senior software engineer helping curate a benchmark for bug fixing.

Objective
Given a PR with its issue report, code patch, and test patch, your task is to decide whether it fixes a valid bug that falls into the target taxonomy. Exclude PRs whose root causes fall outside the taxonomy, such as feature requests, documentation changes, refactoring, test-only fixes, and performance optimizations without correctness impact. If the PR fixes a target bug, classify the bug into exactly one category from the target taxonomy.

Input

- **[Issue Report]**
- **[Code Patch]**
- **[Test Patch]**

Bug Taxonomy
[Bug Taxonomy]

Bug Taxonomy

Logic & Computation

- **Incorrect Condition / Inverted Logic**: incorrect comparison, boolean operator, negation, or branch condition in an existing conditional.
- **Incomplete Conditional**: missing a necessary branch, case, or variant within an existing `if`/`switch` structure.
- **Expression / Computation Error**: arithmetic, precision, operator-precedence, overflow, boundary, or off-by-one errors.

Reference & Data Flow

- **Incorrect Variable / Property / Key**: use of an incorrect variable, field, property, attribute, or dictionary key.
- **Incorrect Method / API Call**: invocation of an incorrect API variant, constructor, operator, or similarly named method.
- **Missing Propagation / Context Loss**: failure to propagate required parameters, configuration, metadata, state, or context across calls.

Domain-specific Workflow

- **Missing Business-Rule Guard**: absence of a required domain constraint, policy check, permission check, or feature guard.
- **Incomplete Registry / Allowlist**: missing a valid entry in an existing registry, mapping, allowlist, or configuration table.
- **Incorrect Operation Ordering**: operations executed in an invalid, priority-violating, or rule-violating order.

Data Processing & Type

- **String / Regex / Path Error**: incorrect string manipulation, regular-expression matching, delimiter handling, escaping, or path/URL construction.
- **Missing Parser / Grammar Rule**: missing token, grammar rule, syntax definition, or parser/compiler handling.
- **Type Mismatch / Implicit Conversion**: incorrect type assumptions, unintended implicit conversions, or missing type-specific handling.

Exception Safety

- **Missing Null / Existence Check**: missing checks for `null`/`None`/`nil`/`undefined`, absent keys, empty collections, or valid falsy values.
- **Unhandled Exception**: failure to catch, handle, recover from, or propagate an expected exception, panic, rejection, or error path.
- **Silent Exception**: exception handling that masks, swallows, fails to log, or fails to propagate an error.

State & Lifecycle

- **Incorrect State Transition**: incorrect state update, invalid transition, or violation of readiness/order dependencies.
- **Stale Data / Concurrency Safety**: stale cached or closure-captured data, race conditions, unsynchronized shared state, or deadlock-prone concurrency.
- **Resource Lifecycle Error**: improper acquisition, release, cancellation, cleanup, or lifetime management of resources.

Proactive Bug Fixing Template

Role
You are a senior software engineer performing proactive bug fixing.

Objective
Given the list of **files pending review**, your task is to identify the high-impact bugs and provide the corresponding fixes. Ignore all non-functional concerns, including style, formatting, and naming, to focus solely on functional correctness and system integrity.

Review Dimensions
The following are the primary bug categories to be reviewed:
[Target Bug Taxonomy]

Multi-Bug Review Protocol
The provided files pending review in the repo may contain multiple independent critical failures. You must execute an exhaustive code review:

- **No Early Exit**: Scan exhaustively. Do not stop after finding the first bug.
- **Surgical Isolation**: Document and fix each bug independently. Never combine unrelated patches into a monolithic refactor.

Input

- **[Files Pending Review]**

Test Generation Template

Role
You are a senior software engineer generating tests to reproduce revealed bugs.

Objective
You have developed code patch that may fix one or more independent bugs during the bug fixing task. Now, your task is to write exactly one test function for each bug fixed in the code patch. Each test function must satisfy the following requirements:

- Fails on the buggy codebase before the patch is applied.
- Passes on the fixed codebase after the patch is applied.

Do NOT write tests that verify normal or already-correct behavior. Every generated test must expose a bug fixed by the code patch.

Input

- **[Proactive Bug Fixing Template]**
- **[Predicted Code Patch]**

Judge Task Template

Role
You are a senior software engineer for validating revealed bugs and generated tests.

Objective
Your task is to evaluate whether the predicted code patch fixes valid bugs and whether the provided test patch truly reproduces these bugs.

Key Definitions

- **Effective Bug**: a real functional bug that is fixed by the predicted code patch. Cosmetic edits, formatting changes, pure refactoring, dead-code changes, test-only changes, and undocumented behavior changes are NOT effective bug fixes.
- **Effective Test**: a test that reproduces an effective bug. Tests are NOT effective if they only assert patched outputs, duplicate implementation logic, or check behavior unrelated to the bug.

Bug Taxonomy Mapping
Map each bug in the predicted code patch to one of the 7 taxonomies:

1. Logic & Computation
2. Reference & Data Flow
3. Domain-specific Workflow
4. Data Processing & Type
5. Exception Safety
6. State & Lifecycle
7. Others

Evaluation Protocol
You must execute a rigorous evaluation based on the following rules:

- **Bug Identification**: identify each distinct bug fixed in the predicted patch. Multiple fixes addressing the same root cause must be grouped into ONE effective bug.
- **Taxonomy Mapping**: map each identified effective bug to exactly one bug taxonomy.
- **Test Validation**: determine the effectiveness of each Fail-to-Pass (F2P) test and explicitly link it to the specific bug it verifies.

Input

- **[Proactive Bug Fixing Template]**
- **[Predicted Code Patch]**
- **[Predicted Test Patch]**
- **[Fail-to-Pass Tests]**

F CASE STUDY

In this section, we conduct case studies on proactive bug fixing tasks. As shown in Fig. 13-17, we provide successful and failed cases in recorded bug fixing and potential bug discovery.

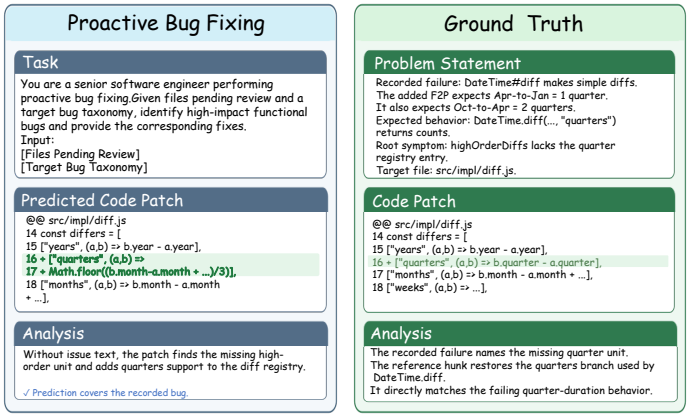

Figure 13: Successful recorded-bug fixing case where the predicted patch fixes the missing quarters branch.

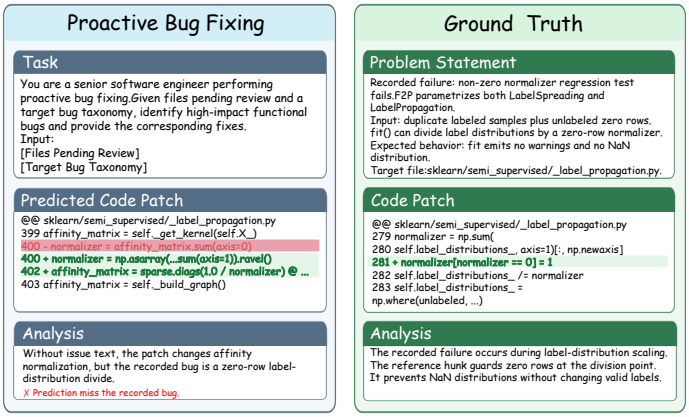

Figure 14: Failed recorded-bug fixing case where the predicted patch misses the label-distribution scaling failure.

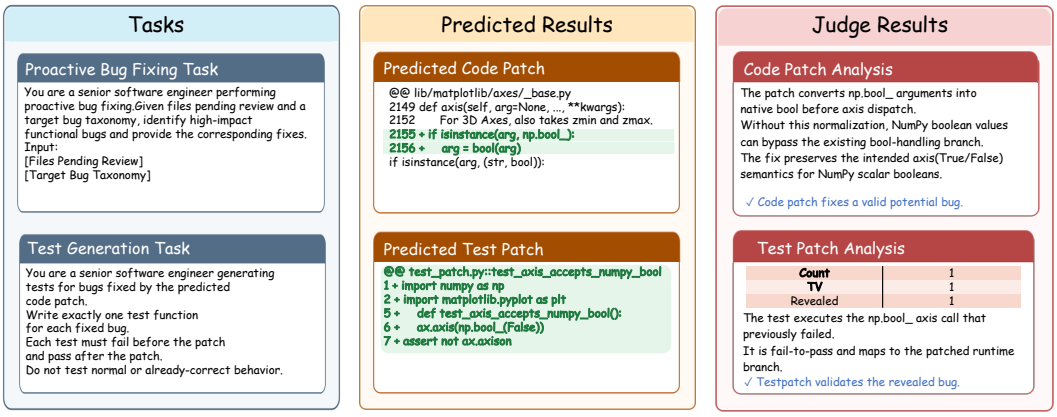

Figure 15: Successful potential bug discovery case where the generated tests validate the revealed bug.

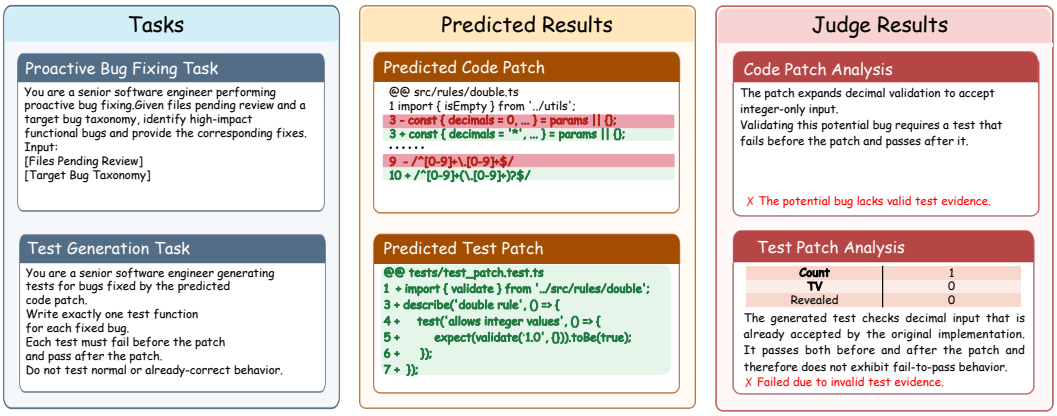

Figure 16: Failed potential bug discovery case where the generated tests don't exhibit fail-to-pass behavior.

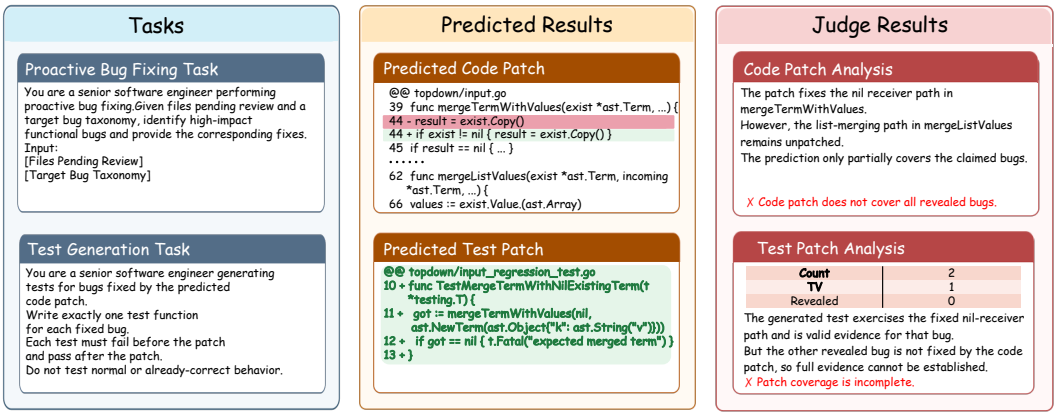

Figure 17: Failed potential bug discovery case where not all claimed bugs are supported by valid test evidence.