DO CODING AGENTS CATCH THEIR OWN BUGS? (THEY DON'T)

Anonymous authors
Paper under double-blind review

ABSTRACT

Real software development requires engineers to foresee and fix potential problems with their code that are not explicitly specified in requirements—such as preemptively considering vulnerabilities or other bugs from a code change. But current benchmarks often do not measure these future-looking consequences. We introduce **Foresight-SWE**, a benchmark for measuring whether coding agents can anticipate and address unstated problems during software engineering tasks. Foresight-SWE consists of 212 samples spanning 4 languages and 35 popular open-source repositories. Each sample contains two pull requests (PRs), the second of which resolves defects the first one introduced. To measure whether agents exhibit foresight, we define the **foresight resolve rate (FRR)** as the fraction of instances where an agent fully resolves the original task while also avoiding the defect identified by a later PR. We evaluate eleven large language models and find that FRR ranges from only 11.32% to 24.53%, largely failing to address problems introduced by the first PR change. This raises important questions about how models are trained and aligned in coding contexts, as well as how code review tools should be improved for more fail-safe coding. Foresight-SWE shifts evaluations in this direction.

1 INTRODUCTION

Coding agents are typically evaluated on fully specified software engineering tasks, where the complete requirements are explicitly provided (Jimenez et al., 2024; Deng et al., 2025; Xu et al., 2025; Badertdinov et al., 2026). Real software tasks, however, are often under-specified (Yang et al., 2026a), and recent work shows that coding-agent performance degrades under such conditions (Larbi et al., 2025; Laban et al., 2025). As coding agents take on more software engineering work, an important capability is forecasting defects that are not explicitly described in the task. Figure 1 illustrates such a case. A Pytest contributor changed virtual-environment detection to use Python's standard marker file, passed review, and merged the change. Twenty-three days later, a regression surfaced because conda environments do not create this marker, causing Pytest to incorrectly collect test files from installed packages.

Such latent defects are common and can impose substantial costs on maintainers and users. Django's issue tracker[1] shows 4,225 tickets opened between August 2021 and August 2026, of which 1,988 (47.05%) were classified as bugs. Of these, 947 were resolved as fixed, amounting to roughly one fixed bug every other day. These bugs involved a median of four participants and five comments, and took a median of 5.3 days from report to fix. For the 22% not fixed within a month, the median time to resolution rose to 121 days, during which the defect remained live in released code.

Prior work studies related forms of robustness in coding agents, but existing benchmarks either frame the task as proactive debugging in an existing codebase with defects, or restrict evaluation to a narrow class of defects. Active-SWE evaluates whether agents can proactively find bugs when prompted to search for them in a repository with defects (Li et al., 2026), while SecureVibeBench evaluates whether an agent, given an ordinary software task, can introduce similar security vulnerabilities as the original developer (Chen et al., 2026). However, the two benchmarks differ from Foresight-SWE. In Active-SWE, the defect already exists in the repository before the agent begins, whereas

[1] See here for Django's issue tracker (called Trac).

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.

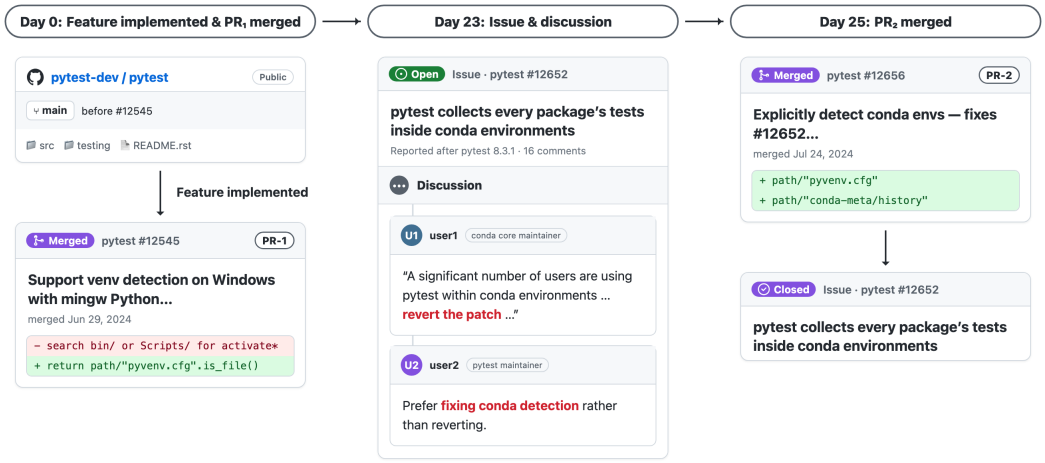

Figure 1: **Motivating Foresight-SWE.** Consider a real case where, on Day 0, a tested and reviewed PR_1 was merged. By Day 23, users had reported substantial downstream breakage, prompting an urgent discussion that also included a maintainer conda. On Day 25, a follow-up PR_2 repaired the problem and closed the issue. This timeline motivates Foresight-SWE, which measures whether coding agents can foresee consequential failures.

SecureVibeBench is limited to security vulnerabilities in C/C++ and relies on security-specific analysis tools for evaluation. Instead, we study how to create a scalable pipeline to evaluate an agent's own implementation using tests from later fixes of real-world PRs, allowing broader coverage across defect types and programming languages.

We introduce **Foresight-SWE** to measure whether coding agents can anticipate and address unstated edge cases while implementing a scoped task. To construct this benchmark, we mine PRs from open-source repositories and look for candidate pairs where one PR makes a change and a later PR fixes an edge case it missed. To identify valid pairs, we combine explicit maintainer references with information based on code history to reliably trace changes back to the earlier PRs. We then reconstruct executable repository states for both PRs, and validate each pair through test execution. We start from 11,609 candidate PR pairs, of which 492 remain after our construction and validation pipeline. From these validated pairs we introduce Foresight-SWE consists of 212 instances spanning four programming languages and 35 popular open-source repositories.

We find all tested models struggle with foresight compared to resolving the original task. On average, agents resolve the original task in 60.7% of instances. However, when considering **foresight resolve rate (FRR)**—the fraction of instances where the agent both resolves the original task and anticipates the defect later repaired by the follow-up PR—models achieve an average FRR of only 14.8%. When models are told the defect specifications, models can achieve a much higher average resolve rate of 91.2%, indicating the challenge is in recognizing the unstated defect. We find that proactive prompting can improve foresight (e.g., FRR increases by 11.32 percentage points for Claude Opus 5) but often trades off with the resolve rate of the original task. Surprisingly, in contrast, popular code review skills are not able to increase FRR. Together, these results suggest that current agents are often capable of fixing a defect once it is known, but are much less reliable at recognizing during implementation that their own change introduces a failure mode that will surface later.

Contributions. First, we develop a scalable pipeline that mines real-world repositories, inspects their code history to produce paired instances of an original software change and a later fix for a defect introduced by that change. Second, we introduce Foresight-SWE, a benchmark built from real PR pairs where an initial change introduces a defect that is repaired only by a later PR. Third, we show across eleven models that they are much worse at foresight than task completion and realistic prompt interventions are not necessarily a reliable way of counteracting lack of foresight.

2 Related Work

Agents deployed to find bugs. Recent works have focused on building an agent to search and fix defects. Active-SWE (Li et al., 2026) withholds the issue report from 1,663 tasks across eight languages and asks agents to find and fix bugs from a repository snapshot, reporting a best resolved rate of 20.0%. RepoAudit (Guo et al., 2025) isolates identification by never repairing, reporting 40 true positives at 78.43% precision. Live-kBench (Huang et al., 2026) supplies a crash report and reproducer and still reports a low equivalent-patch rate against the developer fix, with partial fix and latent bug surfacing as named failure modes. SWE-Review (Wang et al., 2026a) evaluates residual-defect detection once a candidate patch exists, and the security-lifecycle line (Wang et al., 2026b; Zhu et al., 2025) presupposes a known vulnerability and measures PoC construction or repair. In all of these prior works, the agent knows it is looking for bugs. We expect coding agents to actively forecast potential defects introduced by their changes as part of the implementation process, without necessarily deploying a separate approach as in prior work.

Defect prediction before implementation. Another relevant line of work predicts where defects will appear without generating code. Just-in-time Software Defect Prediction (JIT-SDP) estimates whether a code change is likely to introduce a defect using properties of the code change and its development history, such as its size, the files it touches, developer experience (Kamei et al., 2013; Zhao et al., 2023). Ticket-level prediction moves the defect prediction point earlier in the development lifecycle, estimating whether implementing a software ticket will introduce a defect before any code is written (Prova et al., 2025). ReDef extends JIT-SDP to LLM models, using them to classify individual code changes as defect-inducing or clean (Nam et al., 2026), while CodeFlowLM further studies which changed lines are responsible for the defect (Monteiro et al., 2025). Our work goes beyond predicting defect-prone changes by measuring whether an agent can identify an unstated failure mode and prevent the resulting defect while implementing the task itself.

Forecasting future outcomes. Forecasting has been studied broadly as a way to evaluate whether LLMs can reason about future outcomes from currently available evidence. For example, Forecast-Bench (Karger et al., 2025) and MIRAI (Ye et al., 2024) evaluate forecasting of future events from historical information. Fewer works have considered forecasting in software engineering use cases. SecureVibeBench (Chen et al., 2026) measures whether agents reproduce the same vulnerability introduced by the original developer for a limited set of C/C++ projects. In contrast, we develop a general-purpose pipeline to mine PRs where coding agents need to foresee and fix potential problems with their code, capturing a broader range of software defects, including silent incorrect behavior, across multiple programming languages.

3 Foresight-SWE

We outline the process of constructing Foresight-SWE, as illustrated in Figure 2. The pipeline identifies software changes where an original PR successfully addresses its stated task but leaves behind a defect that is repaired by a later PR. See Appendix A for more details on benchmark construction and validation.

3.1 Benchmark construction

Step 1: Scrape candidate PRs. In the scraping process, we want to discover pairs of PRs from the same repository such that PR_2 repairs a defect introduced by PR_1. To mine such pairs, we use two approaches. First, we use explicit author testimony, such as "introduced in PR_1," "broken by PR_1," or "follow-up to PR_1." Explicit references to the origin of a defect depend heavily on repository conventions and maintainer practices. Since many repositories do not contain explicit references, we also identify candidate pairs using changelogs, Trac tickets (for Django), and CVE/GHSA advisories, and subsequently use SZZ (Śliwerski et al., 2005) to infer PR_1. SZZ traces the lines deleted or rewritten by PR_2 back to the commits that introduced them. It then maps these commits to PRs and selects the PR responsible for introducing the largest number of affected lines as the candidate PR_1. While this scraping process isn't guaranteed to be exhaustive, it captures many bugs that were critical enough to be raised and resolved by humans in real-world codebases. The initial pool identified by this process had 11,609 candidates.

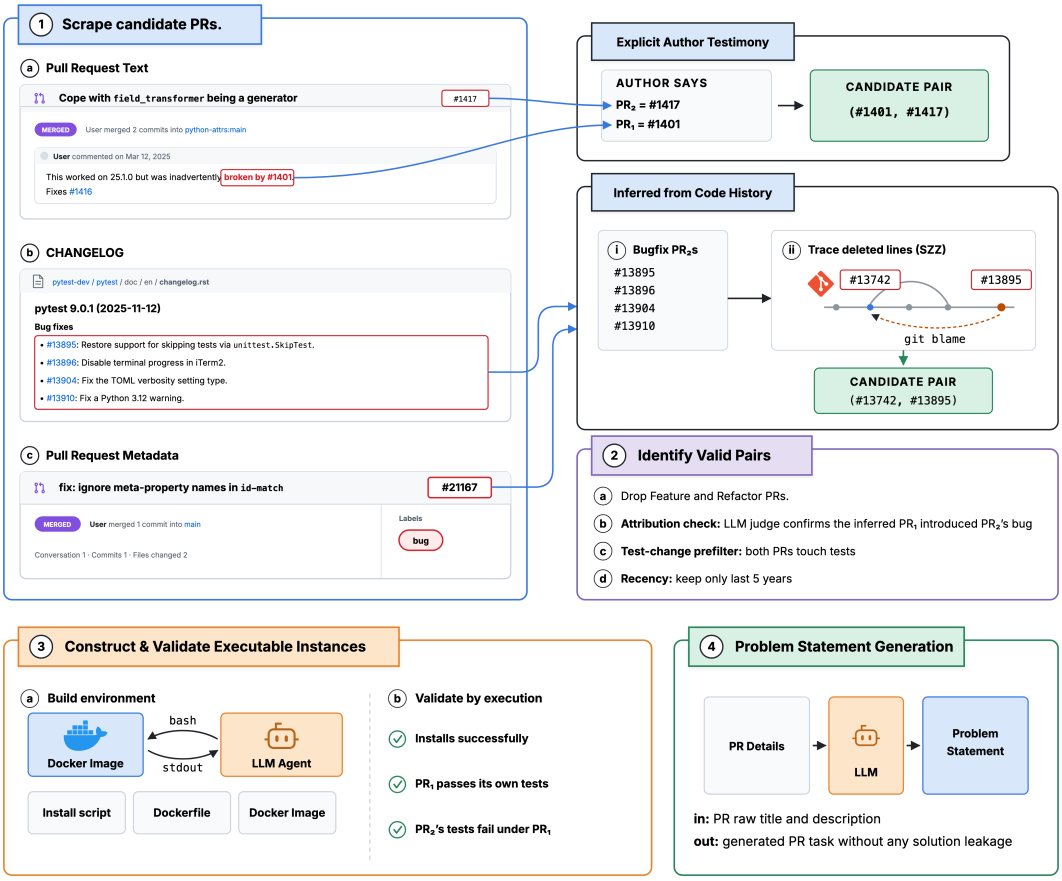

Figure 2: **Overview of Foresight-SWE Construction.** We scrape candidate (PR_1, PR_2) pairs from GitHub through either explicit author testimony or by inferring from code history. The candidates then pass through filtering to identify valid pairs, environment synthesis, and finally execution filtering.

Step 2: Identify valid foresight pairs. From the larger set of mined pairs, we want to ensure that it satisfies the following criteria: First, PR_2 must repair a genuine defect caused by PR_1, and that defect must be a reasonable target for anticipation. We therefore exclude cases where PR_2 primarily implements a new feature, performs a refactor, simply reverts PR_1, or entangles the defect fix with substantial unrelated changes. Second, when PR_1 is inferred using SZZ, the inferred PR must be the change that actually caused the defect. We apply two LLM-as-a-judge filters to enforce these criteria. We first judge whether PR_2 repairs an anticipatable defect associated with PR_1 (Badertdinov et al., 2026). We then judge only SZZ-inferred pairs and verify that the inferred PR_1 is indeed the PR responsible for introducing the defect. Importantly, we require both PRs to modify test files so that PR_1 can be scored on the original task and PR_2 can provide anticipation tests. Finally, we restrict both PRs to a five-year window from 2021-2026 to ensure repositories can be reliably rebuilt. After filtering, 2,262 pairs remain.

Step 3: Construct and validate executable instances. Evaluating each PR pair requires reproducing both the codebase and its execution environment, so we build isolated containerized environments for all selected PRs. For each instance, we reconstruct the repository at four states: the base and head of PR_1, and the base and head of PR_2. A PR's base is the repository state immediately before the PR is applied, while its head is the state after the PR's changes are applied. Following SWE-ReBench V2 (Badertdinov et al., 2026), we use an LLM agent to infer the installation and test commands needed to execute these states. We then replay each generated environment from scratch and validate the PR pair by executing its tests across the reconstructed states. PR_1 and PR_2's tests must fail at its base and pass at its head. The PR_2's test must also fail at PR_1 head, confirming that the defect was already present after PR_1. We retain only instances which satisfy these conditions, and produce machine-readable outputs, leaving 815 pairs.

	Attribute	Mean	Med.	Max
PR_1	Lines edited	367	99	10833
	Files edited	5.8	3	86
	Hunks	18.4	8	245
	F2P tests	20.3	3	666
	P2P tests	1423	151	28074
PR_2	Lines edited	49.8	15	1403
	Files edited	1.8	1	41
	Hunks	5.2	2	191
	F2P tests	4.5	2	194
Surface time (days)		158	88	1215

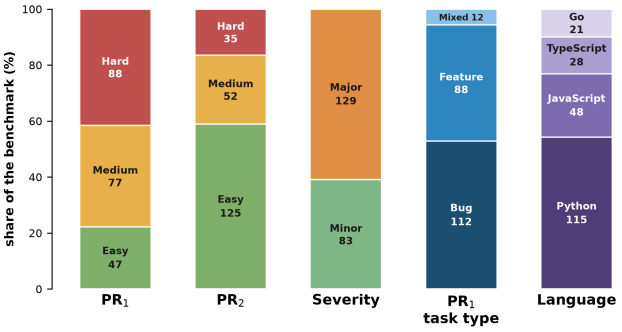

Figure 3: **Foresight-SWE dataset distribution.** The benchmark contains 212 instances from 35 repositories across four languages. (Left) Summary statistics of both PR_1 and PR_2. (Right) Distribution of problem difficulty, severity, task type, and programming language.

Step 4: Problem Statement Generation. Raw PR titles and descriptions are not always suitable as standalone benchmark tasks because they may be incomplete, rely on discussion context, or expose implementation details that bias an agent toward the developer's solution (Jimenez et al., 2024; Badertdinov et al., 2026; Xu et al., 2025). Following recent benchmarks (Badertdinov et al., 2026; Xu et al., 2025; Lam et al., 2026), we generate a self-contained problem statement for PR_1 using an LLM conditioned on its title, description, and source patch. We verify that the new problem statement (1) describes the intended behavior without prescribing the reference implementation or revealing the latent defect, and (2) is sufficiently specified for an agent to satisfy PR_1's tests. We discard instances that fail either criterion. After validation, 492 instances remain, from which we draw a stratified evaluation set of 212.

3.2 Dataset Overview

Foresight-SWE currently comprises 212 instances from 35 repositories across four languages, containing 958 anticipation tests, 4,307 task tests, and 301,659 regression tests (Figure 3). We observe that a median of 88 days separates PR_1 from PR_2 (mean 158, maximum 1,215), so these are defects that survived review and release.

Task difficulty. We annotate each instance with task difficulty and defect severity following the approach in SWE-ReBench V2 (Badertdinov et al., 2026). Difficulty reflects the expected implementation effort based on the amount of code change, logic complexity, and required domain knowledge. We classify tasks as easy if they are expected to take less than 15 minutes, medium if they require 15 minutes to 1 hour, and hard if they require more than 1 hour. The PR_1 tasks skew harder, with 47 easy, 77 medium, and 88 hard instances, while PR_2 skews easier, with 125 easy, 52 medium, and 35 hard instances. This indicates that the missed defects are often substantially easier to repair than the original tasks were to implement.

Defect severity. We classify severity into three categories, adapting common software severity schemes that consider user impact and the availability of workarounds (Saha et al., 2015; Lou et al., 2025).Critical defects cause severe failures on common paths, such as crashes, data corruption, or security bypasses. Major defects silently produce incorrect behavior for majority of users. Minor defects are explicit, narrowly triggered, low reach, easily worked around, or cosmetic. We use LLM-as-a-judge to annotate each instance. Among the 212 instances, 124 are classified as major, 83 as minor, and 5 as critical. And, in 129 of the 212 instances, the PR_2 test fails on an assertion rather than an exception, indicating a silent incorrect result rather than an explicit failure.

We provide the prompts used for annotation of task difficulty and defect serverity in Appendix B.

Table 1: **Agents rarely anticipate later defects.** Across eleven models, agents achieve an average resolve rate of 60.68%, but an average FRR of only 14.84%. Their average anticipation score is 26.39%, indicating that agents often capture only part of the behavior required to avoid the later defect. Bold represents the best-scoring model for each column's metric.

Model	Resolve Rate (↑)	Ant. Score (↑)	**FRR** (↑)
GLM-5.3 Flash	80.66	19.64	11.32
Qwen3.8 Flash Next	**91.98**	22.88	20.75
Qwen3.5-397B-A17B	21.70	26.44	7.08
DeepSeek-V4 Flash	58.49	**31.76**	18.87
Qwen3.8-27B	25.47	28.37	7.55
Qwen3.5-27B	18.40	26.00	6.13
GPT-5.6 Sol	84.43	23.98	18.40
GPT-5.6 Terra	74.53	23.91	15.09
GPT-5.6 Luna	74.06	28.01	16.51
Claude Sonnet 5	59.91	28.76	16.98
Claude Opus 5	77.83	30.57	**24.53**

4 RESULTS

We first describe the experimental set-up and then present the results, which comprise evaluations of 11 models on Foresight-SWE and subsequent experiments to explain where the gap in implementation and foresight arises from.

4.1 EXPERIMENTAL SET UP

Evaluation protocol. The agent receives the repository at PR_1's base together with the generated task description, with no information about PR_2 or the later defect. The agent produces a patch, which we evaluate on tests for the original task and on held-out tests for the defect later repaired by PR_2.

Metrics. Following Jimenez et al. (2024), we define the resolve rate as the fraction of instances for which an agent passes all PR_1 tests. These tests are divided into two categories: Fail-to-Pass (F2P) tests, which fail before PR_1 and pass after it, and Pass-to-Pass (P2P) tests, which already pass before PR_1 and must continue to pass after the agent's change. Together, they measure whether the agent implements the requested behavior without introducing regressions. To measure foresight, we define the **anticipation score** as the mean fraction of PR_2 F2P tests passed. These tests capture the defect later repaired by PR_2, so higher scores indicate that the agent handled more of the unstated behavior associated with that defect. Finally, we define our primary metrics, which is the **foresight resolve rate (FRR)**: the fraction of instances for which the agent passes all PR_1 F2P and P2P tests and all PR_2 F2P tests. FRR captures the desired behavior, measuring *both* how often an agent both resolves the original task and avoids the later defect.

Models. We evaluate 11 models, including 6 open-weight and 5 closed models. For closed-weight models, we use the providers' native coding-agent harnesses, such as Codex (Team, b) for OpenAI models and Claude Code (CC) (Team, a) for Anthropic models, since these are the interfaces through which users interact mainly and are optimized for their respective models. Open-weight models generally do not have an equivalent standardized native harness, so we evaluate them under a common scaffold using mini-swe-agent (MSA) (Yang et al., 2024), a lightweight harness widely used in coding-agent evaluation (Yang et al., 2026b; Li et al., 2026; Xu et al., 2025). We provide additional budget and version details in Appendix C.

4.2 FORESIGHT PERFORMANCE ACROSS MODELS

Agents resolve the stated task far more often than they anticipate the later defect. Across the 11 models, the average resolve rate is 60.7%, while the average FRR is only 14.8% (Table 1). Even among stronger task-solving agents, foresight is much weaker. For example, Qwen3.8 Flash Next

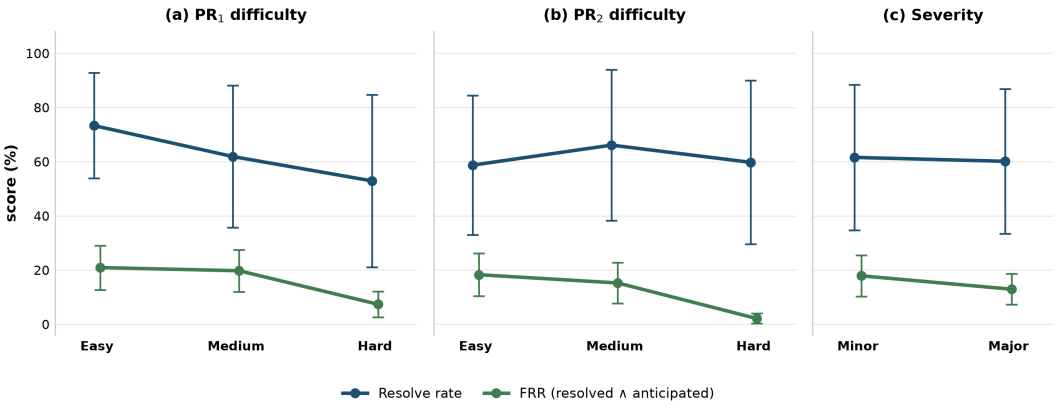

Figure 4: **Foresight decreases as task difficulty and defect severity increase.** We stratify the 212 instances along three axes. Across PR_2 difficulty (b) FRR degrade drastically but resolve rate stays mostly stable. Across PR_1 difficulty (a), all metrics degrade. Defect severity (c) has little effect on resolve rate, but FRR is 5.0 percentage points lower for major defects, with nine of eleven models performing better on minor defects. Error bars show ± 1 SD across models.

achieves the highest resolve rate at 92.0%, but an FRR of only 20.8%. Claude Opus 5 attains the highest FRR at 24.5%, meaning that no model both resolves the original task and avoids the later defect on more than a quarter of instances. Recall that 59% of PR_2 tasks are easy (Figure 3), and yet we still see low FRR. We provide qualitative evidence of such examples. For example, one task asks the agent to change the order of operations when constructing a class from its declared fields. The library originally validated the field ordering before running a user-supplied callback function, and users frequently supplied a generator for the callback function. In Python, a generator output can only be read once, a second attempt to read it yields no values. However, reordering the operations causes the generator output to be consumed twice. While the PR_2 fix is a single line change that wraps the generator output in a tuple, all eleven models failed this instance.

Foresight degrades with difficulty and severity. Stratifying results by difficulty (Figure 4) reveals the difference between task completion capabilities and defect anticipation. Across all three PR_2 difficulty, resolve rate across all models does not decrease monotonically. In contrast, FRR falls from 18.3% to 2.1%, all models score lower on hard defects than on easy ones, and, in particular, GPT 5.6 Sol scored zero on the hard difficulty defects. Across PR_1 difficulty, both task completion and foresight decline. We also observe average degradation in FRR of 5.0% when comparing minor and major defects, while the resolve rate shows no degradation. As part of annotating instances for severity, we also label instances as silent (the code runs and returns a wrong answer without any error trace) or loud (the code raises an error). Prior work has illustrated that silent failures often require deeper reasoning about intended behavior and greater codebase knowledge to identify (Lou et al., 2025; 2022). We find that agents achieve an FRR of 18.5% on loud defects, compared with only 12.5% on silent defects, suggesting that defects without an explicit failure signal are substantially harder to anticipate. We also observe variation in foresight performance across programming languages (Figure D.1). Per-model stratified results for difficulty, severity, and detectability are provided in Appendix D.1.

4.3 Is the gap identification or implementation?

A low foresight score may either be caused by the agent being unable to implement the fix, or it failed to foresee and identify the defect.

Setup. To separate defect identification from defect resolution, we evaluate five models on the PR_2 task. The agent receives the PR_2 task description and starts from PR_2's base commit, measuring how well it can repair the defect when the defect is explicitly specified. We compare this performance with the original foresight setting, where the agent receives only the PR_1 task and must both identify and avoid the defect, as measured by FRR and Foresight Score. For this experiment, we select the

Table 2: **Foresight is an issue of identification.** When the defect is explicitly specified, agents receive PR_2's task description and resolve 91.23% of instances on average. When the same defects must instead be anticipated from the original task, the models achieve an average FRR of only 19.91%. We conduct this ablation on a subset of 5 open-weight and closed models.

	Defect specified	Defect unspecified	
Model	Resolve Rate (↑)	Ant. Score (↑)	**FRR** (↑)
Qwen3.8 Flash	97.64	22.88	20.75
DeepSeekV4 Flash	85.85	31.76	18.87
GPT-5.6 Sol	94.81	23.98	18.40
Claude Sonnet 5	86.32	28.76	16.98
Claude Opus 5	91.51	30.57	24.53

Table 3: **Prompting improves foresight, but reduces task completion.** For a subset of models, we compare the baseline against proactive prompting, self-review, and their combination. The FRR improves, but the resolve rate drops steadily across different techniques. Bold represents the best setting for each model and column.

Model	Setting	Resolve Rate (↑)	Ant. Score (↑)	**FRR** (↑)
GPT-5.6 Sol	Baseline	**84.43**	23.98	18.40
	+ Proactive prompt	82.55	39.77	31.60
	+ Self-review	69.81	34.10	24.06
	+ Proactive & self-review	70.28	**44.90**	**32.55**
Claude Opus 5	Baseline	**77.83**	30.57	24.53
	+ Proactive prompt	75.47	44.00	32.55
	+ Self-review	68.40	46.38	**35.85**
	+ Proactive & self-review	64.62	**52.97**	**35.85**
Claude Sonnet 5	Baseline	**59.91**	28.76	16.98
	+ Proactive prompt	44.81	31.87	13.68
	+ Self-review	57.08	30.47	**18.40**
	+ Proactive & self-review	42.45	**32.81**	14.15

two open-weight models with the highest FRR in the main evaluation together with three frontier closed-weight models.

Results. Table 2 shows a large gap between fixing a known defect and anticipating it from the original task. When the defect is explicitly specified, models resolve 91.23% of instances on average. In the foresight setting, where the defect is not revealed, the same models achieve an average FRR of only 19.91% and a Foresight Score of 27.59%. This pattern holds across all five models, despite substantial differences in their ability to solve the original task. These results suggest that the primary bottleneck is identifying the latent defect rather than implementing its repair once identified.

4.4 Can prompting alone close the gap?

Section 4.3 shows that models can usually repair a defect once it is explicitly described. We therefore ask whether simple and realistic prompting-based interventions can elicit more foresight.

Setup. We test three interventions that target different stages of the agent's workflow. The first, proactive prompt, modifies the initial task prompt by explicitly asking the agent to consider edge cases and missing details that may not be stated in the problem description. The second, self-review prompt, adds a self-review step after the agent submits its patch. We invoke the harness's native review command, `/self-review` for Claude Code and `/review` for Codex, and ask the same agent to inspect its completed patch and repair any additional issues it identifies. The proactive prompt tests whether agents fail to search beyond the stated requirements during implementation, while self-review tests whether they stop once the explicit task appears complete. The third combines

both approaches, applying the proactive prompt during implementation followed by self-review after the patch is completed.

Results. Table 3 shows that both interventions can recover part of the foresight gap, but often at the cost of task completion. Proactive prompting raises FRR for GPT-5.6 Sol and Claude Opus 5 by 13.20 and 13.43 points, respectively, while reducing resolve rate by only 1.88 and 2.36 points. However, even the best-performing model, Claude Opus 5, reaches an FRR of only 35.85%, far below its 91.51% resolve rate when the defect is explicitly specified. Thus, prompting recovers only part of the foresight that the model is capable of once the defect is known. In contrast, the native review mechanisms in Claude Code and Codex increase FRR for all three models but reduce resolve rate in every case, and in case of GPT-5.6 Sol it decreases by 14.62 points. This suggests that broader review can uncover unstated defects while also introducing regressions in the original task. A qualitative example from `aiohttp` illustrates this behavior. PR_1 fixes a https defect where canceled requests could release a connection too early, but in doing so routes `close()` through `release()` logic. PR_2 later restores the distinction between releasing and closing a connection. Baseline Claude Opus 5 reproduces PR_1's approach, resolving PR_1 but missing the defect. With proactive prompting, Opus anticipates the PR_2 fix, but makes additional cleanup changes that cause PR_1 tests to fail.

5 Conclusion

As coding agents become increasinglyinto software development cycle, it is important to measure whether they can anticipate defects that unstated in the task. We develop a dataset construction pipeline that collects real-world GitHub pull requests spanning four programming languages and 35 popular open-source repositories. Across 11 models, agents achieve an average resolve rate of 60.7%, but an average foresight resolve rate of only 14.8%, showing that anticipating unstated defects remains challenging even for current state-of-the-art models. Our findings highlight the need to train future coding agents not only to satisfy stated requirements, but also to anticipate potential failure modes and produce more robust implementations.

Limitations and Future Work Foresight-SWE includes only defects that were eventually identified by maintainers, fixed, and successfully reproduced through execution. While this setup is sufficient for studying whether agents can anticipate unstated defects in the task description, it excludes defects that remain undiscovered by human maintainers. Our FRR metric rewards agents for handling latent defects but does not penalize unnecessary defensive changes. An agent that anticipates many defects may still produce low-quality or overly generalized code. Future work could therefore develop more robust evaluation metrics that jointly capture defect prevention and code quality, while penalizing excessive generalization. Finally, the benchmark can be expanded to cover additional languages and defect types.

6 AI use statement

We used AI-based tools (Claude Code and Codex) generating code to populate plots and tables. We also used AI tools for grammar checking and minor language editing.

7 Ethics statement

This work introduces Foresight-SWE, a benchmark for evaluating whether coding agents can anticipate and prevent latent software defects during implementation. We state that this work is intended for ethical and constructive purposes. While no immediate societal harms are identified, coding agents should be evaluated and deployed with adequate oversight to mitigate potential misuse and unintended software failures.

8 Reproducibility statement

We describe the dataset construction process in Section 3 and the experimental settings in Section 4. More details including prompts, and experimental setting are provided in the Appendix. We will also make our benchmark and codebase public in the future.

REFERENCES

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale, 2026. URL https://arxiv.org/abs/2602.23866.

Sicong Cao, Jinxuan Xu, Le Yu, Jing Yang, Xingwei Lin, Linlin Zhu, and Fu Xiao. Mas-szz: Multi-agentic szz algorithm for vulnerability-inducing commit identification, 2026. URL https://arxiv.org/abs/2604.24398.

Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, Xin Zhou, Xing Hu, and David Lo. Securevibebench: Benchmarking secure vibe coding of ai agents via reconstructing vulnerability-introducing scenarios, 2026. URL https://arxiv.org/abs/2509.22097.

R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992. doi: 10.1109/32.177364.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chengyu Hou, Chenhao Xu, Chenze Shao, Chong Ruan, Conner Sun, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Donghao Li, Dongjie Ji, Erhang Li, Fang Wei, Fangyun Lin, Fangzhou Yuan, Feiyu Xia, Fucong Dai, Guangbo Hao, Guanting Chen, Guoai Cao, Guolai Meng, Guowei Li, Han Yu, Han Zhang, Hanwei Xu, Hao Li, Haofen Liang, Haoling Zhang, Haoming Luo, Haoran Wei, Haotian Yuan, Haowei Zhang, Haowen Luo, Haoyu Chen, Haozhe Ji, Hengqing Zhang, Honghui Ding, Hongxuan Tang, Huanqi Cao, Huazuo Gao, Hui Qu, Hui Zeng, J Yang, JQ Zhu, Jia Luo, Jia Song, Jia Yu, Jialiang Huang, Jialu Cai, Jian Liang, Jiangting Zhou, Jiasheng Ye, Jiashi Li, Jiaxin Xu, Jiewen Hu, Jieyu Yang, Jin Chen, Jin Yan, Jingchang Chen, Jingli Zhou, Jingting Xiang, Jingyang Yuan, Jingyuan Cheng, Jingzi Zhou, Jinhua Zhu, Jiping Yu, Joseph Sun, Jun Ran, Junguang Jiang, Junjie Qiu, Junlong Li, Junmin Zheng, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Kexing Zhou, Kezhao Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Wang, Leyi Xia, Li Zhang, Liang Zhao, Lihua Guo, Lingxiao Luo, Linwang Ma, Linyan Zhu, Litong Wang, Liyu Cai, Liyue Zhang, Longhao Chen, MS Di, MY Xu, Max Mei, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Mingxu Zhou, Minmin Han, Ning Wang, Panpan Huang, Panpan Wang, Peixin Cong, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qingyang Li, Qinyu Chen, Qiushi Du, Qiwei Jiang, Rui Tian, Ruifan Xu, Ruijie Lu, Ruiling Xu, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runqian Chen, Runqiu Yin, Runxin Xu, Ruomeng Shen, Ruoyu Zhang, Ruyi Chen, SH Liu, Shanghao Lu, Shangmian Sun, Shangyan Zhou, Shanhuang Chen, Shaofei Cai, Shaoheng Nie, Shaoqing Wu, Shaoyuan Chen, Shengding Hu, Shengyu Liu, Shiqiang Hu, Shirong Ma, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, Shuying Yu, Songyang Zhou, Tao Ni, Tao Yun, Tian Jin, Tian Pei, Tian Ye, Tianle Lin, Tianran Ji, Tianyi Cui, Tianyuan Yue, Tingting Yu, Tun Wang, W Zhang, WL Xiao, Wangding Zeng, Wei An, Weilin Zhao, Wen Liu, Wenfeng Liang, Wenjie Pang, Wenjing Luo, Wenjing Yao, Wenjun Gao, Wenkai Yang, Wenlve Huang, Wenqing Hou, Wentao Zhang, Wenting Ma, Xi Gao, Xiang He, Xiangwen Wang, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaokang Zhang, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingchen Liu, Xingkai Yu, Xingyou Li, Xinyu Yang, Xinyu Zhang, Xu Chen, Xuanyu Wang, Xuecheng Su, Xueyin Chen, Xuheng Lin, Xuwei Fu, YC Yan, YQ Wang, YW Ma, Yanfeng Luo, Yang Zhang, Yanhong Xu, Yanru Ma, Yanwen Huang, Yao Li, Yao Li, Yao Xu, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Qian, Yi Shao, Yi Yu, Yichao Zhang, Yifan Ding, Yifan Shi, Yijia Wu, Yiliang Xiong, Yiling Ma, Ying He, Ying Tang, Ying Zhou, Yingjia Luo, Yinmin Zhong, Yishi Piao, Yisong Wang, Yixiang Zhang, Yixiao Chen, Yixuan Tan, Yixuan Wei, Yiyang Ma, Yiyuan Liu, Yonglun Yang, Yongqiang Guo, Yongtong Wu, Yu Wu, YuKun Li, Yuan Cheng, Yuan Ou, Yuanfan Xu, Yuanhao Li, Yuduan Wang, Yuehan Yang, Yuer Xu, Yuhan Wu, Yuhao Meng, Yuheng Zou, Yukun Zha, Yunfan Xiong, Yupeng Chen, Yuping Lin, Yuqian Cao, Yuqian Wang, Yushun Zhang, Yuting Yan, Yutong Lin, Yuxian Gu, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuxuan Zhou, Yuyang Zhou, Yuzhen Huang, ZF Wu, Zehao Wang, Zehua Zhao, Zehui Ren, Zekai Zhang, Zhangli Sha, Zhe Fu, Zhe Ju, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zheren Gao, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhixian Huang, Zhixuan Chen, Zhiyu Wu, Zhizhou Ren, Zhongyu Wu, Zhuoshu Li, Zhuping Zhang, Zian Xu, Zihao Wang,

Zihua Qu, Zihui Gu, Zijia Zhu, Zilin Li, Zipeng Zhang, Ziwei Xie, Ziyi Gao, Ziyi Wan, Zizheng Pan, and Zongqing Yao. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing, 2025. URL https://arxiv.org/abs/2501.18160.

Chenxi Huang, Alex Mathai, Feiyang Yu, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Eugene Wu, Kostis Kaffes, Junfeng Yang, and Baishakhi Ray. Outrunning llm cutoffs: A live kernel crash resolution benchmark for all, 2026. URL https://arxiv.org/abs/2602.02690.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013. doi: 10.1109/TSE.2012.70.

Ezra Karger, Houtan Bastani, Chen Yueh-Han, Zachary Jacobs, Danny Halawi, Fred Zhang, and Philip E. Tetlock. Forecastbench: A dynamic benchmark of ai forecasting capabilities, 2025. URL https://arxiv.org/abs/2409.19839.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation, 2025. URL https://arxiv.org/abs/2505.06120.

Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau sing Li, Jen tse Huang, Terry Yue Zhuo, and Michael R. Lyu. Swe-chain: Benchmarking coding agents on chained release-level package upgrades, 2026. URL https://arxiv.org/abs/2605.14415.

Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions, 2025. URL https://arxiv.org/abs/2507.20439.

Haobin Li, Ping Deng, Weizhong Qian, Liang Jiang, Zhenyu Huang, Mouxing Yang, and Xi Peng. Active-swe: Benchmarking coding agents for proactive bug fixing without issue reports, 2026. URL https://arxiv.org/abs/2608.04682.

Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 91–107, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/lou-demystifying.

Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. Deriving semantic checkers from tests to detect silent failures in production distributed systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 19–38, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-47-2. URL https://www.usenix.org/conference/osdi25/presentation/lou.

Yunbo Lyu, Jieke Shi, Hong Jin Kang, Ratnadira Widyasari, Junda He, Yuqing Niu, Chengran Yang, Junkai Chen, Zhou Yang, Julia Lawall, and David Lo. Agentszz: Teaching the llm agent to play detective with bug-inducing commits, 2026. URL https://arxiv.org/abs/2604.02665.

Monique Louise Monteiro, George G. Cabral, and Adriano L. I. OLiveira. Codeflowlm: Incremental just-in-time defect prediction with pretrained language models and exploratory insights into defect localization, 2025. URL https://arxiv.org/abs/2512.00231.

Doha Nam, Taehyoun Kim, Duksan Ryu, and Jongmoon Baik. Redef: Do code language models truly understand code changes for just-in-time software defect prediction?, 2026. URL https://arxiv.org/abs/2509.09192.

Daniele La Prova, Emanuele Gentili, and Davide Falessi. Anticipating bugs: Ticket-level bug prediction and temporal proximity effects, 2025. URL https://arxiv.org/abs/2506.14290.

Ripon K. Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E. Perry. Are these bugs really "normal"? In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pp. 258–268. IEEE Press, 2015. ISBN 9780769555942.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, pp. 1–5, 2005. doi: 10.1145/1083142.1083147.

Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. Llm4szz: Enhancing szz algorithm with context-enhanced assessment on large language models. *Proc. ACM Softw. Eng.*, 2 (ISSTA), June 2025. doi: 10.1145/3728885. URL https://doi.org/10.1145/3728885.

Anthropic Team. Claude code, a. URL https://claude.com/product/claude-code.

OpenAI Team. Introducing codex, b. URL https://openai.com/index/introducing-codex/.

Ruoyu Wang, Jierun Chen, Shaowei Wang, Chaofan Tao, Sidi Yang, Yuxin Jiang, Kim-Hui Yap, Lifeng Shang, Xiaohui Li, and Haoli Bai. Swe-review: Closing the loop on issue resolution with agentic code review, 2026a. URL https://arxiv.org/abs/2607.06065.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale, 2026b. URL https://arxiv.org/abs/2506.02548.

Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, Kepeng Lei, Yifan Yao, Xinping Lei, Wenqiang Zhu, Zongxian Feng, Han Li, Junqi Xiong, Dailin Li, Zuchen Gao, Kun Wu, Wen Xiang, Ziqi Zhan, Yuanxing Zhang, Wuxuan Gong, Ziyuan Gao, Guanxiang Wang, Yirong Xue, Mengtong Li, Mengfei Xie, Xiaojiang Zhang, Jinghui Wang, Wenhao Zhuang, Zheng Lin, Huiming Wang, Zhaoxiang Zhang, Yuqun Zhang, Haotian Zhang, Bin Chen, and Jiaheng Liu. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models, 2025. URL https://arxiv.org/abs/2511.05459.

Chenyang Yang, Yike Shi, Qianou Ma, Michael Xieyang Liu, Christian Kästner, and Tongshuang Wu. What prompts don't say: Understanding and managing underspecification in llm prompts, 2026a. URL https://arxiv.org/abs/2505.13360.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Programbench: Can language models rebuild programs from scratch?, 2026b. URL https://arxiv.org/abs/2605.03546.

Chenchen Ye, Ziniu Hu, Yihe Deng, Zijie Huang, Mingyu Derek Ma, Yanqiao Zhu, and Wei Wang. Mirai: Evaluating llm agents for event forecasting, 2024. URL https://arxiv.org/abs/2407.01231.

Z.ai. Glm-5.3-flash: Frontier intelligence, flash cost. URL https://z.ai/blog/glm-5.3-flash.

Yunhua Zhao, Kostadin Damevski, and Hui Chen. A systematic survey of just-in-time software defect prediction. *ACM Comput. Surv.*, 55(10), February 2023. ISSN 0360-0300. doi: 10.1145/3567550. URL https://doi.org/10.1145/3567550.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. Cve-bench: A benchmark for ai agents' ability to exploit real-world web application vulnerabilities, 2025. URL https://arxiv.org/abs/2503.17332.

APPENDIX

A BENCHMARK CONSTRUCTION

A.1 IDENTIFY VALID FORESIGHT PAIRS

A.1.1 PR PAIR VALIDITY JUDGE

Pair Validity Template

Evidence
Repository: **[Repo]**
PR1's task (what the earlier change set out to do): <<< **[Pr1 Task]** >>>
PR2's description (the fix for the missed edge case): <<< **[Pr2 Description]** >>>
PR2's fix patch (the gold anticipation fix): <<< **[Fix Patch]** >>>
PR2's test patch (the HELD-OUT tests — fail before PR2, pass after): <<< **[Test Patch]** >>>
Human discussion (PR + linked issues/tickets): <<< **[Discussion]** >>>
Objective evidence (gap, held-out test ids, changelog lines): <<< **[Evidence]** >>>

Weigh everything together
No single source is authoritative — piece the whole story together from ALL of the above. Different facts surface in different places: a false-positive vs a genuine miss is often clearest from the fix patch + tests; a revert/softening or the reason for it often only appears in the discussion or changelog; the descriptions frame intent. Synthesize, then extract the two facts.

Fact 1 — eligibility: is PR2 a genuine, anticipatable missed-edge-case?

- "clean" — PR2 fixes a genuine BUG: an edge case of PR1's behavior that produced a defect (wrong output, crash, silent wrong result) on some input. A real user-facing defect an agent could have anticipated. This is the only KEEP class. SECURITY fixes count as clean: a change that closes an attack (weak/predictable randomness, injection, XSS, path traversal, auth/authorization bypass, CRLF/header injection, SSRF, ReDoS/DoS) is a real bug even if there is NO observable defect for legitimate users — security bugs are silent to the victim.
- "not_a_bug" — no user-facing defect AND not a security fix: pure refactor, rename/alias, docstring/docs, deprecation bookkeeping, internal hygiene, type-only change. (Say what non-bug purpose it served.) NB: "no observable defect for legitimate users" does NOT imply not_a_bug — if the change closes an attack, it is "clean" (see above).
- "new_feature" — PR2 ADDS new capability rather than fixing a defect in existing behavior; the held-out test exercises a NEW API/feature, not a missed edge case of what PR1 did.
- "unpredictable" — PR2 REVERTS or SOFTENS a change PR1 made INTENTIONALLY, for SOCIAL/POLICY reasons (community pushback, "too disruptive", deprecation timing) — not an objective technical defect. An agent cannot foresee a maintainer's later change of heart; the reversal reason usually appears in the discussion or changelog. (BUT: if PR2 reverts because PR1's intentional change had an OBJECTIVE defect visible from spec/code — e.g. its stricter validation rejected valid inputs — that is "clean", not unpredictable.)
- "weak" — a real but trivial/borderline defect, or you are genuinely unsure it is a meaningful edge case.

Fact 2 — task validity: are PR2's held-out tests a well-formed, FAIR task?
"A" if none of the issues below apply. Otherwise the FIRST applicable Bxx code. These flag when passing the held-out tests requires more (or other) than solving the stated edge case — read the test patch closely.

Code	Issue	Signal
B1	TEST_SUITE_COUPLING	tests span unrelated modules; a correct fix could fail on unrelated regressions
B2	IMPLICIT_NAMING	tests assert specific names / signatures / error-message strings / output values that come from the patch, not derivable from the problem
B3	EXTERNAL_DEPENDENCY	essential info lives in external URLs not quoted in the issue/PR
B4	AMBIGUOUS_SPEC	missing expected behavior, repro, or acceptance criteria
B5	PATCH_ARTIFACTS	the fix patch bundles UNRELATED changes that the held-out tests nonetheless require
B6	IMPLICIT_KNOWLEDGE	tests assert a specific approach/internals not stated or inferrable from the repo
B7	INLINE_TEST	a test is inlined in the fix (source) patch rather than the test patch
B8	VISUAL_REQUIREMENT	task needs images/screenshots/visual context (incl. markdown images) not readable as text
B9	NON_ENGLISH_TEXT	the problem statement is partially/fully non-English
B10	TEXT_FILE_TESTS	the test patch only changes plain-text fixtures (`.txt`/`.csv`/...), not code tests
B11	MULTI_ISSUE_PR	PR2 fixes multiple independent issues; ambiguous which change the tests require

Output (JSON only, NO keep/drop decision)

```
{
  "eligibility": {"value": "clean|not_a_bug|new_feature|unpredictable|weak", "
   evidence": "short quote"},
  "validity":    {"code": "A|B1|B2|B3|B4|B5|B6|B7|B8|B9|B10|B11", "evidence": "short
   quote or 'none'"},
  "reasoning":   "2-3 sentences: what PR2 fixes, why that eligibility, and any
   validity concern",
  "confidence":  0.0
}
```

A.1.2 SZZ-ATTRIBUTION JUDGE

SZZ Attribution Template

You are auditing an automated bug-attribution claim in the repository **[Repo]**.
An automated tool claims that an earlier pull request (PR1) introduced the defect that a later pull request (PR2) fixes. It reached that claim **mechanically**: it took the lines PR2 deleted or modified, ran `git blame` on them, and followed the blame back to PR1.
That procedure is frequently wrong. Blame finds whoever last *touched* a line, which is not the same as whoever *broke* it. Your job is to decide whether the causal claim actually holds.
Treat the claim as a hypothesis, not a fact. Nothing below is evidence that PR1 is the true introducer — PR1 is simply where the blame landed. Do not assume the link is correct because this prompt describes PR1 as the introducer. Judge it from the code.

Evidence
Repository: **[Repo]**
PR1 — what the earlier change set out to do: <<< **[Pr1 Task]** >>>
PR1's diff. **[Pr1 Scope Note]** <<< **[Pr1 Diff]** >>>
The connecting lines — lines PR2 deleted or modified, which `git blame` traces to a commit inside PR1. These are the entire basis of the tool's claim: <<< **[Blamed Lines]** >>>
Changes to those files since PR1 — [Intervening N] commit(s) touched the connecting lines' file(s) between PR1 and PR2. If one of these introduced the condition PR2 fixes, PR1 is not the origin: <<< **[Intervening]** >>>
PR2 — the fix: <<< **[Pr2 Description]** >>>
PR2's fix patch: <<< **[Fix Patch]** >>>
PR2's test patch (the tests that fail before the fix and pass after): <<< **[Test Patch]** >>>

Human discussion (PR comments and linked issues, both PRs): <<< **[Discussion]** >>>
Issue and PR numbers have been replaced with <PR-Number> throughout. That is deliberate — decide from the code and the described behaviour, not from cross-references.

How to decide
Everything turns on one question: **what did PR1's diff actually DO to the connecting lines?** Work through it in this order.

1. **Find the defect.** Read PR2's fix patch and its tests. What input or state produced the wrong behaviour, and what does the fix change?
2. **Find that same logic in PR1's diff.** Look at the connecting lines and locate them in PR1's change.
3. **Ask what PR1 did to them.** Did PR1 *write* the flawed logic — the wrong condition, the missing guard, the bad default — as new lines of its own? Did PR1 write those lines *correctly* for what existed at the time, with PR2 now handling a case PR1's change never covered or created? Did PR1 merely *move* the lines — a rename, a file move, reformatting, an import or type-annotation change, a lint or codemod pass? A PR1 that swept a large number of files is a strong hint of this.
4. **Check the change list against the defect before you accept.** You named the defect in step 1. Now read the commit subjects in the change list above and ask a single question of each: *does this describe work on the same behaviour, subsystem or symptom as the defect?* If any subject does, you may not choose A without saying why that commit is not the origin — name it in `later_change` and give the reason. This is the most common way this audit goes wrong: the culprit is sitting in the change list, described plainly, and the judge accepts PR1 without ever comparing the two. Scan subjects only; you are not being asked to reason about their contents.

A defect can have more than one contributing change. Judge whether PR1 is a **credible cause of this specific defect**, not whether it is the only possible one.

Fact — attribution
Choose exactly one code. The illustrations below are schematic, not real cases.

- "A" — AUTHORED. PR1's own diff introduces the logic PR2 corrects. The flawed condition, missing case, wrong default or absent guard is visible among the lines PR1 added or changed. **This is the only code meaning the attribution HOLDS.** *Example:* PR1 adds `if timeout:` to apply a socket timeout. PR2 changes it to `if timeout is not None`, because `timeout=0` was silently ignored. PR1 wrote the faulty condition -> A.
- "C1" — STALE_ORIGIN. PR1 authored the connecting lines, but they were correct as written and the defect arrived later. To use C1 you must **name the specific later change** from the list above (or show the flawed logic predates PR1 entirely). This is the most common way blame misleads: it reaches back to whoever originally wrote a line that only later became inadequate. *Example:* PR1 adds `def decode(self, data):` as a synchronous method, correct for every caller then. The change list shows a later commit "make decode() async"; PR2 restores the sync API. A specific later change is nameable -> C1.

A long change list is not itself C1. Many commits may have touched the file without touching the behaviour PR2 fixes. Conversely, an empty change list is strong evidence *against* C1: nothing happened in between, so PR1's code is what PR2 is correcting.

- "C2" — CARRIED. PR1 only relocated or reshaped the lines — a move between files, a rename, reformatting, import reorder, type annotation, or a lint/codemod sweep. The behaviour predates PR1; PR1 carried it across unchanged, so blame landed on PR1 by accident. A PR1 that swept a large number of files is a strong hint. *Example:* PR1 moves `_normalize()` from `utils.py` to `text.py` with its body unchanged, and reformats it. PR2 fixes a bug inside `_normalize` that was present before the move -> C2.
- "C3" — UNCLEAR. The evidence does not let you tell: PR1's diff does not show enough of the relevant logic, or the defect cannot be located in what you were given.

The line between A and C1 is **not** whether PR1 touched the code — it did, in both. It is whether PR1's change *created the flaw*, or merely wrote something that was fine at the time. Ask: given only what existed when PR1 landed, would a careful reviewer have called PR1's code wrong? If yes, A. If PR1's code was reasonable then and something later made it inadequate, C1.
Prefer a definite code when the evidence supports one. Use C3 when it genuinely does not — not as a hedge.

Output (JSON only)
Answer the three questions FIRST, then commit to a code. Emit ONE flat JSON object with exactly these six keys — no nested objects, no extra keys, nothing outside the object.

```
{
  "defect": "one sentence: what PR2 actually fixes",
  "pr1_role": "one sentence: what PR1 did to the connecting lines",
  "reasoning": "2-3 sentences working through steps 1-3 above",
  "later_change": "for C1: quote the commit from the change list that introduced the
   defect. Otherwise \"none\".",
  "code": "A|C1|C2|C3",
  "evidence": "short quote from PR1's diff or the connecting lines that decides it"
}
```

All six keys are required. Write `reasoning` before choosing `code` — the analysis is what the code follows from, not a justification added afterwards.

A.2 Construct and validate executable instances

Each instance must be reproduced at four different commits, and a single installation recipe is not sufficient across a repository's history. Over the time span we mine, projects often move test dependencies between files such as `setup.cfg`, and `pyproject.toml`. We therefore generate the environment separately for each commit using an LLM agent, which produces the installation command artifact used by SWE-ReBench V2 (Badertdinov et al., 2026). We use DeepSeek-V4-Flash (DeepSeek-AI et al., 2026) as LLM in a simple harness.

Installer Agent Template

System
{preamble}
GOAL: find the commands that install this project and its TEST dependencies, so that this exact command produces a JUnit XML file containing at least one <testcase>:
{accept_cmd}
*** FAILING TESTS ARE EXPECTED AND ARE NOT YOUR PROBLEM. ***
You are configuring an ENVIRONMENT, not fixing code. Some of the test files in this working tree were deliberately copied in from a LATER commit, so they test behaviour this commit does not have yet. They are SUPPOSED to fail. A run that reports "4 failed, 128 passed" is a COMPLETE SUCCESS – the tests executed, which is all we need. Report your config immediately when you see output like that.
You are done as soon as the command above RUNS the tests and writes the XML. You are NOT done only when tests pass, and you must never try to make them pass. Do not read or edit source files under src/, do not investigate an AssertionError, do not downgrade or pin a package to make a test green.
The ONLY failures that mean your job is unfinished are ones that stop tests from RUNNING at all:

- ModuleNotFoundError / ImportError for a missing dependency
- "no tests ran", an empty XML, or a usage error before collection
- a collection error caused by an uninstalled pytest plugin

Response format – read this twice
Your response must contain a THOUGHT section and then EXACTLY ONE bash code block holding ONE command (or commands joined with && or ——). Format it exactly as shown:

<format_example> THOUGHT: Your reasoning and analysis here

```
your_command_here
```

</format_example>
The block MUST be a markdown fence opened with three backticks followed by `bash` and closed with three backticks. Do NOT use XML-style tags such as <bash>...</bash>. Do NOT emit more than one block. Anything else is unparseable, your command will not run, and the turn is wasted.
Failure to follow these rules will cause your response to be rejected.
I will run the command and reply with its output. Rules: {toolchain_rules}

- SYSTEM PACKAGES ARE YOURS TO INSTALL. You are root with network access, so if a

build needs a compiler, a header, a library, or a toolchain the image does not have – `apt-get update && apt-get install -y <pkg>` and carry on. A vendored C or JS dependency may need building before the package will compile (for example a submodule under vendor/ with its own Makefile or package.json); `git submodule update --init --recursive` and that build are legitimate steps too.

- EVERY command needed to reproduce the environment must appear in your final

`install` list, including apt-get and any vendored build. The list is REPLAYED in a fresh container that has none of your changes – a step you leave out is a step that will be missing when it matters.

- Prefer making the project's REAL build work over disabling it. An env var that

skips the C extension (e.g. *_NO_EXTENSIONS=1) is a last resort, not a shortcut: it runs a different code path from the one the project ships and tests.

- Do not modify tracked files. Do not run `git checkout`, `git clean`, or `git commit`.
- Beware: a dependency that is itself a pytest PLUGIN may be incompatible with the

version of the project under test at this commit. If the test command starts failing after you install something, remove it.
When the acceptance command works, reply with EXACTLY ONE fenced json block and nothing else – same fence rules, three backticks and `json`, no XML tags:
{output_spec}

Task preamble (pytest runner)

You are configuring a Python project's test environment inside a Linux container.
You are root. The repository is already cloned at {repo_dir} and checked out at commit {commit} ({repo}). `uv` and `git` are installed. You have network access.

Toolchain rules (pytest runner)

- Inspect the repo's own declarations first (pyproject.toml [project.optional-dependencies]

and [dependency-groups], setup.cfg [options.extras_require], requirements*.txt, tox.ini). Different commits of the same repo declare these DIFFERENTLY.

- PREFER commands that read the repo at the CURRENT commit – `uv pip install -e '.[test]'`,

`uv pip install -r requirements/test.txt` – over listing package names by hand. A recipe that reads the repo stays correct at other commits; a hardcoded list does not.

- Create the venv at {venv} with: uv venv –python {pyver} {venv}
- Install with `uv pip install --python {venv}/bin/python ...`.

Output specification (pytest runner)

```
["Install": ["<Cmd 1>", "<Cmd 2>"], "Test Cmd": ["<Full-Suite Test Command>"]]
```

`install` must list ONLY the commands that were necessary and succeeded, in order, starting with the `uv venv` command. `test_cmd` is the general (no file selectors) command that runs the whole suite with per-test verbose output and no colour.
NOTE: `test_cmd` is PROVENANCE ONLY – nothing downstream executes it. The grader uses one canonical invocation per test framework, chosen by detecting the framework at this commit, so that a single JUnit parser serves every repo and every language. Your job is the ENVIRONMENT: make the acceptance command above work.

Task preamble (node runner)

You are configuring a JavaScript/TypeScript project's test environment inside a Linux container. You are root. The repository is already cloned at {repo_dir} and checked out at commit {commit} ({repo}). Node {pyver}, `corepack`, `npm` and `git` are installed, and you have network access.
USE THE PACKAGE MANAGER THIS COMMIT DECLARES, not the one you would pick. Read package.json's `packageManager` field first; if it is absent, infer from the committed lockfile – pnpm-lock.yaml means pnpm, yarn.lock means yarn, package-lock.json means npm, none means npm. Activate it through corepack, e.g. `corepack prepare pnpm@8.6.3 --activate`. The manager and the lockfile format migrate across this repo's history, so the commit decides, not the repo.
PREFER A FROZEN INSTALL – `pnpm install --frozen-lockfile`, `npm ci`, `yarn install --immutable`. The committed lockfile pins the whole transitive tree, which is the strongest reproducibility guarantee available here. Fall back to a non-frozen install ONLY if the frozen one fails (an old lockfile can reference a since-yanked package, which no amount of cleverness fixes).
DO NOT add, remove or upgrade a dependency to make the tests run, and do not edit package.json or the lockfile. The JUnit reporters (`jest-junit`, `mocha-junit-reporter`) are already installed GLOBALLY and must stay that way – installing one into the repo would mutate the lockfile and destroy the frozen install.

Toolchain rules (node runner)

- Inspect package.json FIRST: `packageManager`, `engines`, `scripts.test`, and which of

vitest / jest / mocha is in devDependencies. Then check which lockfile is committed. Different commits of the same repo answer these DIFFERENTLY – the framework and the package manager both migrate across the window.

- PREFER a lockfile-driven install (`pnpm install --frozen-lockfile`, `npm ci`,

`yarn install --immutable`) over naming packages by hand. It stays correct at other commits; a hardcoded list does not.

- If the repo is a workspace/monorepo, install at the ROOT. Tests are run per package

afterwards, but the dependency tree is resolved once at the top.

- There is NO venv here and nothing to create: `node_modules` lands in the repo, which

is correct and expected. Do not try to install outside the worktree.

Output specification (node runner)

```
["Install": ["<Cmd 1>", "<Cmd 2>"], "Test Cmd": "<Shell Template, Or Omit>"]
```

Task preamble (go runner)

You are configuring a Go project's test environment inside a Linux container.
You are root. The repository is already cloned at **[Repo Dir]** and checked out at commit **[Commit]** (**[Repo]**). Go **[Go Version]** and `git` are installed and you have network access. GOTOOLCHAIN=local is set deliberately: this image serves ONE planned Go version, so a go.mod asking for a newer toolchain must fail loudly rather than silently downloading one the plan never chose.

Toolchain rules (go runner)

- There is almost never anything to install. `go test` compiles from source, so the environment is just two warm caches: `go mod download all` fills $GOMODCACHE, and a `go build ./...` fills $GOCACHE so the first test run does not pay for it.
- If the build fails, the cause is usually a MISSING SYSTEM LIBRARY for a cgo package (sqlite, libgit2, libmagic), not a Go dependency. Install it with apt.
- Do NOT edit go.mod or go.sum, and do NOT run `go mod tidy`. That would change the commit's own dependency graph, which is the thing we are trying to reproduce.
- If a package needs a build tag to compile, note it — but prefer leaving the default build configuration alone.
- There is no venv and nothing to create.

Output specification (go runner)

```
{"install": ["<cmd 1>", "<cmd 2>"], "test_cmd": "<shell template, or omit>"}
```

`install` must list ONLY the commands that were necessary and succeeded, in order. For most Go commits that is one or two lines (`go mod download all`), and an empty list is a legitimate answer when the toolchain alone suffices.
The `test_cmd` rules are identical to the node runner's (above).
`install` must list ONLY the commands that were necessary and succeeded, in order.
`test_cmd` IS EXECUTED – it is not documentation. Include it ONLY if running the framework directly on a changed test file does not work. It is a SHELL SNIPPET run once per test file with two variables set:
$FILE the test file path, relative to the repo root $JUNIT the JUnit XML path it must write
OMIT `test_cmd` entirely when the changed file is itself a runnable test – the default invocation already handles that, and an unnecessary override is a liability.
Include it when the changed file is a FIXTURE: data that some other file turns into a test. Symptom: the run reports 0 tests even though the install is fine. Then find the file that actually contains the test code and write a snippet that derives it from $FILE, so ONE template works for every chain in this repo. Worked example – a repo whose tests live in `<suite>/test.<ext>` and whose fixtures sit in `<suite>/samples/<name>/`:
SUITE="$(dirname "$(dirname "$(dirname "$FILE")")")/test.ts"; \ npx –no-install vitest run "$SUITE" \ –reporter=junit –outputFile="$JUNIT"
Derive from $FILE with dirname/basename – do NOT hardcode one sample or one suite, or the recipe only works for the chain you happened to test.
DO NOT NARROW THE RUN TO THE CHANGED TEST (no `-t`, no `--testNamePattern`, no single sample). Run the whole file or suite that contains it and let us select afterwards. Narrowing looks harmless and quietly destroys the PASS_TO_PASS set: the only tests left running are the ones the PR changed, and those FAIL at the pre-fix commit by definition, so nothing is ever observed to pass and the chain ships with no no-regression signal. Measured on svelte – filtered gave 3 usable outcomes and p2p=0; unfiltered gave 2,995 and cost 19 extra seconds.

A.3 Problem statement generation

Task Synthesis Template

You are a senior engineer writing a pull request description.

Input
You will receive a code patch (diff) and the original description. Use it only to infer the problem and required behavior. Do NOT describe the implementation.

Goal
Produce a PR description that explains the problem and what correctness requires, without revealing details unique to this patch.

Rules

- Do NOT mention file names, symbols, line numbers, commit hashes, or code snippets.
- Do NOT describe algorithms, control flow, or data structures.
- Avoid code-level "changed X to Y" statements.
- High-level component references are allowed.
- If intent is unclear, state a brief assumption.

Format

- **Title** — one line.
- **Problem** — 2-3 sentences max.
- **Root Cause** — 1-2 sentences, conceptual.
- **Fix / Expected Behavior** — bullet list, max 5 bullets.
- **Risk & Validation** — 2-3 bullets total.

Style Constraints

- Prefer bullet points over paragraphs.
- No section should exceed 3 lines.
- Omit anything non-essential to understanding or reviewing the change.

Output
Return only the PR description.
—

Inputs
REPO: **[Repo]**
ORIGINAL DESCRIPTION

```
[Pr Description]
```

PATCH**

```
[Patch]
```

Statement Clarity Template

We have a dataset of GitHub issues from various open-source repositories. Each issue comes with a PR that successfully solves the issue described. Each PR consists of 2 parts: (1) code that resolves the issue (2) changes to the test files of the repository, which check whether the issue has been resolved.
We intend to use samples in this dataset as a benchmark for coding ability: For each sample, we give an engineer the issue text and ask them to write code to resolve the issue (without revealing the solution from the original PR). Then, we apply the test files from the original PR to their code and run the tests to check whether their solution passes.
Importantly, this setup assumes that:

- The issue description is sufficiently well-specified to understand what the problem is, and what the correct solution should look like.
- The tests are correctly scoped to the issue i.e. they correctly test for a solution exactly as described in the issue description, and do not test any other unrelated functionality.

In this task, you will help to check those assumptions and identify which issue + test samples are suitable for use in our benchmark.
You are now considering an issue from the **[Repo]** repository.
Please take a moment to read the issue description below.
IMPORTANT: Our setup will only provide the main issue text as shown below, so please answer the questions on the basis of only the text shown here.
<problem_statement> **[Problem Statement]** </problem_statement>
How well-specified is the issue text? Imagine that you are an experienced software engineer who has been instructed to create a PR that successfully resolves the above GitHub issue. You

have full access to the codebase, and can see the issue description as it is above. But you are not able to ask for clarification and would need to work exclusively from this information. Is the issue description well-specified enough for a meaningful attempt at a solution?
You should classify this issue based on the following options.

- 0: The issue is well-specified and it is clear what is required for a successful solution.

<example> Title: `parse_date` raises on ISO strings that carry a timezone offset. Problem: Calling `parse_date('2021-01-01T00:00:00+05:00')` raises `ValueError: unconverted data remains: +05:00`. It should instead return a timezone-aware datetime carrying the given UTC offset. Expected behavior: `parse_date('2021-01-01T00:00:00+05:00')` returns `datetime(2021, 1, 1, 0, 0, tzinfo=timezone(timedelta(hours=5)))`. Naive inputs such as `parse_date('2021-01-01')` already work and must keep returning a naive datetime unchanged. </example>

- 1: There are some blanks to fill in about the issue, but there is a sensible interpretation of what is required for a successful solution.

<example> Title: HTTP client should not retry on 4xx responses. The client currently retries every failed request up to its configured maximum. Client errors (4xx) will not succeed on retry, so they should be returned immediately instead of retried; only network errors and 5xx responses should be retried. Existing retry count and backoff behavior for the retryable cases should stay as they are. </example>

- 2: The issue is vague and there is room for ambiguity. It is unclear what a successful solution would look like.

<example> Title: Improve error messages for config parsing. The error messages produced when a configuration file is malformed are not very helpful. They should be made clearer so the user can understand what went wrong and fix their config. </example>

- 3: It is almost impossible to understand what you are being asked to do without further information.

<example> Title: Make the scheduler more robust. The scheduler sometimes behaves unexpectedly when the system is under load. We should make it more robust and reliable. See the discussion in the linked thread for the background and context. </example>
Please respond with a short explanation of your choice (minimum 100 characters) followed by the selected option number.</pre>

Statement Leakage Template

You are auditing an agent-facing PROBLEM STATEMENT to ensure it is a FAIR test of whether an agent ANTICIPATES an edge case.
Setup: an earlier PR (PR1) introduced/changed behavior in **[Repo]**. A later PR (PR2) then fixed an edge case PR1 missed. We give an agent PR1's problem statement, have it solve PR1 from clean intent, and score whether its solution passes PR2's held-out tests — i.e. whether it ANTICIPATED the edge case WITHOUT being told.
For this to be fair the statement must sit in a Goldilocks zone:

- state the GOAL so a competent solution is possible;
- NOT prescribe the specific (flawed) implementation PR1 used — else following the task faithfully FORCES the bug, so the task contradicts the anticipation (UNFAIR);
- NOT reveal the edge-case handling PR2's tests check — else the agent is simply told the answer (LEAKY, anticipation becomes trivial).

The edge case — ANSWER KEY (the agent must NOT be able to derive this from the statement)
PR2's fix (what the missed edge case was): <<< **[Anticipation Patch]** >>>
PR2's held-out tests (what "anticipation" is scored on): <<< **[Anticipation Test Patch]** >>>

PR1's actual change (the introducer — may embody the flawed mechanism)
<<< **[Pr1 Patch]** >>>

The PROBLEM STATEMENT under audit (what the agent sees)
<<< **[Pr1 Statement]** >>>

Decide the PRIMARY verdict

- "clean" — intent-level: a correct anticipating solution satisfies it, and it neither prescribes the flawed mechanism nor reveals the edge-case handling.
- "prescribes_mechanism" — dictates the specific implementation/approach that produces the missed edge case (a control-flow or data-structure choice PR2 had to undo), so faithfully following it forces the flaw.
- "leaks_anticipation" — states or clearly implies the CORRECT handling of the edge case PR2's tests verify (the thing the agent should have to DISCOVER on its own).
- "leaks_future_fix" — references the later fix / PR2 / the bug report / the specific failure symptom the held-out tests target.

Severity order if several apply: prescribes_mechanism > leaks_future_fix > leaks_anticipation > clean.
Be strict about "leaks_anticipation": any sentence that tells the agent how the edge-case-relevant cases must behave (e.g. "multi-result methods must still collect all matches", "empty input returns X", "must guard against null") is a leak — the agent should derive that.

Output (JSON only)

```
{
  "verdict": "clean|prescribes_mechanism|leaks_anticipation|leaks_future_fix",
  "evidence": "the exact phrase from the statement that triggers it, or 'none'",
  "reasoning": "2-3 sentences tying the statement to the specific edge case",
  "repair_hint": "how to rewrite to intent-only: what to remove, what to keep"
}
```

A.4 Dataset validation

More than 90% of our corpus relies on SZZ to identify PR_1. Although SZZ is widely used to identify bug-introducing changes, its line-level blame heuristic can produce incorrect attributions (Lyu et al., 2026; Cao et al., 2026; Tang et al., 2025). For example, a later refactoring may overwrite the original blame, causing SZZ to identify the refactor rather than the change that introduced the defect. A separate failure mode occurs when the PR_2 fix is implemented in a different file or function from the change that originally caused the defect. Because SZZ traces the history of lines modified by PR_2, it relies on overlap between the fix location and the causal change; when this overlap is absent, the inferred PR_1 can be incorrect. SZZ can only trace blame through lines that are modified or deleted by PR_2. If the fix consists entirely of newly added lines, there is no pre-existing line to blame back to an earlier change, so SZZ cannot infer a PR_1 and abstains.

To reduce contamination from incorrectly inferred PR_1s, we use a panel of LLM judges to filter candidate pairs before they enter the benchmark. Specifically, we use DeepSeek V4 Flash (DeepSeek-AI et al., 2026) and GLM 5.3 Flash (Z.ai). We further evaluate the reliability of this filtering setup against ground-truth PR_1s explicitly identified by maintainers in PR and issue discussions. We estimate a contamination rate of 7.3% (95% CI [4.2, 12.4]). Importantly, this estimate should be interpreted as a strict upper bound on contamination in the final benchmark: of the 252 SZZ-derived candidates considered in this analysis, only 28 survive the subsequent stages of our data-curation pipeline.

Dataset. A subset of fix chains includes an explicit human attribution identifying PR_1, typically stated by a maintainer in the associated PR or issue discussion. Our dataset contains 376 instances across 46 repositories and 4 programming languages, representing 2.8% of all mined instances. For these chains the true PR_1 is known independently of SZZ, which lets us measure both how often SZZ's inferred PR_1 is correct and how much of its error the judges remove.

Results. We run the SZZ algorithm on the 376 instances in our dataset. Of 376 chains it abstains on 114 and proposes a PR_1 for 262, of which 172 match the maintainer and 90 do not: a contamination

rate of 34.4% before running LLM-judge panel. The LLM-judge panel rejects 85.6% of the incorrect PR_1s (77 of 90) while discarding 10.5% of the correct ones (18 of 172). Contamination among retained chains falls from 34.4% to 7.8% (95% CI [4.6, 12.9]), at a yield of 89.5%

Further, validate the entire Foresight-SWE construction pipeline end-to-end by sampling 10% of all instances and ask a co-author who was not involved in the benchmark construction process whether each instance has a PR_2 that fixes a bug introduced by PR_1. The co-author found that 90.9% of validated instances contained such PR_2s and the 9% that didn't were debatably so.

B Dataset Annotation

B.1 Instance difficulty and severity annotation

We use three prompts in sequence to produce the reported difficulty and defect severity. The first identifies the functions and classes exercised by the anticipation tests. The second prompt assigns a difficulty label, based on estimated engineering effort, and task validity code. The difficulty definitions are given in Table B.1. Both prompts are taken verbatim from SWE-ReBench V2 (Badertdinov et al., 2026). The third prompt is our own and assigns the severity of the defect. The severity definitions of the defect are given in Table B.1. Inspired by ODC (Chillarege et al., 1992) we define three axis and based on their values annotate an instance as critical, major or minor. The first axis is impact which measures what is affected by the defect. The second axis is reach and it measures how common the triggering setup is among it's users and according to the repositories documentation. Lastly, detectability which measures whether the user sees an error or silently receives a wrong result, inferred from the held-out test's assertions. A fixed rule maps these facts to a tier (Table B.2). We use DeepSeek-V4 Flash (DeepSeek-AI et al., 2026) for all three prompts.

We use DeepSeek-V4 Flash DeepSeek-AI et al. (2026) to judge task validity and assign difficulty and defect-severity labels.

Table B.1: **Category definitions.** The two axes along which we stratify the benchmark. *Difficulty* is scored by an LLM judge from the amount of code change, logic complexity, and domain knowledge required, expressed as expected implementation time, and is applied to both the original task (PR_1) and the fix (PR_2). *Severity* are judged on PR_2 (the fix).

Axis	Category	Definition
Difficulty (PR_1 & PR_2)	Easy	Expected developer implementation time < 15 min.
	Medium	Expected implementation time 15 min–1h.
	Hard	Expected implementation time $>$ 1h.
Severity	Critical	Persisted data corruption, data loss, or a security bypass reachable from a default or mainstream setup.
	Major	Broken or wrong output on a mainstream path; or silent wrong output on a niche path; or corruption behind a niche trigger.
	Minor	Loud failure on a niche path; any failure needing a rare intersection of conditions; or degraded/cosmetic impact.

Table B.2: **Severity rule.** The judge extracts impact, reach and detectability and the tier is computed from them.

Impact \ Reach	Broad	Narrow	Rare
Corruption	Critical	Major	Major
Broken	Major	Major if silent, else Minor	Minor
Degraded / Cosmetic	Minor	Minor	Minor

Interface Extraction Template

You analyze code changes in pull requests. Your task is to extract information about changed interfaces and signatures that are explicitly checked in the tests.

Decision Process

1. Identify new functions or classes
2. Identify functions or classes whose signatures have changed significantly (arguments changed, not just indentation)
3. Use the test patch to see which of them, if any, are explicitly called in the new test code

Ignore as interfaces:

- Purely internal helpers (for example names starting with "_" and not used directly in tests)
- Type aliases, enums, simple data or config holders
- Constructors or trivial overrides that only support other documented interfaces
- Constants, flags, small config values, and simple data declarations
- Anything that is not explicitly called in the test patch

Output Format
Output must be wrapped in a single root tag:

```
<ANSWER>Method: <ClassName.method_name(self, key_params...)> Location:  Inputs: <
     important parameters, types or roles, and key constraints> Outputs: <return type or
      behavior, including important error conditions> Description: <1 to 2 sentences on
     what it does and when to use it>

Function: <function_name(key_params...)>
Location:
Inputs: <important parameters, types or roles, and key constraints>
Outputs: <return type or behavior, including important error conditions>
Description: <1 to 2 sentences on what it does and when to use it>

...repeat blocks for each interface...</ANSWER>
```

If there are no qualifying interfaces, output exactly:

```
<ANSWER>No new interfaces are introduced.</ANSWER>
```

Guidelines:

- Only include symbols that:
- appear in the production patch with a changed signature as new or clearly changed, and
- are directly constructed or called in **[Test Patch]**
- Keep descriptions technical and concise

—

Inputs
REPO: **[Repo]**

ISSUE_TEXT**

```
[Problem Statement]
```

TEST_PATCH**

```
[Test Patch]
```

GOLDEN PATCH (reference only):

```
[Patch]
```

Task Validity and Difficulty

You are classifying software engineering tasks for RL validity. Determine if a task is of high-quality or reflects environment preparation issues (B1-B11).

Classification Codes

Code	Category	Description
A	SOLVABLE	Problem is clearly specified, tests align with stated requirements
B1	TEST_SUITE_COUPLING	Correct fix may fail due to unrelated test regressions
B2	IMPLICIT_NAMING	Tests expect specific names, signatures, error message strings, or output values derived from the patch implementation but not specified in the problem statement
B3	EXTERNAL_DEPENDENCY	Essential info in external URLs not quoted in issue
B4	AMBIGUOUS_SPEC	Missing expected behavior, repro steps, or acceptance criteria
B5	PATCH_ARTIFACTS	Reference patch includes unrelated changes tests may require
B6	IMPLICIT_KNOWLEDGE	Requires domain knowledge not stated or inferrable from repo
B7	INLINE_TEST	Inline test in the patch
B8	VISUAL_REQUIREMENT	Task requires images/visualizations/visual UI context not captured in plain text — including inline markdown images (`![...](url)`) embedded in the issue or PR description
B9	NON_ENGLISH_TEXT	Task description contains non-English language that must be understood
B10	TEXT_FILE_TESTS	Test patch only modifies plain-text fixture files (e.g. `.txt`, `.csv`) rather than code; expected behavior is encoded as opaque text diffs, making it hard to understand what is being tested
B11	MULTI_ISSUE_PR	PR addresses multiple independent issues or bugs simultaneously, creating ambiguity about which change is required to pass the tests

Decision Process

1. Extract intent from ISSUE_TEXT: expected behavior, observed behavior, reproduction, acceptance criteria
2. Check test alignment: Do assertions verify stated requirements or introduce new ones?
3. Scan for B-category signals:

- B1: Tests span unrelated modules
- B2: String assertions for names, error messages, or specific output values not derivable from the issue; assertIn/assertEqual on literal strings present only in the golden patch
- B3: URLs to external docs/specs
- B4: Vague or missing acceptance criteria
- B5: Patch modifies unrelated files that are then required by tests
- B6: Tests assert specific approaches not mentioned
- B7: Inline test in the normal PATCH
- B8: Visual artifacts are required to understand/solve task — check for markdown image syntax `![...](url)` in ISSUE_TEXT, or references to screenshots/mockups that cannot be read as plain text
- B9: Issue/problem statement is partially or fully non-English
- B10: All changed files in TEST_PATCH are `.txt`, `.csv`, or similar non-code fixture files; no Python/JS/etc. test functions are added or modified
- B11: Problem statement or PR references/closes multiple separate GitHub issues; tests appear to cover two or more independent fixes

1. Classify: A if clean, B[1-11] for primary issue

Rubric: PR Category
Choose one or more categories that best describe the PR.
Allowed categories:

```
{
  "critical_bug", "major_bug", "minor_bug", "regression_bug", "edge_case_bug", "
     performance_bug",
  "security_bug", "integration_feat", "core_feat", "ui_ux_feat", "dev_ops_enh", "
     documentation_enh"
}
```

Rubric: Task Difficulty
Estimate task difficulty based on the amount of code changes required, logic complexity, and domain knowledge needed. Choose one level assuming expected time to implement: easy (<15 min), medium (15 min–1h), hard (>1h).
Allowed difficulty levels:

```
{
  "easy", "medium", "hard"
}
```

Output (JSON only)

```
{
  "reasoning": "3-5 sentences: (1) summarize issue intent, (2) assess test alignment,
     (3) note any B-category signals, (4) justify classification",
  "intent_completeness": "complete | partial | insufficient",
  "test_alignment_issues": ["list specific misalignments or empty if aligned"],
  "detected_issues": {
    "B1": false, "B2": false, "B3": false,
    "B4": false, "B5": false, "B6": false, "B7": false,
    "B8": false, "B9": false, "B10": false, "B11": false
  },
  "external_urls": ["extracted URLs if any"],
  "pr_categories": ["one or more from the PR category rubric"],
  "difficulty": "easy | medium | hard",
  "code": "A|B1|B2|B3|B4|B5|B6|B7|B8|B9|B10|B11",
  "confidence": 0.0
}
```

—

Inputs
REPO: **[Repo]**

ISSUE_TEXT**

```
[Problem Statement]

These are interfaces that you need to implement, they will be tested:
[Interface]
```

TEST_PATCH**

```
[Test Patch]
```

GOLDEN PATCH (reference only):

```
[Patch]
```

Defect Severity Template

You are a senior software maintainer characterizing a real bug for a research dataset. An earlier pull request (PR1) in **[Repo]** introduced or changed some behavior and missed an edge case; a later PR (PR2) fixed that missed edge case. Your job is to EXTRACT STRUCTURED FACTS about the bug PR2 fixed — one value per axis, each backed by a short quote from the evidence. Do NOT output a severity tier. Severity is COMPUTED from your axis values by a downstream function. Your job is to get the axes right, not to grade. Judge from the perspective of a USER of the library, between PR1 and PR2. This judge runs only on chains already confirmed to be genuine bugs, so every input IS a real user-facing bug.

Evidence
Repository: **[Repo]**
PR1's task (what the earlier change set out to do): <<< **[Pr1 Task]** >>>
PR2's description (human-written fix for the missed edge case): <<< **[Pr2 Description]** >>>
PR2's fix patch: <<< **[Fix Patch]** >>>
PR2's test patch (the held-out tests — fail before the fix, pass after; read the assertions to reason about the pre-fix behavior): <<< **[Test Patch]** >>>
Human discussion (PR + linked issues/tickets; each item carries a URL): <<< **[Discussion]** >>>

Objective evidence (gap, held-out test ids, changelog lines): <<< **[Evidence]** >>>

1. impact — the CONSEQUENCE to a user (not the mechanism)

- "corruption" — PERSISTED or irreversible wrong data (written to a DB / file / durable state), data loss, OR a security/authorization bypass. The dangerous, hard-to-undo class; the user trusts a wrong answer that sticks.
- "broken" — a documented capability produces WRONG or MISSING output, or stops working: a wrong value / URL / string; a false-positive diagnostic (the feature reports a wrong verdict); an exception where a result was owed; a feature so defeated it is unusable for its purpose (e.g. resume that forces a full re-download); a control-flow contract silently altered; OR a standard workflow halts (e.g. warning spam failing a zero-warning CI gate). Wrong but TRANSIENT / recoverable output is "broken", NOT "corruption".
- "degraded" — still produces CORRECT results and blocks no workflow, but worse: perf regression, non-fatal noise.
- "cosmetic" — output formatting, messages, listings.

Judge from the DOWNSTREAM consumer, assuming the least-informed one.

2. detectability — does the user OBSERVE it?

Reason from the held-out test's assertions about what the PRE-FIX behavior was:

- "silent" — pre-fix returns a WRONG VALUE (assertion on a value), or an EXPECTED EXCEPTION IS MISSING (the fix adds a raise / the error was swallowed).
- "loud" — pre-fix raises an UNEXPECTED exception / crash the user sees.

3. reach — how COMMON is the setup needed to trigger it?

Decompose the trigger into its conditions (from the test's arrange/act), then classify by the RAREST (limiting) condition — NOT by "is there a flag".

- "broad" — default settings, OR a mainstream usage pattern.
- "narrow" — a specific niche opt-in (obscure config/dialect/platform, or a minority API).
- "rare" — an intersection of $\geq$2 unusual conditions.

CRITICAL: opt-in $\neq$ narrow. A single opt-in to a COMMON setting is broad. (E.g. requiring `--import-mode=importlib` is NOT narrow — pytest recommends it as the future default; the limiting condition is whatever ELSE the test needs.) You MUST justify prevalence with cited evidence: a documented default/recommendation, the library's own tutorial usage, OR the count of independent reporters in the discussion. No evidence -> output "narrow" and set "reach_estimated": true.

4. security — an ORTHOGONAL yes/no tag (a bug can be security AND have any severity)

"is_security": true/false — does the fix close an attack (injection, XSS, path traversal, auth/authorization bypass, CRLF/header injection, weak randomness, ReDoS/DoS, SSRF, info leak)? NOTE: this is a tag ONLY — it does NOT change the severity. Judge impact/reach/detectability on the actual consequence regardless (a defense-in-depth hardening fix with no demonstrated exploit can still be low impact).

5. priority_evidence — REALIZED signals (record; they DO NOT set severity)

Version pins/downgrades, reaction counts, number of independent reporters, a maintainer hotfix release. These are urgency (IEEE-1044 Priority), used only to corroborate the reach estimate, never the tier. Quote them; mark realized=true.

6. mechanism — did PR1 break working behavior, or miss a case in new code?

- "regression" — something that worked before PR1 stopped (PR1 broke it).
- "incomplete_coverage" — PR1's new behavior was correct but missed a case in its own domain.

Output (JSON only, NO tier)

```
{
  "impact":        {"value": "corruption|broken|degraded|cosmetic", "evidence": "
   short quote"},
```

```
  "detectability": {"value": "silent|loud", "evidence": "the assertion you reasoned
   from"},
  "reach":         {"value": "broad|narrow|rare", "limiting_condition": "the rarest
   condition",
                    "prevalence_evidence": "doc default / tutorial / N reporters / or
    'none'",
                    "reach_estimated": false},
  "security":      {"is_security": false, "evidence": ""},
  "priority_evidence": {"signals": "pins/reactions/reporters/hotfix or 'none'", "
   realized": true},
  "mechanism":     {"value": "regression|incomplete_coverage", "evidence": "short
   quote"},
  "user_impact":   "1-2 sentences: what a user of the library experiences",
  "confidence":    0.0
}
```

C Additional Experimental Setup Details

Agent Harness	Version	Reference
mini-swe-agent	1.9.1	https://github.com/swe-agent/mini-swe-agent
Claude Code CLI	2.1.278	https://claude.com/product/claude-code
Codex CLI	0.155.0	https://openai.com/codex/

Table C.1: Agent harnesses used in this work.

Model	Steps / Turns	Cost limit	Timeout
GLM-5.3 Flash	250	$3	—
Qwen3.8 Flash Next	250	$3	—
Qwen3.5-397B-A17B	250	$15	—
DeepSeek-V4 Flash	350	$7.20	—
Qwen3.8-27B	250	$3	—
Qwen3.5-27B	250	$3	—
Claude Opus 5	250	$5	90 min
Claude Sonnet 5	250	$5	90 min
GPT-5.6 Sol	—	—	90 min
GPT-5.6 Terra	—	—	90 min
GPT-5.6 Luna	—	—	90 min

Table C.2: **Per model budgets.** Steps apply to mini-swe-agent and turns to Claude Code. Codex exposes neither a turn nor a cost limit, so it is bounded only by the wall-clock timeout. All models use reasoning effort as high except Qwen3.8 Flash Next which has only xhigh.

C.1 Proactive Prompt Template

The below prompt is attached at the end of the problem statement for each harness.

Proactive Prompt

<important>
The PR description above states the behavior that was asked for. It covers the intent of the request, but it may not cover every edge case that behavior has to handle. You should attempt to forecast any potential edge cases that your implementation may not cover, or edge cases introduced by your implementation.
Work those out yourself, from the codebase: read how the code you are changing is called, what its neighbours already handle, and which inputs and expected outputs the description

never names. Do this before you start, while you implement, and again before you submit — what you learn from the code as you go will change what counts as an edge case here.
One limit. Stay inside the scope of the requested behavior: cover the inputs of THAT behavior, and do not redesign or harden unrelated parts of the codebase.
Verify the cases you decide to handle with your own scratch script, not by changing the repository's tests.
</important>

D Additional Results

D.1 Stratified results by task difficulty, defect severity and defect detectability

Table D.1: **Per-model results stratified by task difficulty.** Resolve rate and FRR for all eleven models.

Model	PR_1 difficulty			PR_2 difficulty		
	Easy n=47	**Medium** n=77	**Hard** n=88	**Easy** n=125	**Medium** n=52	**Hard** n=35
Resolve rate						
Qwen3.8	95.7	97.4	85.2	91.2	98.1	85.7
GLM-5.3	76.6	80.5	83.0	73.6	90.4	91.4
DeepSeek-V4	68.1	59.7	52.3	55.2	69.2	54.3
GPT-5.6 Sol	93.6	81.8	81.8	80.0	90.4	91.4
GPT-5.6 Terra	85.1	79.2	64.8	76.8	71.2	71.4
GPT-5.6 Luna	87.2	76.6	64.8	72.0	76.9	77.1
Opus 5	85.1	74.0	77.3	74.4	88.5	74.3
Sonnet 5	76.6	57.1	53.4	56.8	65.4	62.9
Qwen3.5-397B	46.8	23.4	6.8	22.4	23.1	17.1
Qwen3.8-27B	53.2	27.3	9.1	25.6	28.8	20.0
Qwen3.5-27B	38.3	23.4	3.4	17.6	25.0	11.4
FRR						
Qwen3.8	27.7	31.2	8.0	28.8	15.4	0.0
GLM-5.3	17.0	14.3	5.7	12.0	15.4	2.9
DeepSeek-V4	21.3	29.9	8.0	21.6	23.1	2.9
GPT-5.6 Sol	21.3	23.4	12.5	24.8	15.4	0.0
GPT-5.6 Terra	25.5	16.9	8.0	20.8	9.6	2.9
GPT-5.6 Luna	23.4	23.4	6.8	20.8	17.3	0.0
Opus 5	38.3	26.0	15.9	28.0	30.8	2.9
Sonnet 5	21.3	20.8	11.4	19.2	19.2	5.7
Qwen3.5-397B	10.6	11.7	1.1	8.8	5.8	2.9
Qwen3.8-27B	12.8	10.4	2.3	8.8	7.7	2.9
Qwen3.5-27B	10.6	9.1	1.1	7.2	7.7	0.0

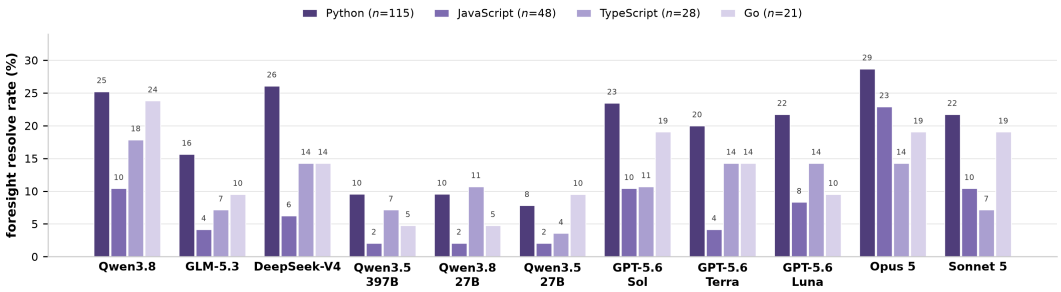

Figure D.1: **Foresight across programming languages.** We report the foresight resolve rate for all eleven models, stratified by repository language. Python is highest for every model and JavaScript lowest or tied-lowest for ten of eleven, with pooled rates of 19.1%, 13.4%, 11.0% and 7.6% for Python, Go, TypeScript and JavaScript. The Go and TypeScript strata contain 21 and 28 instances, so single instances move those bars by 4.8 and 3.6 points. JavaScript instances come largely from a single repository.

Table D.2: **Per-model results stratified by severity and detectability.** Resolve rate and FRR for all eleven models.

	Severity		Detectability	
Model	**Minor** n=83	**Major** n=129	**Loud** n=83	**Silent** n=129
Resolve rate				
Qwen3.8	91.6	92.2	92.8	91.5
GLM-5.3	79.5	81.4	80.7	80.6
DeepSeek-V4	59.0	58.1	61.4	56.6
GPT-5.6 Sol	88.0	82.2	90.4	80.6
GPT-5.6 Terra	77.1	72.9	83.1	69.0
GPT-5.6 Luna	74.7	73.6	78.3	71.3
Opus 5	77.1	78.3	78.3	77.5
Sonnet 5	62.7	58.1	65.1	56.6
Qwen3.5-397B	24.1	20.2	24.1	20.2
Qwen3.8-27B	25.3	25.6	26.5	24.8
Qwen3.5-27B	18.1	18.6	16.9	19.4
FRR				
Qwen3.8	20.5	20.9	25.3	17.8
GLM-5.3	13.3	10.1	15.7	8.5
DeepSeek-V4	24.1	15.5	24.1	15.5
GPT-5.6 Sol	18.1	18.6	20.5	17.1
GPT-5.6 Terra	18.1	13.2	18.1	13.2
GPT-5.6 Luna	18.1	15.5	20.5	14.0
Opus 5	33.7	18.6	32.5	19.4
Sonnet 5	22.9	13.2	21.7	14.0
Qwen3.5-397B	10.8	4.7	9.6	5.4
Qwen3.8-27B	9.6	6.2	10.8	5.4
Qwen3.5-27B	7.2	5.4	4.8	7.0