SWE-Bench+: Enhanced LLM Coding Benchmark

Anonymous Author(s)

Abstract

SWE-Bench has become one of the most widely used benchmarks for evaluating whether large language models (LLMs) can resolve real-world GitHub issues. However, despite its broad adoption, a systematic evaluation of the quality of the benchmark remains lacking. In this paper, we present *SWE-Bench+*, an enhanced benchmark framework that improves evaluation reliability by addressing two benchmark-quality risks in *SWE-Bench*: solution leakage in issue descriptions and weak tests that allow plausible but incorrect patches to pass. To construct *SWE-Bench+*, we first analyze 217 commonly resolved issues across three top-performing agents in the *SWE-Bench* leaderboard during our study: SWE-agent 1.0, OpenHands+CodeAct v2.1, and AutoCodeRover-v2.0), yielding 651 model-generated patches, and identify five quality-problem patterns under the two broader risks of solution leakage and weak tests. Based on this analysis, we develop two components: `SoluLeakDetector`, which identifies solution-leaking content for issue sanitization, and `TestEnhancer`, which strengthens test suites for more reliable patch validation. Our results show that 60.83% of the commonly resolved issues exhibit solution leakage, and 77.88% are problematic overall. After removing leaked information from issue descriptions, model resolution rates drop substantially, showing that leakage materially inflates reported performance. `SoluLeakDetector` achieves 80.45% accuracy on solution-leak detection, and `TestEnhancer` identifies plausible patches for 97.11% of weak-test issues while reducing average resolution rates by 27.00 percentage points on *Lite* and 36.27 percentage points on *Verified*. These findings show that existing SWE-Bench evaluations can substantially overestimate true issue-resolution performance and that *SWE-Bench+* provides a more rigorous benchmark framework for LLM-based software engineering agents.

Keywords

Benchmarking, Large Language Models, Issue Resolution

ACM Reference Format:
Anonymous Author(s). 2026. SWE-Bench+: Enhanced LLM Coding Benchmark. In *Proceedings of 3rd ACM International Conference on AI-powered Software (AIware '26)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The *SWE-Bench* dataset was created to systematically evaluate the capabilities of large language models (LLMs) in resolving software issues [14]. Given an issue description, the task for the LLM is to modify the corresponding codebase to produce a correct resolution. Each task instance is derived from a GitHub pull request associated with an issue and includes the issue text, the repository snapshot, evaluation test cases, and the developer-written gold patch. The original *SWE-Bench* contains 2,294 issues from 12 popular Python repositories. To support different evaluation goals, later variants such as *SWE-Bench Lite*[1] and *SWE-Bench Verified*[2] have been introduced as smaller and curated subsets intended to provide faster and more reliable assessment. Since its release in 2023, *SWE-Bench* has become the central benchmark and been widely used for evaluating LLM-based SE agents [2, 4, 12, 16, 24, 29, 32, 35, 37, 39, 40, 42]. However, despite its broad adoption, the validity of the *SWE-Bench* dataset remains underexplored.

In this paper, we introduce *SWE-Bench+*, an enhanced benchmark framework built on top of *SWE-Bench Lite* and *SWE-Bench Verified* for more rigorous evaluation of LLM-based software engineering agents. *SWE-Bench+* preserves the original real-world GitHub issue-resolution setting of *SWE-Bench*, while refining benchmark instances and their validation procedure to address two benchmark-quality risks: solution leakage in issue descriptions and weak tests. To construct *SWE-Bench+*, we analyze a subset of issues from the *SWE-Bench Lite & Verified* that are commonly resolved by three top-performing agents from the *SWE-Bench* leaderboard during our study: SWE-agent 1.0, OpenHands+CodeAct v2.1, and AutoCodeRover-v2.0. Specifically, we retain the instances for which all three agents were marked as successful in their evaluation logs, yielding 217 issues and 651 model-generated patches (217×3). Using issue descriptions, gold patches, model-generated patches, and model logs and trajectories, we identify five recurring quality-problem patterns, which fall into two broader benchmark-quality risks: solution leakage and weak tests.

To address these issues, we introduce *SWE-Bench+* through two refinement stages. First, we develop `SoluLeakDetector`, an LLM-based tool that identifies direct and hint-based solution-leaking content in issue descriptions to support leakage sanitization, and `TestEnhancer`, an LLM-based test-generation approach to strengthen validation against weak-test problems. We then evaluate both the benchmark-quality risks and the effectiveness of these two tools. Our results show that 60.83% of the commonly resolved issues exhibit solution leakage, and 77.88% are problematic overall. After removing leaked information from issue descriptions, model resolution rates drop substantially across the 132 leakage cases. `SoluLeakDetector` achieves 80.45% accuracy in identifying solution-leak issues, and `TestEnhancer` identifies plausible patches for 97.11% of issues with weak tests and reduces resolution rates by 27.00 percentage points on *SWE-Bench Lite* and 36.27 percentage points on *SWE-Bench Verified* across the three top-performing agents.

Overall, this paper makes the following contributions:

[1]https://www.swebench.com/lite.html
[2]https://openai.com/index/introducing-swe-bench-verified/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AIware '26, July 6–7, 2026, Montreal, Canada
© 2026 Association for Computing Machinery.
https://doi.org/XXXXXXX.XXXXXXX

- We introduce *SWE-Bench+*, an enhanced benchmark for improving reliability by sanitizing solution-leaking in issue descriptions and strengthening test suites.
- We design and evaluate `SoluLeakDetector` and `TestEnhancer`, two LLM-based tools that effectively detect solution leaks and identify plausible patches caused by weak tests, enabling more trustworthy evaluation of LLMs.
- We provide a systematic analysis of benchmark-quality risks in *SWE-Bench* to motivate and validate the design of *SWE-Bench+*.

Replication Package and the Dataset: https://doi.org/10.5281/zenodo.18005835

2 *SWE-Bench+* Overview

SWE-Bench+ is an enhanced benchmark built on top of *SWE-Bench Lite* and *SWE-Bench Verified* for more rigorous evaluation of LLM-based software engineering agents. It is motivated by benchmark-quality risks we identified in *SWE-Bench*, particularly solution leakage in issue descriptions and weak tests; the empirical details are presented in Section 3. It preserves the original real-world GitHub issue-resolution setting of *SWE-Bench* while refining both the benchmark instances and associated test cases. The construction of *SWE-Bench+* has two main stages. `SoluLeakDetector` detects direct and hint-based solution leakage in issue descriptions. `TestEnhancer` then generates and validates stronger tests for patch evaluation, producing stronger test suites.

Each instance from *SWE-Bench+* remains centered on the same essential software engineering task: given a natural-language issue report and the associated buggy repository state, an LLM-based system must produce a patch that resolves the issue and satisfies the benchmark's validation procedure. The key enhancement over *SWE-Bench* is that *SWE-Bench+* removes solution-leaking descriptions from problem statements and uses stronger tests to improve evaluation quality.

2.1 Building *SWE-Bench+*

Fig. 1 shows the construction pipeline of *SWE-Bench+*. We begin with the source benchmark pool formed by *SWE-Bench Lite & Verified*. Since our goal is to study benchmark risks that can affect reported agent success, we first collect the subset of issues that were commonly resolved by the three top-performing agents from the *SWE-Bench* leaderboard during the time of our empirical study: SWE-Agent 1.0, OpenHands+CodeAct v2.1, and AutoCodeRover-v2.0. Using the agents' evaluation logs, we retain the instances for which all three agents were marked as successful, yielding 217 instances in total. To determine which benchmark refinements were necessary for these source instances, we analyze each issue using its issue text, developer gold patch, model-generated patches, and the corresponding model logs and trajectories. Fig. 2 shows our analysis workflow. We compare the gold and model-generated patches by analyzing the corresponding changed files, while logs and trajectories provide additional evidence about the models' patch-generation process. To reduce bias in the comparison analysis, three authors independently review patch validity and resolve disagreements through discussion. This analysis revealed two benchmark-quality risks in the source set: solution leakage in issue descriptions and weak tests that fail to reject plausible but incorrect patches. *SWE-Bench+* addresses these two risks by applying two refinement stages shown in Fig. 1. First, `SoluLeakDetector` identifies issue descriptions that contain direct solutions or strong solution hints. For the identified leakage cases, we automatically remove the leaked information from the issue descriptions and then manually verify the sanitized descriptions to preserve the original issue context while reducing solution leakage. Second, `TestEnhancer` strengthens the original test suites to better distinguish correct fixes from plausible but incomplete or incorrect patches. The prompt templates for these two tools are shown in Figure 3. These two stages produce a benchmark that better reflects true issue-resolution ability.

2.2 `SoluLeakDetector`: LLM-based Solution Leak Detection

We develop `SoluLeakDetector`, an LLM-based technique that systematically identifies instances that contain either direct solution leaks or solution-hint leaks. `SoluLeakDetector` leverages *GPT-5-mini* to categorize instances into three distinct groups: (1) instances containing direct solution leaks, (2) instances with hint-based solution leaks, and (3) instances free from any form of solution leakage. The classification is performed using a three-shot prompting technique, where three clear examples of each category are provided to guide *GPT-5-mini* in categorizing *SWE-Bench* instances accordingly. The model is prompted with issue descriptions extracted from the root of the instance's pull request. Additionally, `SoluLeakDetector` identifies and extracts potential hints or direct solutions from the issue description, generating an explanation that details why the extracted fragment is classified as either a hint or a direct solution.

2.3 `TestEnhancer`: LLM-based Tests Enhancement

To strengthen validation, we develop `TestEnhancer`, an automated test generation and validation process designed to enhance the original *SWE-Bench* test patches by improving the coverage. As shown in Fig. 1, we begin by selecting specific attributes from the *SWE-Bench* dataset. Specifically, we extract the following attributes: *instance_id*, *gold_patches*, *test_patches*, and *base_commits*. The *base_commits* represent the commit IDs leading to the buggy version of each instance before the corresponding pull request (PR). To retrieve the buggy code patches, we trace these *base_commits* and extract the affected source code files from the repository. Next, we identify the specific files modified in the gold patch and apply regex-based pattern matching to extract the functions or methods that were changed.

After applying the gold patch and running the original tests, `TestEnhancer` analyzes coverage gaps in modified files and prompts GPT-5-mini to generate enhanced tests that follow repository conventions, exercise uncovered behavior, and include regression tests that fail on the buggy version but pass on the gold patch. Iteratively, it generates candidate tests, deduplicates overlaps, and runs them in isolated Docker containers to retain only those that meet the fail-to-pass criterion. Across rounds, it accumulates a curated set of tests that maximize behavioral coverage.

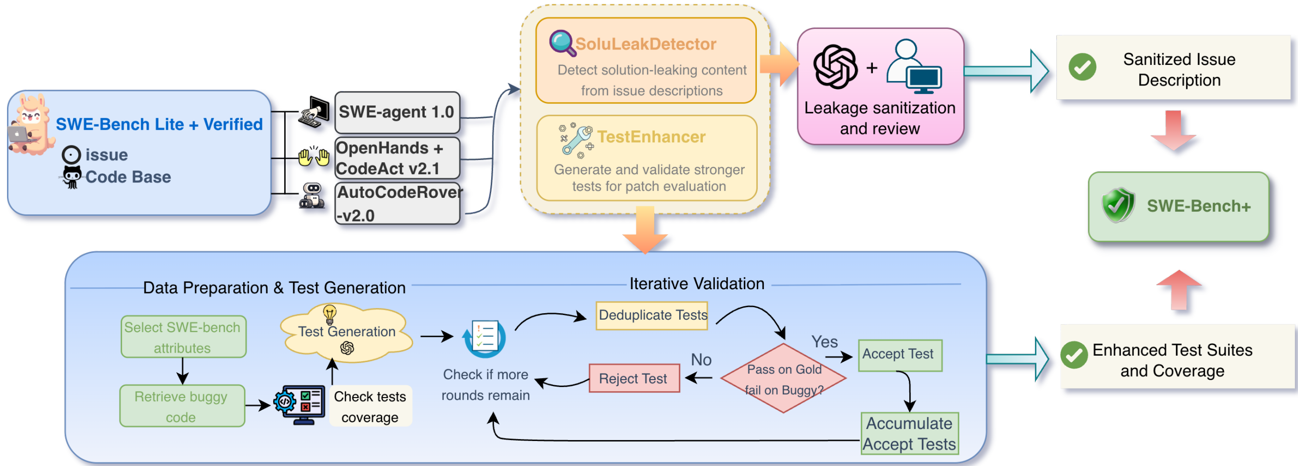


Figure 1: *SWE-Bench+* Build Pipeline.

Solution Leakage Detectiion Prompt:
You are a solution leakage detection expert.
Task: Analyze GitHub issue descriptions problem_statement and related comments hints_text for solution leakage.
Solution leakage occurs when:

1. The solution is **explicitly mentioned**, such as:
   - Code snippets providing a direct fix.
   - Step-by-step instructions leading directly to the solution.
2. The solution is **subtly implied**, for example:
   - Explanatory text that indirectly leads to the correct fix.
   - Comments hinting at how to resolve the issue.

Example 1: Description: I propose to add the following settings, with the following default values:
The default values maintain the current behavior.

```
LANGUAGE_COOKIE_SECURE = False
LANGUAGE_COOKIE_HTTPONLY = False
LANGUAGE_COOKIE_SAMESITE = None
```

These settings do not provide much security value, since the language is not secret or sensitive.
This was also discussed briefly here: https://github.com/django/django/pull/8380\#discussion_r112448195
Reasons for Change:

- Sometimes, auditors require them.
- I personally prefer to set them unless I have a reason not to.
- Browsers are increasingly enforcing HttpOnly and Secure, e.g.,
- https://webkit.org/blog/8613/intelligent-tracking-prevention-2-1/

Expected Output:

```
{
  "solution_leakage_detected": true,
  "reason": "The solution is explicitly provided in the description.",
  "extracted_solution": "LANGUAGE_COOKIE_SECURE = False, LANGUAGE_COOKIE_HTTPONLY = False, LANGUAGE_COOKIE_SAMESITE = None"
}
```

TestEnhancer Prompt
You are an expert software engineer code assistant tasked with generating additional unit tests for a Python source file and its corresponding test file.
TASK: Your objective is to add 30 regression tests that detect the bug, which is resolved by the provided patch.
GUIDELINES:

1. **Analyze the Code:** Examine the provided source code to understand its functionality, inputs, outputs, and core logic.
2. **Analyze the Patch:** Examine the provided patch to understand the bug it fixes in the source code.
3. **Identify Test Cases:** Develop a detailed list of test cases that will fully validate the provided patch.
4. **Add and Review Tests:** Integrate individual tests, ensuring they collectively cover all possible scenarios, including edge cases and exception handling.
5. **Maintain Consistency:** Ensure new tests are consistent with the existing test suite in terms of style, naming conventions, and structure. Assume new tests are part of the same suite if a test suite exists.

ADDITIONAL CONSTRAINTS:

- Return only valid Python in the YAML fields; do not produce syntax errors.
- Octal literals must use digits 0-7 only (e.g., 0o644). Do not generate invalid octal values like 0o800.

Source File: Here is the source file source file that you will be writing tests against. We have manually added line numbers to assist in understanding the code coverage. These line numbers are not part of the original code.
`source_numbered`
Patch: Here is the patch that is applied to the `source_file` to fix a bug.
`patch_content`
Test File: Here is the file `test_file` that contains the existing tests.
test conten
`test_content`
OUTPUT FORMAT: The response should be only a valid YAML object, without any introduction text or follow-up text.

Example Output:

~~~~
```yaml
language: Python
number_of_tests: ...
test_behavior: ...
test_code: ...
new_imports_code: ...
```
~~~~

Figure 3: Prompt templates, i.e., the left side is *SolutionLeak Detection* Prompt, and the right side is *TestEnhancer Prompt*

Table 1: Issue quality problems found among the 217 common fixed issues by the three agents

Quality problems	Numbers (percentage)	Root cause
Solution leak	70 (32.26%)	solution leakage
Solution hint leak	62 (28.57%)	solution leakage
Incorrect fixes	43 (19.82%)	weak tests
Different files/functions changed	22 (10.14%)	weak tests
Incomplete fixes	39 (17.97%)	weak tests

3 Why *SWE-Bench+* Is Needed

Building on the benchmark construction described in Section 2, we now present the empirical findings that motivate *SWE-Bench+*. We first analyze the benchmark-quality problems in *SWE-Bench*, then quantify the impact of solution leakage and weak tests, and assess the effectiveness of `SoluLeakDetector` and `TestEnhancer` in addressing them.

3.1 Quality Deficiencies in *SWE-Bench*

We analyzed 217 commonly resolved issues across the three agents, yielding 651 (217∗3) patches. An issue is considered problematic if it meets either of two mutually exclusive conditions: (a) *Solution-leak (direct or hint)*, or (b) *Weak-test-only*, where at least one patch is incorrect, incomplete, or touches different files/functions among the remaining issues. Under this definition, 132/217 issues (60.83%) exhibit solution leakage, 32/217 issues are weak-test-only, and 77.88% of the commonly resolved issues are problematic overall. Figure 4 presents the distribution of issues in *SWE-Bench Lite* (300 instances) and *SWE-Bench Verified* (500 instances). On *Lite*, 15.67% of all its issues exhibit solution leakage problems, while on *Verified*, the proportion is 22.6%, including both direct leaks and hint-based

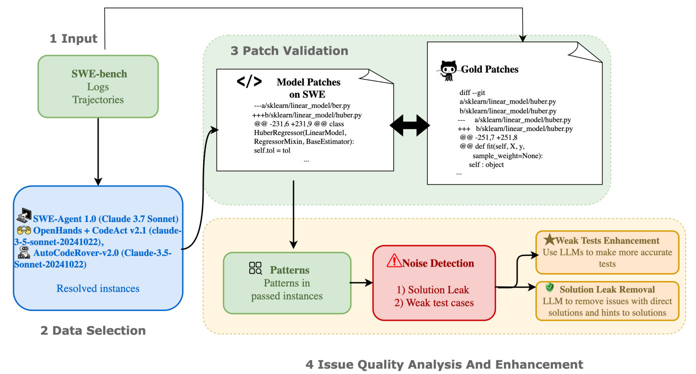
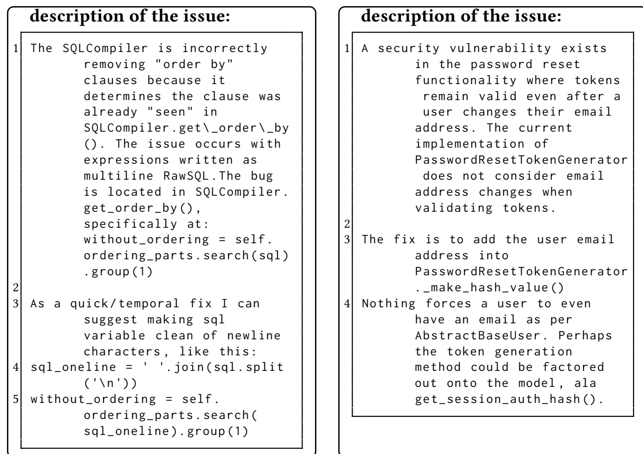


Figure 2: Overview of our analysis for *SWE-Bench* datasets

guidance. Additionally, 13.33% of all issues in *Lite* and 15.2% in *Verified* are identified as problematic due to weak test cases. To better understand these benchmark-quality problems, we categorize them into five patterns: two solution-leak-related and three weak-test-related, which are summarized in Table 1, with definitions, counts, and root causes discussed below.

1. Solution leak: refers to instances where the issue description substantially reveals the fix. In such cases, the problem statements may expose direct solutions about the intended changes, allowing the agents to reproduce solution content that is already present in



description of the issue:

```
1 The SQLCompiler is incorrectly
      "order by"
      clauses because it
      determines the clause was
      already "seen" in
      SQLCompiler.get\_order\_by
      (). The issue occurs with
      expressions written as
      multiline RawSQL.The bug
      is located in SQLCompiler.
      get_order_by(),
      specifically at:
      without_ordering = self.
      ordering_parts.search(sql)
      .group(1)
2
3 As a quick/temporal fix I can
      suggest making sql
      variable clean of newline
      characters, like this:
4 sql_oneline = ' '.join(sql.split
      ('\n'))
5 without_ordering = self.
      ordering_parts.search(
      sql_oneline).group(1)
```

description of the issue:

```
1 A security vulnerability exists
      in the password reset
      functionality where tokens
      remain valid even after a
      user changes their email
      address. The current
      implementation of
      PasswordResetTokenGenerator
      does not consider email
      address changes when
      validating tokens.
2
3 The fix is to add the user email
      address into
      PasswordResetTokenGenerator
      ._make_hash_value()
4 Nothing forces a user to even
      have an email as per
      AbstractBaseUser. Perhaps
      the token generation
      method could be factored
      out onto the model, ala
      get_session_auth_hash().
```

(a) A *Direct Solution Leak* example (django-11001)

(b) An example of *Hint Leak* (django-13551)

Figure 6: Side-by-side examples of (a) direct solution leak and (b) hint leak in problem statements.

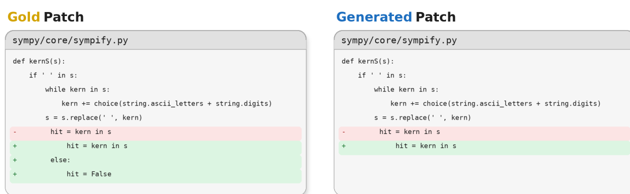


Figure 7: Incorrect fix generated by the model for sympy-19637

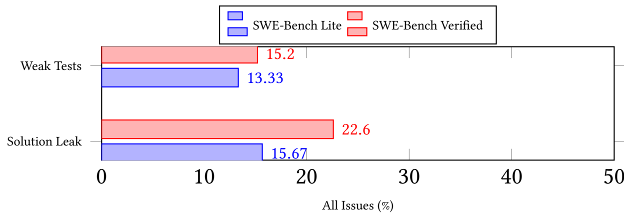


Figure 4: Distribution of problematic issues among all instances in *SWE-Bench Lite* (300 instances) and *SWE-Bench Verified* (500 instances)

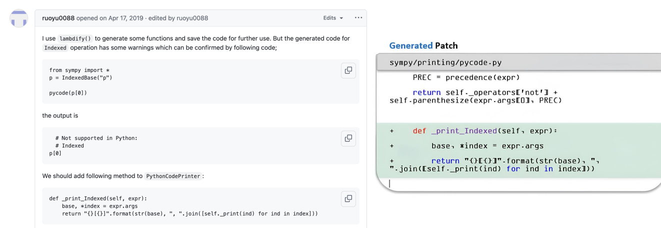


Figure 5: Solution Leakage in issue report for sympy-16669

the task input rather than deriving it independently from repository understanding. In our study, 32.26% of the successfully resolved issues followed this pattern, making it the most common among resolved patches. This raises concerns about whether success in these instances reflects genuine issue-resolution ability and the validity of the *SWE-Bench* instances as benchmarks. If a model simply copies the solution it already has access to, it isn't demonstrating true problem-solving capabilities, but rather replicating what is provided, thus limiting the assessment of its ability to generate new solutions. For example, Figure 6a and Figure 5 show a *sympy* issue whose problem statement provides the exact solution code patch required to resolve the issue, which makes it possible for the model to directly copy the solution from the issue report and generate the same solution as provided.

2. Solution Hint Leak: refers to the instances where the issue description contains partial information or indirect suggestions that guide models toward the solution without explicitly providing the complete fix. This pattern was present in 28.57% of issues. As illustrated in Figure 6b, such hints can influence how a model approaches the solution even when the complete patch is not explicitly provided.

3. Incorrect fixes: refer to cases where the model-generated patches provide incorrect solutions, yet pass the test cases when they should have failed. This pattern was present in 19.82% of the passed instances, suggesting a weakness in test cases that do not correctly capture the functionality of the issue resolution. The fact that incorrect patches can pass the test cases raises suspicion about the relevance and accuracy of the test cases in assessing whether the issue has been fully resolved. Figure 7 shows a comparison between the model-generated patch and the gold patch for sympy-19637. The model-generated patch avoids the original *kern*-before-assignment error by moving *hit* = *kern in s* inside the branch where *kern* is defined. However, it fails to initialize *hit* when the input string contains no spaces. In contrast, the gold patch explicitly sets *hit* = *False* in that case. Therefore, the model patch does not fully fix the issue and may still leave the function in an invalid state for later execution.

4. Mislocalized fixes: This pattern refers to cases where the model-generated patches modify files or functions unrelated to the issue at hand. These files differ from those altered in the gold patch, yet the model's patches still pass the test cases despite this discrepancy. This pattern appears in 10.14% of the passed instances. This pattern highlights a weakness in the model's ability to accurately locate and address the source of the issue. The fact that the test cases pass, even though changes were made in irrelevant files, suggests that the test cases are either weak or irrelevant and should have failed in detecting the incorrect modifications. Figure 8 presents an example from matplotlib-26093 of the Matplotlib project, where the model-generated patch modifies the *cbook.py* file, while the gold patch makes changes to the *_axes.py* file. This shows that the model's patch affects a completely different file from the gold patch, highlighting the model's inability to accurately identify the correct file containing the bug.

5. Incomplete fixes: This pattern refers to model-generated patches that offer incomplete implementations compared to the gold patches, often omitting critical details. This pattern appears in 17.97% of passed instances. For instance, some patches include only partial if-else statements, neglecting edge cases that the gold patch addresses. Although the model-generated patches follow the correct implementation approach, they overlook important aspects that could lead to failures in production or when handling edge cases.

Figure 11: Resolution rates after leakage removal

Table 2: The accuracy of `SoluLeakDetector` on solution-leak detection.

Dataset	# Solution-Leaks	Correctly Detected	Accuracy (%)
SWE-Bench Lite	47	40	85.11
SWE-Bench Verified	113	88	77.88

Figure 8: Different files changed by model for matplotlib-26093

Figure 9: Incomplete fix generated by the model for django-12155

Figure 10: Distribution of Patterns Across Models. Counts per category for AutoCodeRover-v2.0, OpenHands+CodeAct-v2.1, and SWE-Agent-1.0.

This underscores a weakness in the test cases, as they fail to capture the finer details necessary for a comprehensive issue resolution.

The example provided in Figure 9 uses the instance django-12155, which concerns a rendering error in Django's admindocs when a docstring's first line is not empty. The gold patch provides a complete fix by replacing the legacy docstring normalization logic with inspect.cleandoc() in django.contrib.admindocs.utils, and also updating django.contrib.admindocs.views callsite to apply cleandoc(verbose) when formatting method docstrings (removing the dependency on utils.trim_docstring). In contrast, the model-generated patch provides only a partial fix: it modifies the trimming logic in utils.py but fails to update the views.py callsite. As a result, the model patch misses an essential part of the end-to-end fix, ensuring that the view rendering path uses the corrected normalization method, so it can still trigger docutils rendering issues that are exposed by our stronger tests.

3.2 Impact of Solution Leakage and Performance of `SoluLeakDetector`

To quantify how solution leakage affects model performance, we re-evaluated the 132 commonly resolved leakage cases after removing the leaked information from their issue descriptions. Before leakage removal, all 132 instances were successfully resolved by the models. We then automatically removed solution-related information, including direct code solutions, suggestions, and hints, and re-ran the three models to assess whether they could still resolve the issues without leaked guidance.

Figure 11 shows that resolution rates dropped substantially after removing leaked information. OpenHands+CodeAct v2.1 resolved 52/132 instances (39.39%, a 60.61-point drop), AutoCodeRover-v2.0 resolved 87/132 (65.91%), and SWE-agent 1.0 resolved 70/132 (53.03%), demonstrating the positive impact of solution leaks. Examining leakage types, for 70 solution-leak-direct instances, AutoCodeRover-v2.0 led with 70% (49/70), SWE-agent 1.0 65.71% (46/70), and OpenHands+CodeAct v2.1 35.71% (25/70). For 62 solution-leak-hint instances, AutoCodeRover-v2.0 achieved 61.29% (38/62), OpenHands+CodeAct v2.1 43.55% (27/62), and SWE-agent 1.0 38.71% (24/62). Higher resolution rates for direct-solution issues indicate that direct solutions were particularly helpful for the models.

Beyond measuring the impact of leakage on model performance, we further evaluate whether `SoluLeakDetector` can accurately identify solution-leaking issue descriptions in *SWE-Bench Lite & Verified*. To ensure the classification's reliability, two authors independently reviewed the categorized instances by examining all issue reports for *Lite & Verified* cases of solution leakage. This human evaluation process helped mitigate potential biases and verify the correctness of `SoluLeakDetector`'s predictions. The manual review confirmed that `SoluLeakDetector` achieved an overall accuracy of 80.45% in correctly classifying solution leaks across *SWE-Bench Lite & Verified*. For each dataset, `SoluLeakDetector` achieves 85.11% on *Lite* and 77.88% in *Verified* as shown in Table 2. These results show that solution leakage substantially inflates apparent resolution rates and that `SoluLeakDetector` can reliably identify such cases for leakage sanitization in *SWE-Bench+*.

3.3 Impact of Weak Tests and Performance of `TestEnhancer`

We then quantify how weak-test-only cases inflate headline resolution rates. For each model and dataset (i.e., *Lite* and *Verified*), we recompute resolution rates after excluding only weak-test-only issues, while keeping solution-leak cases intact to isolate the impact of weak tests. Following the weak-test breakdown in Table 1,

Table 3: Patches Passing Due to Weak Test Cases (aka. plausible patches): *SWE-Bench Lite* (82/300) vs. *SWE-Bench Verified* (179/500)

Weak tests	AutoCodeRover v2.0	OpenHands+CodeAct v2.1	SWE-Agent v1.0
SWE-Bench Lite (n=82)			
Incorrect fixes	9 (10.98%)	8 (9.76%)	9 (10.98%)
Incomplete fixes	21 (25.61%)	14 (17.07%)	13 (15.85%)
Different files/functions	3 (3.66%)	7 (8.54%)	5 (6.10%)
SWE-Bench Verified (n=179)			
Incorrect fixes	22 (12.29%)	20 (11.17%)	25 (13.97%)
Incomplete fixes	19 (10.61%)	19 (10.61%)	16 (8.94%)
Different files/functions	9 (5.03%)	10 (5.59%)	7 (3.91%)

Table 4: Resolution rates before (Orig.) and after (TE) the enhanced test suite (values in %). Δ is the percentage-point drop.

Model	Dataset	Orig.	TE	Δ (pp)
AutoCodeRover v2.0	Lite	37.33	10.00	27.33
	Verified	45.00	9.20	38.80
OpenHands+CodeAct v2.1	Lite	42.00	15.67	26.33
	Verified	52.40	18.20	34.20
SWE-Agent v1.0	Lite	47.00	19.67	27.33
	Verified	57.60	21.80	35.80

Figure 13: Case study of weak vs. enhanced tests and the corresponding gold patch for `django__django-12155`.

(a) SWE-Bench Lite **(b) SWE-Bench Verified**

Figure 12: Resolution rate changes (Original vs Weak Test Removed) for three models.

Table 3 reports the distribution of weak-test patterns among passed patches (Lite: n=82, Verified: n=179). Using the protocol from Section 3.1, we manually review each passed patch, flag weak-test-only cases, and recount them as failures while keeping the instance set and denominator unchanged, yielding the weak-test-removed resolution rate.

In *SWE-Bench Lite*, AutoCodeRover-v2.0 produced incorrect fixes in 10.98% of cases, incomplete fixes in 25.61%, and fixes involving different files or functions in 3.66%. OpenHands+CodeAct v2.1 showed incorrect fixes in 9.76% of cases, incomplete fixes in 17.07%, and different files/functions in 8.54%. SWE-agent 1.0 generated incorrect fixes in 10.98% of cases, incomplete fixes in 15.85%, and different files/functions in 6.10%. In *SWE-Bench Verified*, AutoCodeRover-v2.0 produced incorrect fixes in 12.29% of cases, incomplete fixes in 10.61%, and fixes involving different files/-functions in 5.03%. OpenHands+CodeAct v2.1 showed incorrect fixes in 11.17% of cases, incomplete fixes in 10.61%, and different files/functions in 5.59%. SWE-agent 1.0 generated incorrect fixes in 13.97% of cases, incomplete fixes in 8.94%, and different files/functions in 3.91%. These suspicious fix patterns resulted in a substantial reduction in the actual resolution rates across all models. Figure 10 also shows the distribution of weak-test-related patterns across models.

Figure 12 contrasts the original resolution rates with weak test issues removed: we treat those instances with weak test patterns as failures, leaving the instance set and denominator unchanged. The difference shows how weak tests inflated the original rates. The new resolution criteria led to significant reductions in resolution percentages across all models. For AutoCodeRover-v2.0, the performance drops by 21.33 percentage points (from 37.33% to 16%) on *SWE-Bench Lite* and drops by 26 percentage points from 45% to 19% on *SWE-Bench Verified*. OpenHands+CodeAct v2.1 showed a decline from 42% to 22% on *Lite* and from 52.4% to 26.8% on *Verified*. SWE-agent 1.0 experienced a drop from 47% to 27.33% on *Lite* and from 57.6% to 31.8% on *Verified* when considering only correct fixes.

We further evaluate whether `TestEnhancer` can strengthen patch validation by automatically generating enhanced tests. We apply the enhanced test suite to patches generated by the three models for issues in *SWE-Bench Lite & Verified* and measure the reduction in resolution rate when applying the newly generated test cases. Our evaluation reveals that `TestEnhancer` enhances patch validation and identifies plausible patches for 97.11% of issues with the weak tests. As shown in Table 4, the enhanced test suite causes substantial drops in resolution rates across all models. The drops are consistent on both *Lite* and *Verified*, and are larger on *Verified*, ranging from 34.2 to 38.8 percentage points, compared with 26.33 to 27.33 percentage points on *Lite*. Notably, `TestEnhancer` also substantially improved coverage, raising average line coverage from 33.39% to 55.55% (a 22.16-point gain). These results show that `TestEnhancer` effectively identifies plausible patches. The enhanced tests expose flaws missed by the original *SWE-Bench* suite, showing that weak tests inflate apparent LLM patch success and that `TestEnhancer` provides a more robust framework for evaluating LLM-generated patches.

Figure 13 illustrates how weak tests can inflate apparent patch success and how `TestEnhancer` tightens the specification. In django-12155, Django's `admindocs` integration can trigger docutils rendering errors when a docstring's first line is not empty, because docstring normalization is not consistently applied across the parsing utilities and the view-level rendering path. The original *SWE-Bench*

test patch is weak because it primarily checks a narrow parsing scenario (e.g., `parse_docstring` and `parse_rst` on a short synthetic docstring) and does not comprehensively enforce that the end-to-end admindocs rendering pipeline uses the corrected normalization (in particular, it does not strongly constrain the `views.py` callsite that formats method docstrings). In contrast, the injected suite adds explicit regression checks that (i) `trim_docstring` is removed, (ii) `utils.parse_docstring` uses `inspect.cleandoc`, and crucially (iii) `django.contrib.admindocs.views` uses `cleandoc(verbose)` and does not reference `utils.trim_docstring`. These stronger tests directly constrain the full fix and expose partial solutions: the gold patch is complete because it updates both `django.contrib.admindocs.utils` and the `views.py` callsite, whereas the model-generated patch is incomplete because it only modifies `utils.py`, leaving the view-level formatting path inconsistent and thus still vulnerable to docutils rendering failures.

4 Related Work

Code Generation Benchmarks LLMs have emerged as powerful tools and demonstrated impressive capabilities in various software engineering tasks, including code generation [5, 11, 13, 18, 20], program repair [7, 34, 41], test generation [25, 28], and bug detection [1, 8]. The development of code generation benchmarks has been crucial for evaluating LLM performance, and recent work has steadily pushed evaluation beyond simple function-level synthesis toward more realistic software settings. Prior benchmarks have expanded the structural scope of generation from standalone functions to interdependent class implementations [11], incorporated contextual dependencies from libraries, files, and projects to better reflect practical development environments [36], and introduced repository-grounded, domain-sensitive, and periodically refreshed tasks to reduce contamination and better capture evolving real-world coding scenarios [17]. Other efforts have broadened evaluation across languages and task types through multilingual, multitask, execution-based settings [33], highlighted non-functional dimensions such as computational efficiency rather than correctness alone [22], and proposed harder compositional tasks in which models must reuse their own generated code to solve linked follow-up problems [38]. These benchmarks show that code generation ability is highly sensitive to task formulation, context, and evaluation dimension. However, greater breadth and realism do not by themselves guarantee trustworthy evaluation. As highlighted by recent reviews, benchmark results can still be distorted by narrow task distributions, dataset contamination, memorization, weak metadata, and metrics that do not reliably reflect true utility or functional quality [21]. Our work builds on this line of research by shifting attention from benchmark breadth to benchmark validity, focusing specifically on *SWE-Bench* and showing that solution leakage and weak tests can substantially inflate reported issue resolution performance, thereby motivating a more rigorous benchmark framework.

Dataset Quality for Software Engineering Tasks Dataset quality significantly impacts the performance, reliability, and reproducibility of empirical software engineering research and automated tools. Prior work has studied dataset quality issues in vulnerability detection [6, 9, 31], defect prediction [3, 26], code summarization [27], and code generation [10, 15, 19, 23, 30]. In vulnerability detection, Croft et al. [6] identify pervasive issues such as incomplete metadata, inconsistent vulnerability labeling, duplicate samples, and unclear dataset construction processes; Wu et al. [31] show that mislabeled instances are common and can degrade model performance and mislead evaluation results; and Ding et al. [9] find that reported gains of code language models are often sensitive to dataset artifacts, data leakage, and benchmark simplicity rather than true semantic understanding. In code-related LLM evaluation, recent work has increasingly highlighted contamination as a critical threat. Khan et al. [15] propose a naturalness-based approach for detecting code contamination, Riddell et al. [23] quantify contamination in code generation benchmarks and showed that even partial exposure can inflate reported performance, and Liang et al. [19] illustrate "SWE-Bench illusion", showing that LLMs often succeed by recalling previously seen solutions rather than reasoning about unseen software engineering tasks. Different from the above studies, this paper presents the first in-depth empirical analysis of *SWE-Bench*. Beyond identifying potential shortcomings, we analyze their impact on evaluation outcomes and explore practical strategies to mitigate them.

5 Threats to Validity

The manual labeling process for identifying solution leaks and weak-test patches introduces potential human bias. Although multiple authors independently reviewed the patches and resolved disagreements through discussion, subjective judgments in classifying leakage types (direct vs. hint) or patch correctness could affect the consistency of our findings. To mitigate this, we employed a structured protocol for patch comparison and provided clear definitions for each problem category. Additionally, the accuracy of `SoluLeakDetector` (80.45%) indicates that some misclassifications remain, which could influence the composition of *SWE-Bench+*.

Our study focuses exclusively on *SWE-Bench*, which, despite its wide adoption, may not capture the full diversity of software engineering tasks, programming languages, or project scales. Future work should therefore examine benchmarks covering additional languages (e.g., Java, JavaScript, C++), broader project types (e.g., systems software, web applications), and tasks beyond bug fixing to assess whether our findings and tools generalize across the broader software engineering landscape.

6 Conclusion

In this paper, we presented *SWE-Bench+*, an enhanced benchmark for evaluating LLM-based software engineering agents. Built on *SWE-Bench Lite* and *SWE-Bench Verified*, it addresses two threats to evaluation validity in *SWE-Bench*: solution leakage in issue descriptions and weak tests. By sanitizing leaked issue content and strengthening test suites through `SoluLeakDetector` and `TestEnhancer`, *SWE-Bench+* provides a more reliable basis for measuring true issue-resolution capability and for improving future software engineering benchmarks.

References

[1] Kamel Alrashedy and Ahmed Binjahlan. 2024. Language Models are Better Bug Detector Through Code-Pair Classification. arXiv:2311.07957 [cs.SE] https://arxiv.org/abs/2311.07957

[2] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2025. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. arXiv:2410.20285 [cs.AI] https://arxiv.org/abs/2410.20285

[3] Kirti Bhandari, Kuldeep Kumar, and Amrit Lal Sangal. 2023. Data quality issues in software fault prediction: a systematic literature review. *Artificial Intelligence Review* 56, 8 (2023), 7839–7908.

[4] Dong Chen, Shaoxin Lin, and Muhan Zeng et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. arXiv:2406.01304 [cs.CL] https://arxiv.org/abs/2406.01304

[5] Mark Chen, Jerry Tworek, and Heewoo Jun et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[6] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.

[7] David de Fitero-Dominguez, Eva Garcia-Lopez, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraiz. 2024. Enhanced automated code vulnerability repair using large language models. *Engineering Applications of Artificial Intelligence* 138 (2024), 109291. https://doi.org/10.1016/j.engappai.2024.109291

[8] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. arXiv:2304.02014 [cs.SE] https://arxiv.org/abs/2304.02014

[9] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).

[10] Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. *arXiv preprint arXiv:2402.15938* (2024).

[11] Xueying Du, Mingwei Liu, and Kaixin et al. Wang. 2024. Evaluating Large Language Models in Class-Level Code Generation *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. https://doi.org/10.1145/3597503.3639219

[12] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. *arXiv preprint arXiv:2507.23370* (2025).

[13] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.CL] https://arxiv.org/abs/2406.00515

[14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[15] Haris Ali Khan, Yanjie Jiang, Qasim Umer, Yuxia Zhang, Waseem Akram, and Hui Liu. 2025. Has My Code Been Stolen for Model Training? A Naturalness Based Approach to Code Contamination Detection. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1046–1067.

[16] Bin Lei, Yuchen Li, Yiming Zeng, Tao Ren, Yi Luo, Tianyu Shi, Zitian Gao, Zeyu Hu, Weitai Kang, and Qiuwu Chen. 2024. Infant Agent: A Tool-Integrated, Logic-Driven Agent with Cost-Effective API Usage. arXiv:2411.01114 [cs.AI] https://arxiv.org/abs/2411.01114

[17] Jia et al. Li. 2024. EvoCodeBench: An Evolving Code Generation Benchmark with Domain-Specific Evaluations. In *Advances in Neural Information Processing Systems*, Vol. 37. Curran Associates, Inc., 57619–57641. https://doi.org/10.52202/079017-1837

[18] Xue Li and Till Döhmen. 2024. Towards Efficient Data Wrangling with LLMs using Code Generation *(DEEM '24)*. Association for Computing Machinery, New York, NY, USA, 62–66. https://doi.org/10.1145/3650203.3663334

[19] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. *arXiv preprint arXiv:2506.12286* (2025).

[20] Hanbin Luo, Jianxin Wu, Jiajing Liu, and Maxwell Fordjour Antwi-Afari. 2024. Large language model-based code generation for the control of construction assembly robots: A hierarchical generation approach. *Developments in the Built Environment* 19 (2024), 100488. https://doi.org/10.1016/j.dibe.2024.100488

[21] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. 87–94. https://doi.org/10.1109/AITest62860.2024.00019

[22] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. COFFE: A Code Efficiency Benchmark for Code Generation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE012 (June 2025), 24 pages. https://doi.org/10.1145/3715727

[23] Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811* (2024).

[24] Ricardo La Rosa, Corey Hulse, and Bangdi Liu. 2024. Can Github issues be solved with Tree Of Thoughts? arXiv:2405.13057 [cs.SE] https://arxiv.org/abs/2405.13057

[25] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.

[26] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on software engineering* 39, 9 (2013), 1208–1215.

[27] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119.

[28] Jiho Shin, Nima Shiri Harzevili, Reem Aleithan, Hadi Hemmati, and Song Wang. 2024. Retrieval-augmented test generation: How far are we? *arXiv preprint arXiv:2409.12682* (2024).

[29] 5 Team and Aohan Zeng et al. 2025. GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models. arXiv:2508.06471 [cs.CL] https://arxiv.org/abs/2508.06471

[30] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314* 4 (2024).

[31] Xiaoxue Wu, Wei Zheng, Xin Xia, and David Lo. 2021. Data quality matters: A case study on data label correctness for security bug report prediction. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2541–2556.

[32] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489 [cs.SE] https://arxiv.org/abs/2407.01489

[33] Weixiang et al. Yan. 2024. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 5511–5558.

[34] Boyang Yang, Haoye Tian, Weiguo Pian, Haoran Yu, Haitao Wang, Jacques Klein, Tegawendé F. Bissyandé, and Shunfu Jin. 2024. CREF: An LLM-Based Conversational Software Repair Framework for Programming Tutors *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 882–894. https://doi.org/10.1145/3650212.3680328

[35] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793 [cs.SE] https://arxiv.org/abs/2405.15793

[36] Hao et al. Yu. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/3597503.3623316

[37] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An llm agent framework for software issue localization. *arXiv preprint arXiv:2502.00350* (2025).

[38] Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199* (2024).

[39] Daoguang Zan, Zhirong Huang, and Ailun Yu et al. 2024. SWE-bench-java: A GitHub Issue Resolving Benchmark for Java. arXiv:2408.14354 [cs.SE] https://arxiv.org/abs/2408.14354

[40] Kexun Zhang, Weiran Yao, and Zuxin Liu et al. 2024. Diversity Empowers Intelligence: Integrating Expertise of Software Engineering Agents. arXiv:2408.07060 [cs.SE] https://arxiv.org/abs/2408.07060

[41] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2024. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. arXiv:2310.08879 [cs.SE] https://arxiv.org/abs/2310.08879

[42] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE] https://arxiv.org/abs/2404.05427