ExplainBench: Evaluating Code Explanations from Agents

Zhiyuan Pan*†
Zhejiang University
Hangzhou, China
zy_pan@zju.edu.cn

Sungmin Kang†
National University of Singapore
Singapore
sungmin@nus.edu.sg

Imam Nur Bani Yusuf†
National University of Singapore
Singapore
inbyusuf@nus.edu.sg

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@nus.edu.sg

Abstract

Large Language Model (LLM) agents have seen rapid adoption in software engineering. As agents take a greater role in the actual generation of code, they are making larger changes, spanning tens to hundreds of lines. This makes manual review of agent results increasingly infeasible, leading developers to turn to explanations to understand enacted changes. Despite this, there are no benchmarks that evaluate the trustworthiness of agent-generated explanations. To bridge this gap, we propose ExplainBench, a benchmark to automatically evaluate explanations from coding agents. ExplainBench is based on the intuition that informative explanations should enable an LLM to correctly answer questions, allowing quantitative comparison of explanation quality between agents. With this observation, we construct a suite of questions that evaluates whether explanations accurately describe (1) the intended behavior of buggy code and (2) the effect of applying the agent patch itself. Experiments first reveal that explanation quality is a distinct axis of agent evaluation: ExplainBench ranks agents differently from the widely-used SWE-bench Verified benchmark. A deeper breakdown of explanation quality in agents shows frequent problems in explanations, such that explanations often claim that a patch is correct when it is not. Based on this insight, we implement and evaluate an explanation audit agent which runs additional tests to validate and refine explanations. This agent improved the explanations of all evaluated agents, demonstrating agent explanations can be automatically made more trustworthy.

CCS Concepts

• **Software and its engineering** → *Software usability*;

Keywords

LLM agents, explainability, benchmark, usability, trustworthiness

*Work done while the author was in the National University of Singapore.
†Three authors contributed equally to this research.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Zhiyuan Pan, Sungmin Kang, Imam Nur Bani Yusuf, and Abhik Roychoudhury. 2018. ExplainBench: Evaluating Code Explanations from Agents. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large Language Model (LLM) agents are LLMs equipped with a scaffolding that allows them to invoke tools to retrieve information and enact changes. Agents provide state-of-the-art performance in software engineering tasks such as test generation [2] or program repair [41], and are reshaping the practice of software engineering. This has led to their quick adoption by industry and a change in how developers work. For example, Anthropic reported that a majority of employees were using Claude Code on a daily basis [11].

As LLM agents become more flexible and deal with complex tasks, it becomes increasingly important that agents provide *trustworthy explanations*. This is because the speed and scale of agent code generation make manual inspection of agent results increasingly costly, while explanations provide a useful summary of the implementation, provided the explanation is reliable. Thus, as the capability of LLM agents grows, developers would increasingly treat agents like junior developers summarizing their work – making it important for agents to provide accurate and honest explanations, in addition to being capable in software engineering tasks.

However, the importance of explanations for agents has not yet been recognized, and there is a lack of benchmarks that evaluate explanation trustworthiness. This is in contrast to the large number of benchmarks evaluating agent efficacy [12, 13, 24]. This absence poses two problems. First, it is difficult to tell which agent explanations are the most reliable, as opposed to which agents are the most effective. Secondly, the lack of a benchmark makes it difficult for researchers to iterate on their design and find techniques that can improve the correctness of patch explanations.

This work introduces ExplainBench, a benchmark that automatically and quantitatively assesses the quality of patch explanations generated by agents. The key challenge in evaluating explanation quality is that agent explanations are in natural language, making automated and objective assessment of the content difficult. To overcome this challenge, ExplainBench evaluates the natural language explanation in the form of an *LLM questionnaire*. In particular, ExplainBench provides an explanation as input to an LLM, and has the LLM answer multiple-choice questions about the bug and

patch. The intuition is that informative explanations will be sufficient to correctly answer questions, while vacuous explanations with no meaningful content will cause an LLM to answer questions incorrectly. Quantitatively, we use the proportion of correctly answered questions as the explanation score. In turn, this score allows comparison between agent explanations, while facilitating experimentation to improve agent explanations.

Evaluating five agents on ExplainBench first revealed that agent explanation quality is independent from agent efficacy. For example, the trae-agent [34] in our evaluation had the highest SWE-bench Verified resolution rate, but had the second-lowest explanation score on ExplainBench. Further analysis reveals that existing agent explanations are over-optimistic about the correctness of the patch, and suffer from inaccurate inference of function-level intended behavior. To address these issues, we propose ExplanationAuditAgent, an LLM agent that runs differential testing on the code before and after the patch and audits claims in agent explanations. The audit agent could generate higher-quality refined explanations for all agents we experiment with, demonstrating its capability as a general explanation improver for any software agent.

Our contributions are as follows:

- We propose ExplainBench, which automatically evaluates agent explanation quality.
- We find patch efficacy does not always correspond with explanation quality, showing the need for evaluation of explanations.
- We implement ExplanationAuditAgent, which performs differential testing based on the content of the explanation and provides a refined and more trustworthy explanation.

While we evaluate certain well-defined aspects of explanations, we do not argue that these are the only quality aspects that can be evaluated. In particular, we clarify that ExplainBench evaluates explanation correctness in terms of explanation content, which is a key issue in software documentation [1]; it does not measure developer satisfaction or readability. We publish our code [29], written in a modular fashion, such that the benchmark can be extended to support the evaluation of explanations on criteria other than those used in this paper.

2 Related Work

2.1 Agents for Software Engineering

LLMs have been rapidly adopted in many industries, but their application in software engineering has been notably widespread. Among the first agents for software engineering were the issue-resolving agents SWE-agent [41] and AutoCodeRover [47], which provided a scaffolding around LLMs to autonomously retrieve information and generate patches. Subsequently, other domain-specific agents have been proposed, such as the Otter [2] agent for generating bug-reproducing tests or the ExecutionAgent [4] agent for building execution environments. In the opposite direction, more general agents which can handle many types of tasks such as OpenHands [38] or USEAgent [3] have also been explored. Despite such domain and architecture diversity, and the frequency and ease with which they provide natural language explanations, the trustworthiness of their explanations is poorly understood.

2.2 Benchmarks for Agentic SE

Many benchmarks evaluating agents on software engineering tasks have been proposed. A prominent benchmark is SWE-bench Verified [8], which consists of 500 human-verified instances from the earlier SWE-bench [12] benchmark of reproducible real-world issues. Influenced by SWE-bench Verified, many other benchmarks that evaluate agent code generation capability have been proposed. SWE-Poly Bench [30] and Multi-SWE-bench [45] are multilingual variants on SWE-bench. SWE-bench-Live [46] and SWE-Bench-CL [14] order bugs by time: SWE-bench-Live updates regularly to avoid LLM contamination, while SWE-Bench-CL evaluates the ability of agents to utilize prior issues. Benchmarks in specialized domains have also been proposed. DSBench evaluates the ability of agents to perform data science tasks [13], while Terminal-Bench evaluates the ability of agents to perform complicated tasks in the terminal [35]; GitTaskBench [25] proposes a collection of realistic tasks across different modalities and domains.

To the best of our knowledge, ExplainBench is the first benchmark to automatically evaluate agent explanation quality. In this work, we compare explanation quality with issue resolution rate on SWE-bench Verified to show that explanation quality is an independent axis of evaluation.

2.3 Explanations in Software Engineering

Surveys on developers have revealed a consistent demand for explanations when using automated software engineering techniques. Noller et al. [26], who surveyed developer perception of what is needed to trust program repair tools, found that explanations were the second most sought after artifact (after patches themselves) for program repair tools. Developers also indicate that explanations would be used extensively in their reasoning process: one developer survey by Kochhar et al. [16] noted that they would judge the result in the context of the explanation.

While earlier surveys on automated debugging techniques noted an absence of explainable techniques, the introduction of LLMs has significantly changed the situation. LLMs, which are capable of generating both source code and natural language, have been frequently deployed to make claims to explainability. As a result, there are several works [15, 23, 39] that uses LLMs to generate explanations along with patches and fault localization results. However, due to an absence of a framework to evaluate explanations, each research paper had to perform an ad-hoc manual evaluation of explanations. While this is qualitatively revealing, the lack of quantitative evaluation makes it difficult to compare explanation generation techniques. Moreover, as a natural byproduct of their chain-of-thought operation, many agents also generate natural language explanations, as described in Section 5.1, which have not yet been evaluated to the best of our knowledge.

3 Motivation

To illustrate why agent explanations need to be evaluated, we discuss the motivating example in Fig. 1, based on output from the Lingxi agent [42]. When interacting with agentic systems, developers often see a natural language summary before inspecting the generated patch. For instance, the popular Claude Code[1] tool presents

[1] https://claude.com/product/claude-code

```
### Issue Requirements Check
**Original Issue**: Setting `min-similarity-lines` to
0 should disable the duplicate code check
...
### Final Verification
The fix successfully addresses the GitHub issue by:
1. **Disabling the duplicate code check** when `min-similarity-lines=0`
...
The fix resolves the issue completely (...)
```

(a) Natural language explanation.

```
diff --git a/pylint/checkers/similar.py b/pylint/checkers/similar.py
--- a/pylint/checkers/similar.py
+++ b/pylint/checkers/similar.py
@@ -830,6 +830,8 @@ class SimilarChecker(BaseChecker, Similar, MapReduceMixin):
...
+ if self.min_lines <= 0:
+   return
...
```

(b) Submitted patch (never covered).

Figure 1: Example explanation and submitted patch from the Lingxi agent [42].

a natural language summary upon completion. In our example presented at Fig. 1a as well, a developer would read the summary and naturally assume that a patch was successfully implemented and resolved the issue. Even if they saw the patch (Fig. 1b), it would appear reasonable. However, a developer in this example would be surprised: the patch-modified method is actually never covered in bug-reproducing tests, indicating that the patch has no impact on the bug. Whenever developers experience such an explanation-behavior mismatch, their *trust* in agentic systems erodes.

To assess the extent of such problems, this work builds a benchmark to evaluate to what degree agents would generate inaccurate explanations. The main challenge in automatically evaluating explanations is that the explanations provide information in natural language, and thus the same information may take many different forms, making it difficult to automatically evaluate explanations. To overcome this challenge, we formulate a *questionnaire* with verifiable answers, and check if the natural language explanation provides sufficient information for an LLM to correctly answer questions. See Fig. 2(B) for a schematic of our core intuition: informative explanations would allow an LLM to correctly answer questions about the bug and patch, while uninformative or misleading explanations would lead to incorrect answers. In turn, based on the accuracy of the questionnaire, one can deduce how informative an explanation is. Through careful choice of the question types, one can evaluate whether specific types of information are present in the explanation as well, allowing a fine-grained assessment of explanation quality.

In summary, our *problem statement* is that we are interested in automatically evaluating the quality of explanations generated by coding agents, making quality comparison and improvement feasible. Our *core assumption* to solve this problem is that informative explanations would help LLMs accurately answer questions about the bug. LLMs are an imperfect yet useful evaluator of natural language artifacts, as evidenced by the large number of benchmarks using them [19, 20, 48]; LLMs are particularly adept at identifying relevant information from text [31], relevant to our setup. In this work, we have the LLM answer questions with objective answers as an additional measure of rigor.

4 ExplainBench Framework

This section describes the general structure of ExplainBench and what questions it generates; the detailed question construction process is described in Section 5.

4.1 Overview

Fig. 2 presents an overview of our evaluation framework, ExplainBench. Our goal is to assess whether a patch explanation conveys information relevant to (1) the intended change in program behavior and (2) the actual effect of the agent patch. To this end, ExplainBench first gathers behavioral information from developer tests, developer patches and agent patches. ExplainBench then constructs questions about the bug and patch (Fig. 2(A)). Finally, explanations are evaluated by providing them to an LLM, and having the LLM answer the questions (Fig. 2(B)).

The remainder of this section describes the principles behind the benchmark construction and the question types used in our study. The specific process of artifact preparation and question construction on top of the SWE-bench Verified [12] benchmark is described in Section 5.

4.2 Principles

ExplainBench evaluates explanations by running questions against the explanations using an LLM. In constructing these questions for ExplainBench, we followed the following principles:

(1) **When possible, focus on questions answerable with general expressions, not specific values.** Questions with values as answers, as opposed to symbolic *expressions*, require explanations to contain unnecessary details about how to calculate the expected value. In contrast, explanations should concisely describe key characteristics of the bug, which can be translated by an LLM into expressions.
(2) **LLMs should not directly synthesize expressions when answering questions.** Generating valid code expressions requires a significant amount of context. It is thus unreasonable to expect that an explanation should contain all of this only to reconstruct expressions.

These principles motivate our use of *multiple-choice questions (MCQs)* in ExplainBench. ExplainBench automatically generates validated expressions that capture the essence of both the developer patch and agent patch. This avoids the direct use of complex values and the synthesis of expressions during question-answering.

4.3 Evaluation Desiderata

This work evaluates whether explanations contain the *intent* and *effect* of the agent patch. Intent refers to what the agent patch should do, while effect refers to what the agent patch actually does. While intent and effect would ideally be identical, they can be different in practice; good explanations would distinguish the two. In our setting, questions about intent are derived from the developer patch, which we consider an oracle for the developer intent, as is done in prior work [37]. Meanwhile, questions about effect are derived from the agent patch, representing the actual change to the code.

We focus on intent and effect in this work based on prior research on how developers document changes. First, intent is often missing

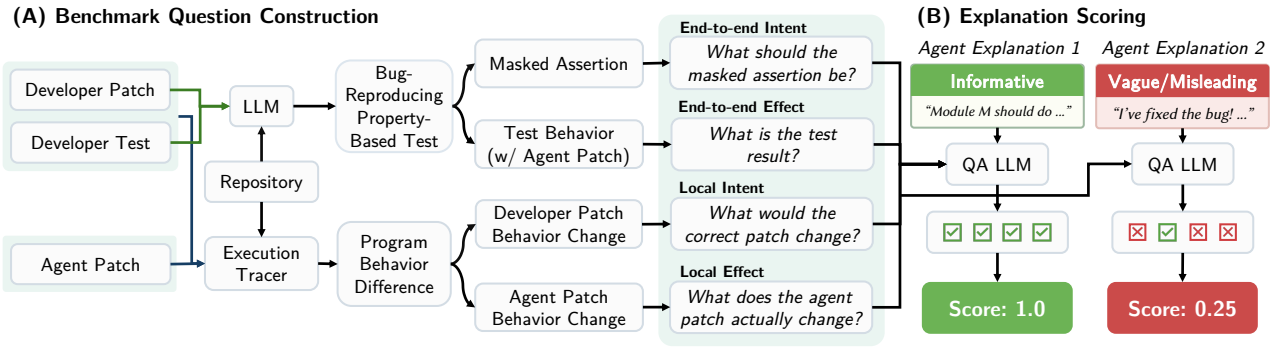

Figure 2: Overview of ExplainBench.

Masked Test:
```
@given(...)
def test_396c8f15(array):
    table = Table([array],
    names=['col1'])
    col = table['col1']
    assert [[MASKED]]
```
Choices:
A. hasattr(col, 'mask')
B. isinstance(col, Column)
C. isinstance(col, NdarrayMixin)
D. not isinstance(col, Column)
E. Explanation insufficient to answer.
Q: Based on the explanation, what should go in [[MASKED]]?

(a) End-to-end intent question.

Test:
```
@given(...)
def test_396c8f15(array):
    table = Table([array],
    names=['col1'])
    col = table['col1']
    assert isinstance(col, Column)
```
Choices:
A. Test passed.
B. Test failed (line 4, TypeError)
C. Test failed (line 5, KeyError)
D. Test failed (line 6, AssertionError)
E. Explanation insufficient to answer.
Q: What happens when running the test before and after applying the patch?

(b) End-to-end effect question.

Function:
```
def _convert_data_to_col(self, ...):
    ...
```
Choices:
A. getattr(col, 'name', None)
B. tuple(col.shape)
C. list(col.dtype.names)
D. hasattr(col, 'name')
E. Explanation insufficient to answer.
Q: Which expression best describes the developer-intended change?

(c) Local intent question.

Function:
```
def _convert_data_to_col(self, ...):
    ...
```
Choices:
A. len(__return__.dtype.names)
B. len(data.dtype.names)
C. tuple(data.dtype.names)
D. The patch has no effect.
E. Explanation insufficient to answer.
Q: Which expression has different values before and after the patch?

(d) Local effect question.

Figure 3: Example questions. Edited for brevity.

yet desired from LLM-generated explanations: Xiao et al. [40] find that one of the common manual edits to LLM-generated pull request descriptions was to add the intent of the change. Prior work on commit messages and pull requests [21, 36] also notes the need to describe the expected behavior of the code. Meanwhile, the explanation needs to accurately describe the actual effect of a patch to help developers understand whether to accept patches [22].

Meanwhile, we further divide evaluation of intent and effect by granularity, namely based on an end-to-end scope and a local scope. Ideally, an explanation would describe how the program works as a whole. Furthermore, developers often need to know which program component is behaving incorrectly and what its intended behavior should be [16]. While there are different granularities one could use for local questions, we opt to focus on the function level. This is because existing studies suggest that functions are cohesive logical units of software [5], making them the most useful unit to explain.

4.4 Question Types

Based on evaluation desiderata, ExplainBench contains four questions from the intersection of intent/effect and end-to-end/local, as described next. Examples for each question are given in Fig. 3.

4.4.1 End-to-End Intent. This question type evaluates whether an explanation conveys what the program is intended to do in contrast to the buggy behavior. This may be understood as the program-level specification that the buggy implementation violates.

Question Format. An LLM is given an agent explanation as well as a *masked* bug-reproducing property-based test (PBT). Property-based tests are tests that sample inputs satisfying a precondition, then check that program output follows a property. This makes PBTs well-suited to our design principles [9]. In the question, the PBT is masked so that the expression describing the expected behavior must be guessed. The LLM is then asked a multiple-choice question to determine which candidate expression would make the PBT fail on the buggy version and pass on the fixed version. This question thus evaluates whether the explanation provides sufficient information for an LLM to identify the expression that correctly describes the intended behavior change of the program as a whole.

4.4.2 End-to-End Effect. This question type evaluates whether an explanation sufficiently describes how the agent-submitted patch changes the behavior of the program. The reliability of this information has important implications for real-world use: if an agent reports that it has 'fixed a bug', how much should this be trusted?

Question Format. An LLM is given an agent explanation as well as a bug-reproducing PBT. The LLM is then asked a multiple-choice question on the result of running the test (i) before the patch and (ii) after the patch is applied. The options given are: the test passed, a few options stating the test failed at line `N` with exception `Exception`, or that there is insufficient information in the explanation to answer the question. This question thus evaluates whether an explanation is enough to derive how the program behavior would change, as revealed by a test.

4.4.3 Local Intent. This question type evaluates whether an explanation describes the expected behavior of the faulty component at the function level. This may be understood as evaluating whether the explanation correctly pinpoints the faulty component and describes what function-level specification was violated.

Question Format. An LLM is given an agent explanation as well as (1) a function influenced by the bug (selection process described

in Section 5), (2) a source line where the bug manifests, and (3) the inputs to the function. For each local question, the provided context also specifies whether expressions are evaluated immediately before or after executing the specified line; Fig. 3 omits the line and timing context for space. The LLM is then asked the question *"Within the context of the provided function and inputs, at the specified timing point relative to the specified line, which expression best describes the developer-intended change?"*

4.4.4 Local Effect. This question type evaluates whether the explanation describes the actual effect of the agent patch at the function level. This information is important to let the end-users know how the patch influences the program state at the function level.

Question Format. An LLM is given an agent explanation as well as (1) the pre-patch version of a function influenced by the bug (selection process described in Section 5), (2) a source line where the bug manifests, and (3) the inputs to the function. We do not provide the agent-generated patch, which prevents the question-answering LLM from inferring local changes directly from the patch rather than from the explanation. The LLM is then asked: *"Within the context of the provided function and inputs, at the specified timing point relative to the specified line, which expression has different values before and after the patch?"*

4.5 Evaluation Metrics

During explanation evaluation, each patch explanation is paired with each of the questions, and an LLM is prompted to answer the question based on the explanation. As the questions are in MCQ format, the LLM generates a single-letter answer (or a pair of letters for end-to-end effect questions). The question is considered correctly answered when the answer from the LLM is identical to the letter(s) corresponding to the ground-truth.

To quantitatively evaluate the explanation quality of a coding agent, we define the *explanation score* based on whether the explanation is sufficient to correctly answer the questions. Simply put, the explanation score is the proportion of questions from ExplainBench that were correctly answered. Formally, let $d \in \mathcal{D}$ denote an evaluation dimension (question type) and let $a_{b,d} \in \{0, 1\}$ indicate whether the explanation for bug instance $b \in \mathcal{B}$ leads to an LLM correctly answering the question associated with dimension d. The explanation score is computed using Equation 1.

$$S_{\text{exp}} = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} \left(\frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} a_{b,d} \right) \tag{1}$$

While Equation 1 averages over all question types, one can get scores for individual question types as well, facilitating deeper analysis of the explanations.

5 ExplainBench Construction

We next describe how the four types of questions introduced in the previous section are implemented in ExplainBench. We build our benchmark on top of SWE-bench Verified [12], a benchmark of human-validated open-source issue descriptions and patches. Given a bug instance from SWE-bench Verified and an extracted agent explanation, our construction pipeline derives multiple-choice questions with correct answers that are grounded either in developer patches (for intent questions) or agent patches (for effect questions).

5.1 Agent Explanation Extraction

First, we establish that modern LLM agents often include a natural language explanation along with their final answer in the last tool call step. Specifically, we find that 9 of the top 10 agents on the SWE-bench Verified leaderboard[2] generate natural language explanations in the last step, based on public submissions[3].

To evaluate these explanations with ExplainBench, we first extract the natural language explanations produced by coding agents during bug-fixing. This is done by parsing agent trajectories to locate the final tool-call step, then collecting the associated natural-language rationale without modification. This ensures that our evaluation is performed directly on the explanations generated by the agents themselves.

5.2 End-to-End Question Construction

5.2.1 Property-based Test Construction. As previously described, end-to-end questions use bug-reproducing property-based tests (PBTs) as a symbolic representation of the input-output behavior of the program. As the SWE-bench Verified benchmark does not provide bug-reproducing PBTs, we needed to generate such PBTs. To do so, we used the test generation agent of SpecRover [33], which takes an issue statement and writes a bug-reproducing test, but with two changes. First, the agent additionally took the developer patch and (example-based) bug-reproducing test as input to aid PBT generation. Second, the agent was prompted to provide a PBT as output, instead of an example-based test. All resulting tests were validated such that they fail on the buggy version and pass on the developer-fixed version. While this process could automatically generate bug-reproducing PBTs for 98% of the bugs in ExplainBench, in 2% of cases, PBT generation failed despite repeated attempts. In such cases, we manually wrote PBTs as necessary.

5.2.2 End-to-End Intent. Explanations should contain sufficient information to reconstruct the expression representing the expected behavior among competing candidates. Thus, we first manually identified the expected behavior expression from the generated PBTs for each bug. In the question, this expression is masked, so that the LLM must choose from a set of expressions to complete a bug-reproducing test. This leaves us with a masked PBT and correct expected behavior expression. However, as our questions are in MCQ format, incorrect answers must also be generated. This is done in two steps. First, an LLM is prompted to generate 'distractors', i.e., incorrect expected behavior expressions similar to the correct expression. Next, these 'distractor' expressions are validated to confirm that they are genuinely not bug-reproducing. Specifically, the distractor expressions are replaced with the correct expression in the PBT, and the PBT is executed. Distractor expressions that make the PBT pass on the buggy version or fail on the fixed version are kept as validated distractors. With a set of validated distractors, the question is composed with the following components: (1) a masked PBT with the correct expression replaced by `[[MASKED]]`, (2) choices, including the correct answer, the distractors, and one 'explanation insufficient to answer' option for analysis, and (3) the

[2]URL: https://www.swebench.com, February 2026.
[3]https://github.com/SWE-bench/experiments

question, asking the LLM what expression should go in `[[MASKED]]`. An example of the end-to-end intent question is shown in Fig. 3a.

5.2.3 End-to-End Effect. While the intent of the code stays constant, as it is defined by the developer patch, the effect of each agent patch differs by agent. As such, end-to-end effect questions need to be generated for each agent submission.

To construct the questions, we first record whether the PBT passes or fails before and after applying the patch. When the test fails, information on which line causes the failure and what exception was triggered is recorded. Note that the behavior may be equivalent before and after the patch, if the agent patch is disconnected with the bug. In addition to behavior from the buggy and patched code, $N_{\text{distractor}}$ distractor options are generated by prompting an LLM to generate alternative lines and exceptions at which the test could fail. With a set of distractors, the question is composed with the following components: (1) the full PBT, (2) choices, including the true pre-patch and post-patch behavior, distractors, and one 'explanation insufficient to answer' option for analysis, and (3) the question asking the test execution result before and after the patch. An example question is shown in Fig. 3b.

5.3 Local Question Construction

For local questions, distinguishing the execution behavior of the buggy and patched programs under the same test input is crucial. Behavioral differences can be represented via expressions that change in value before and after a patch (or the absence of such expressions for ineffective patches). We formalize this as finding *delta behavior* from the execution of the buggy and patched programs. Specifically, given a test input t, let $T_B = \langle (s_0, \sigma_B(0)), (s_1, \sigma_B(1)), \ldots \rangle$ and $T_P = \langle (s_0, \sigma_P(0)), (s_1, \sigma_P(1)), \ldots \rangle$ denote the execution traces of a buggy program B and a patched program P, where s_ℓ is the ℓ-th executed statement and $\sigma_X(\ell)$ is the variable state after executing s_ℓ in program $X \in \{B, P\}$. The delta behavior is defined as the smallest index ℓ where the executions differ in either variable state ($\sigma_B(\ell) \neq \sigma_P(\ell)$) or control-flow ($s_\ell(B) \neq s_\ell(P)$). We choose the earliest index ℓ because this location represents the beginning of all subsequent behavioral differences.

Given the identified delta behavior, we generate expressions characterizing the behavioral difference as question choices, based on our principles in Section 4.2. The ground-truth value of each expression is evaluated at the specified timing point, immediately before or after executing the line associated with the delta behavior. Examples of local intent and effect questions are shown in Figs. 3c and 3d. The choices in the questions consist of: (1) the correct answer, (2) the distractors, and (3) one "*explanation insufficient to answer*" option for analysis. For local effect questions, we additionally include a "*patch has no effect*" option. The key difference between intent and effect questions is that intent questions are constructed based on the delta behavior from developer patches, whereas effect questions are constructed based on the delta behavior from agent patches.

5.3.1 Execution Trace Collection. To compare the execution behavior of buggy and patched programs, it is essential to record both control and data flow. Therefore, we collect execution traces by implementing a Python execution tracer based on the `sys.settrace()` function, as is common in prior benchmarking work [7]. Specifically, we instrument the official SWE-bench test execution harness[4]. As the instrumented pipeline executes the test suites on both program versions, the tracer records execution events (*control flow*) and variable values (*data flow*) at line-level granularity. To difference variables in Python, we implement a serializer embedded in the tracer that converts objects into JSON representations. To reduce the overhead of instrumentation, we prescreen each test case to identify all callers of patch-modified functions, and restrict heavyweight tracing to these filtered functions. The recorded events are subsequently used to construct a hierarchical representation of function frames, forming the basis for comparing buggy and patched program executions and for identifying the behavioral deltas between their corresponding traces in the next step.

Algorithm 1 Delta Behavior Extraction

Input: Buggy trace T_B, Patched trace T_P
Output: Delta behavior Δ or $\emptyset$
Notation: f: *function*

$f_B \leftarrow$ entry frame of T_B; $f_P \leftarrow$ entry frame of T_P
while $f_B \neq \emptyset$ **and** $f_P \neq \emptyset$ **do**
  if $hasNext(f_B)$ **and** $hasNext(f_P)$ **then**
    $s_B \leftarrow next(f_B)$; $s_P \leftarrow next(f_P)$
  else
    $f_B \leftarrow f_B.caller$; $f_P \leftarrow f_P.caller$
    continue
  end if
  if $s_B = s_P$ **then** ▷ control-flow match case
    if s_B is a function call **then**
      $f_B \leftarrow f_B.step\_into(s_B)$; $f_P \leftarrow f_P.step\_into(s_P)$
    else if $\sigma_B \neq \sigma_P$ **then** ▷ variable state differs
      return observed variable difference
    end if
  else ▷ control-flow mismatch case, $s_B \neq s_P$
    if outcomes of f_B and f_P differ **then**
      return observed outcome difference
    end if
    $f_B \leftarrow f_B.caller$; $f_P \leftarrow f_P.caller$
  end if
end while
return $\emptyset$

5.3.2 Delta Behavior Extraction. With execution traces collected in the previous step, we extract *delta behavior*. As defined above, delta behavior is the first semantically observable difference between pre- and post-patch executions, manifesting either as a difference in variable state at the ℓ-th executed statement ($\sigma_B(\ell) \neq \sigma_P(\ell)$) or as a discrepancy in control-flow ($s_\ell(B) \neq s_\ell(P)$).

Algorithm 1 summarizes our procedure for extracting delta behaviors from a pair of buggy and patched execution traces. The algorithm takes as input the buggy and patched program execution traces, T_b and T_p respectively, and returns either the first semantically observable behavioral difference or $\emptyset$ if no difference is observed.

The algorithm traverses both the buggy and patched execution traces from their entry frames while preserving calling context. At each step, it advances by retrieving aligned execution events when available, steps into callee frames for function-call events, and compares observable program states for non-call events, reporting the first detected difference as the delta behavior. Misaligned frames or events are treated as control-flow divergences, in which

[4] https://github.com/SWE-bench/SWE-bench

case the algorithm compares function outcomes (return values or exceptions) and either reports a discrepancy or backtracks to caller frames to continue the traversal. If the traversal completes without detecting any difference, the algorithm returns $\emptyset$, indicating equivalent observable behavior under the given test input.

5.3.3 Candidate Expression Generation and Validation. After the delta behavior is extracted, the next step is to generate candidate expressions that capture the behavioral change the patch causes, along with unchanged expressions (distractors).Given the delta behavior, we prompt a separate LLM to generate candidate changed and unchanged expressions based on the following information: (1) the function in buggy and patched versions in which the divergence occurs, (2) the diverging statement identified by Algorithm 1, and (3) the complete variable states (with value-differing variables indicated) observed at the divergence point in both buggy and patched executions. We generate $N_{\text{candidate}}$ expressions for both changed and unchanged expressions (our experiments use $N_{\text{candidate}} = 10$).

Given $N_{\text{candidate}}$ generated expressions, we validate each to ensure it exhibits the expected behavioral property. Specifically, a *changed* expression must evaluate to different values in the buggy and patched programs, whereas an *unchanged* expression must evaluate identically. We implement a debugger to evaluate each candidate in both programs and retain only those consistent with their assigned category, resulting in N_{changed} and $N_{\text{unchanged}}$ validated candidates for the changed and unchanged categories, respectively.

5.3.4 Candidate Expression Selection. After obtaining validated candidate expressions, we finalize the multiple-choice options. We adopt the following strategy to avoid trivial/obvious choices. Let $C = \{c_1, \ldots, c_{N_{\text{changed}}}\}$ denote the set of changed expressions and $U = \{u_1, \ldots, u_{N_{\text{unchanged}}}\}$ the set of unchanged expressions.

We choose the correct option among the candidates $c_i \in C$ according to its average similarity to the unchanged candidates $u_i \in U$, as defined in Equation 2.

$$\text{sim}_{\text{avg}}(c, U) = \frac{1}{|U|} \sum_{u \in U} \text{sim}(c, u) \tag{2}$$

We use the normalized Levenshtein distance [44] for the similarity $\text{sim}(c, u)$. This favors candidates $c_{\text{selected}} \in C$ with high similarity to unchanged expressions, which are thus easily confusable with unchanged ones, reducing the likelihood of trivial questions.

To select distractors, we leverage a maximal marginal relevance (MMR) criterion [6]. The goal is to discourage selecting multiple distractors that are near-duplicates of each other while still encouraging distractors similar to c_{selected}. Each unchanged candidate $u_i \in U$ is scored as below:

$$\text{MMR}(u) = \lambda \cdot \text{sim}(u, c_{\text{selected}}) \; - \; (1 - \lambda) \cdot \max_{d \in D} \text{sim}(u, d) \tag{3}$$

Here, D denotes the set of already selected distractors and $\lambda \in [0, 1]$ controls the relevance-diversity trade-off (λ defaults to 0.7 in our experiments). The similarity for both $\text{sim}(u, c_{\text{selected}})$ and $\text{sim}(u, d)$ is computed using Levenshtein distance. The top $N_{\text{distractor}}$ expressions are selected iteratively.

5.4 Question Quality Assurance

To ensure the quality of our generated questions, we perform the following quality checks. For the end-to-end questions, we ensure that PBTs fail on the buggy version and pass upon applying the developer patch. For the local questions involving expressions, each delta behavior was manually checked to verify that it reflects a genuine behavioral change from the patch. Two authors jointly reviewed and discussed all uncertain instances to reach consensus. For all questions, we check that instructions are clear by running ExplainBench on blank explanations and ensuring that this yields a score of 0 (as the LLMs pick the incorrect option "*explanation insufficient to answer*"). As an additional quality check, we performed a manual examination through an assessment procedure based on prior work [43]. Specifically, two authors independently answered 40 randomly sampled questions covering all question types from ExplainBench. Any disagreements were resolved by a third author. Agreement between human and LLM answers reached a Cohen's Kappa [10] of 0.7, which is considered as substantial agreement [17, 18, 32]. We show an example of human labeling and disagreement resolution in Figure 4.

```
Explanation:
All the relevant tests pass. The issue was that passing a uint8 array to a
colormap makes Matplotlib assign values like 256, 257, 258. These are out of
range for uint8, causing NumPy 1.24+ deprecation warnings. The fix casts xa to
int. Testing showed no more warnings. ...
Test Content:
...
Question:
Q: What happens before and after applying the patch?
 ...
 C. Fail at line 22 with TypeError.
 D. Fail at line 27 with AssertionError.
 E. Test passed.
Labeler 1: C,E
Line 22 fails because Matplotlib tries to assign 256--258 into a uint8 array.
After the patch, xa is cast to int, so the test passes.
Labeler 2: D,E
The explanation says the bug emits DeprecationWarnings. Before the patch,
dep_warnings should be non-empty, so line 27 fails. After the patch, the test
passes.
Adjudicator: D,E
The explanation identifies warnings as the key buggy behavior and says the patch
yields no more warnings. It does not justify inferring a TypeError at line 22.
Thus, line 27 fails before the patch, and the test passes after the patch.
```

Figure 4: Example of a resolved human-labeler disagreement for the end-to-end question.

6 Auditing Code Explanations from Agents

Beyond evaluating explanation quality with ExplainBench, we further seek a general approach to improve explanations produced by coding agents, to better establish trust between agents and developers in future software development processes. To this end, we propose ExplanationAuditAgent, which audits explanations from arbitrary coding agents and provides a refined explanation.

Agent Operation. Given an agent patch, agent explanation, and original issue description as input, ExplanationAuditAgent first generates tests to double-check the explanation. For each generated test, ExplanationAuditAgent invokes its DiffExecution tool, which executes the generated test before and after the patch. DiffExecution enables the agent to inspect the patch effect by reporting the test results, such as process return code or `stdout` content. ExplanationAuditAgent can also invoke its InspectCallGraph tool to examine the call chain leading to the execution of a target function. Moreover, ExplanationAuditAgent has access to other tools: a bash interface, file I/O utilities, and patch application/reversion to support the testing. After conducting differential testing and call graph inspection, ExplanationAuditAgent compares

Table 1: Evaluated agent frameworks.

Project	Stars	Paper	Commercial
OpenHands/OpenHands [38]	67.3K	✓	✓
bytedance/trae-agent [34]	10.7K	✓	✓
lingxi-agent/Lingxi [42]	208	✓	–
smallcloudai/refact [5]	3.5K	–	✓
SWE-agent/mini-SWE-agent [6]	2.7K	–	–

the input agent explanation against the collected evidence from the invoked tools. If the evidence contradicts the explanation, ExplanationAuditAgent revises the explanation, explicitly noting the conflicting evidence and how it contradicts the original explanation. On the other hand, if the evidence aligns with the explanation, ExplanationAuditAgent appends a confirmation detailing the validation steps performed and explaining why the evidence supports the claim. Overall, ExplanationAuditAgent augments agent explanations with independent, execution-based evidence from testing that the issue-resolving agent may have overlooked.

7 Experimental Setup

7.1 Research Questions

RQ1: How do different agents score on ExplainBench? Our work automatically evaluates the explanation quality of agents. As such, we compare how explanations from different agents score on ExplainBench, and how it differs from agent efficacy (i.e., the number of resolving patches generated).

RQ2: What are the current problems with agent explanations? Using ExplainBench, we analyze what specific problems the current agent explanations have: are low scores on ExplainBench due to vagueness or are explanations confidently wrong?

RQ3: How does ExplanationAuditAgent improve explanations for different agents? We evaluate to what degree ExplanationAuditAgent can enhance explanations from the issue-resolving agents we evaluate, as a showcase of how explanations can be made more trustworthy.

7.2 Evaluated Agents

We evaluate a set of representative open-scaffold agent frameworks on our benchmark, as shown in Table 1. Candidate agents are first identified from the top-20 submissions on the SWE-bench Verified leaderboard[7] as of February 2026. From this pool, we apply the following selection criteria. First, the agent must either demonstrate substantial community adoption, as reflected by its GitHub star count, or be accompanied by a publicly available paper or technical report. Second, complete issue-resolving trajectories for the agent must be publicly available. Third, for at least 90% of bug instances, the agent's execution trajectory must include a patch explanation in the final tool invocation step. In the end, we included the five open-scaffold agents in Table 1.

7.3 Implementation Details

7.3.1 Prompt Design. We evaluate patch explanations by prompting an LLM in a question-answering format. Fig. 5 illustrates the

```
An AI agent fixed a bug in a code repository and provided an explanation for the
patch. You will be given this patch explanation, and your task is to answer
questions about the bug and patch described by the explanation. ...

Patch Explanation:
*** Explanation Start ***
[explanation]
*** Explanation End ***

[context]

Question:
[question]
```

Figure 5: Question-answering prompt template.

Table 2: SWE-bench Verified Issue Exclusion Categories

Reason	# Excluded
SWE-bench harness failing with developer patches	13
Complexity of traced program execution	
- Large traces (can be up to 70 GB for one test case)	6
- 100% CPU usage	1
- Timeout (after 6 hours)	14
Idiosyncratic tests in SWE-bench Verified	
- Buggy program failing before test function is called	14
- Hard-to-handle dynamic features in Python	12
- Repository-specific custom test framework features	7
Fragile program states (logging causes test failure)	67
Question quality control	
- Delta behavior in variables with random values	16
- Delta behavior in fields in deeply nested objects	30
- Expression generation failure after multiple trials	23
Total	203

prompt template in ExplainBench, which is applied across all question types. For each specific question type, we populate the template with an explanation, the corresponding context, and the multiple-choice question, as introduced in Section 4.4. To populate the `[context]` placeholder, we use the generated PBTs for end-to-end questions; for local questions, the pre-patch function containing delta behavior, its inputs, and the corresponding line of divergence. For the `[question]` placeholder, Fig. 3 presents an example for each question type.

7.3.2 Benchmark Statistics. In this work, we reuse bug instances from SWE-bench Verified [8], a widely-adopted benchmark of 500 real-world software engineering issues. Although SWE-bench Verified is human-validated, a small number of bug instances fail to pass the test suites even when developer patches are applied, as reported by many including Anthropic[8]. Therefore, we exclude these instances in our study. To ensure reliable instrumentation, we further exclude instances whose execution exceeds practical tracing limits due to excessive traces, runtime, CPU utilization, or other problems. Based on these factors, we construct ExplainBench based on 297 bug instances. The reasons for excluding the remaining bug instances are summarized in Table 2. Specifically, "fragile program states" refer to implicit side effects during serialization-based logging. We acknowledge this and the tracing limits as limitations of the data construction pipeline. Nevertheless, we verify that the

[7] https://www.swebench.com

[8] https://www.anthropic.com/news/claude-3-7-sonnet

exclusion of bugs does not lead to statistically significant differences between ExplainBench and SWE-bench in terms of project composition, human-rated difficulty, and patch size.

7.3.3 Configurations and Environment. To obtain patch explanations and efficacy scores for the agents, we collect agent trajectories and test execution logs from https://github.com/SWE-bench/experiments. GPT-5.2 [28] was used to generate bug-reproducing PBTs and candidate expressions; GPT-5-mini [27] is used as the question-answering LLM. We use the weaker GPT-5-mini model to answer the questions as we want the model to rely on the explanation, whereas better models may use their superior knowledge to answer the questions. ExplanationAuditAgent also uses GPT-5-mini. All models are accessed via OpenAI API with default options: thus, the question answering LLM, GPT-5-mini, uses a temperature of 1.0 and medium reasoning effort. For each question, we generate the answer five times and report the average explanation score across these generations.

8 Results

8.1 RQ1: Agent Explanation Scores

- **Patch explanation quality is distinct from patch generation efficacy.** Our evaluation of the explanations of five agents on ExplainBench, along with the efficacy scores on SWE-bench, is presented in Table 3. Note that higher efficacy on SWE-bench does not necessarily correspond to better explanation quality. OpenHands, which ranks fourth in efficacy, achieves the highest explanation score among all evaluated agents. In contrast, trae-agent, which attains the highest efficacy, ranks only fourth in explanation quality. These shifts in ranking highlight the importance of independently evaluating explanation quality.
- **Results from ExplainBench are stable.** Although our benchmark involves an LLM in the evaluation process, the results from the benchmark are stable, with a standard error of the mean of less than 0.01 for the explanation scores in Table 3. Furthermore, running ExplainBench with a different question-answering LLM (GPT-5-nano) yields an identical explanation score ranking; results for this experiment can be found in our supplementary material.
- **Explanations describe end-to-end behavior better than local behavior.** We find that for both intent and effect, end-to-end scores are consistently higher than local scores across all agents. This indicates that current agent explanations are more informative at global rationales than localized code-level reasoning.

8.2 RQ2: Agent Explanation Problem Analysis

We analyze explanation quality further through two failure modes: *uninformative* explanations and *misaligned* explanations. Uninformative explanations are cases where the LLM-selected answer is "*Explanation insufficient to answer*" (provided at the end of each question), indicating a lack of information. Misalignment refers to cases where the LLM-selected answer differs from the ground-truth answer and is not classified as uninformative, indicating that the explanation leads to an incorrect conclusion.

- **End-to-end intent in explanations is generally accurate but sometimes absent from agent explanations.** Table 4 shows that failures on the end-to-end intent explanations are dominated by non-informativeness, with misalignment remaining uniformly low. That is, when agent explanations describe what the program as a whole should do, it is generally accurate. However, they omit intent at times, in favor of reporting what the agent patch does.
- **Explanations frequently inaccurately describe local intent.** For local intent, misalignment increases substantially relative to end-to-end intent. Local intent requires precisely describing what behavior a function should show. Even when end-to-end intent is correctly described, agents appear to inaccurately infer what the developer intended at the local level.
- **Explanations are overconfident in the correctness of their patch.** For end-to-end effect, misalignment is the dominant failure case across all agents. We find that agent explanations almost always describe what the patch does, but frequently do so incorrectly. In Table 5, across all agents, 79.30% of incorrect patches are nevertheless predicted by the LLM to *pass* the bug-reproducing test in the end-to-end effect question.

8.2.1 Case Study. Fig. 6 presents a case study of a low-quality agent-generated explanation and its assessment by ExplainBench. To summarize, the intended change is to ensure a function accounts for type heterogeneity by converting all elements of '`fixture_dirs`' to strings before comparing with '`app_dir`', as shown in Fig. 6a. When the bug is fixed, an `ImproperlyConfigured` exception should be triggered when `app_dir` is included in `fixture_dirs`. Instead, the agent patch changes the irrelevant logic of duplicate entry detection in the function, and thus does not fix the bug.

Our main focus, the explanation in Fig. 6c, is also problematic, which ExplainBench automatically reveals. For example, the explanation is uninformative in *end-to-end intent*: it only notes that duplicate detection logic is now 'properly handled', without referencing any specifics of the intended behavior change. Accordingly, when ExplainBench evaluates this explanation with respect to end-to-end intent, the LLM-selected answer is that the explanation is insufficient to complete the PBT. Meanwhile, the end-to-end and local effect descriptions in the explanation are *misaligned* as the explanation falsely claims to have fixed the issue.When ExplainBench evaluates the explanation for the end-to-end and local effect questions, the LLM-selected answers are that the patch makes the test pass and that the patch leads to a change in return values; both are wrong as the patch has no effect. Thus, the explanation is marked as misaligned in effect by ExplainBench.

8.3 RQ3: Improving Explanations

We apply ExplanationAuditAgent to improve patch explanations for all agents evaluated in RQ1, at a cost of $0.05 per explanation on average. The results of running ExplanationAuditAgent are shown in Table 6. The auditing process improved the explanation score for all agents evaluated, with notable improvement for end-to-end questions. This is expected, as ExplanationAuditAgent primarily modifies explanations by executing tests, which are an end-to-end interaction with the program. While the expectation when developing ExplanationAuditAgent was that the effect score would improve, there was also notable improvement for the end-to-end intent scores of all agents. We find that as ExplanationAuditAgent runs tests, it reasons about the expected behavior

Table 3: Evaluation results of agents on ExplainBench, compared with efficacy results on SWE-bench Verified. For explanation quality, we report scores on four individual evaluation components, and the final explanation scores with standard error of the mean (SEM). The same subset used in ExplainBench evaluation is used to evaluate efficacy. The numbers after the # character indicate the agent ranking in the respective column.

Agent	ExplainBench					SWE-bench Verified
	End-to-End		Local		Expl. Score	Efficacy
	Intent	*Effect*	*Intent*	*Effect*		
refact	0.723	0.818	0.355	0.490	0.596 ±0.006 (#2)	0.778 (#3)
Lingxi	0.704	0.805	0.368	0.492	0.592 ±0.005 (#3)	0.785 (#2)
OpenHands	0.723	0.791	0.353	0.522	0.597 ±0.003 (#1)	0.727 (#4)
trae-agent	0.636	0.805	0.332	0.457	0.558 ±0.005 (#4)	0.818 (#1)
mini-SWE-agent	0.467	0.599	0.302	0.370	0.435 ±0.006 (#5)	0.599 (#5)

Table 4: Failure breakdown of agent-generated explanations. Percentages denote the fraction labeled as *misaligned* (incorrect/misleading) or *not-informative* (insufficient information).

Agent	End-to-End				Local			
	Intent		Effect		Intent		Effect	
	Mis.	*NI*	*Mis.*	*NI*	*Mis.*	*NI*	*Mis.*	*NI*
refact	4.0%	23.7%	13.7%	4.5%	37.4%	27.1%	39.8%	11.2%
Lingxi	4.6%	25.0%	14.9%	4.5%	36.2%	26.9%	41.5%	9.3%
OpenHands	3.2%	24.5%	17.9%	3.0%	35.5%	29.2%	40.0%	7.8%
trae-agent	4.6%	31.8%	14.7%	4.7%	37.3%	29.5%	39.4%	14.9%
mini-SWE-agent	3.4%	49.8%	25.4%	14.7%	29.9%	39.9%	38.9%	24.1%

```
- if app_dir in fixture_dirs:
+ if app_dir in [str(d) for d in fixture_dirs]:
    raise ...
```

(a) The intended change.

```
- if len(fixture_dirs) != len(set(fixture_dirs)):
+ normalized_dirs = []
+ for d in fixture_dirs:
+   try:
+       normalized = str(os.path.realpath(...)
+       normalized_dirs.append(normalized)
+   except (OSError, ValueError):
+       ...
+ if len(normalized_dirs) != len(set(normalized_dirs)):
    ...
  if app_dir in fixture_dirs:
    raise ...
```

(b) The agent patch.

```
### **Changes Applied:**
...
**Modified** `.../commands/loaddata.py` - Fixed the duplicate detection logic to
properly handle Path instances
...
### **Comprehensive Test Results:**
1. **Fixtures Regress Tests**: **59/59 tests passed** (including 2 new tests)
...
### **Functionality Verified:**
...
- **Mixed Path/string duplicates** are now properly detected
- **Valid Path usage** continues to work without false positives
...
The GitHub issue **"Fixture dirs duplicates undetected if dir is Path instance"**
has been **completely resolved** with a robust, backward-compatible solution that
properly normalizes paths before duplicate detection.
```

(c) Agent explanation.

Figure 6: Example of explanation misalignment.

Table 5: Overconfidence in end-to-end effect predictions. Overconfidence is the percentage of non-passing patches for which the Q&A LLM incorrectly predicts that the bug-reproducing test would pass after applying the patch.

Agent	# Not Pass	Overconfidence
refact	66	78.79%
Lingxi	64	82.81%
OpenHands	81	71.60%
trae-agent	54	79.63%
mini-SWE-agent	104	83.65%
Average	**73.8**	**79.30%**

of the program and includes it in its final explanation. Thus, it addresses the uninformative problem for end-to-end intent questions reported in Table 4.

As an example, ExplanationAuditAgent improved the explanation in the case study of RQ2. As noted before, the explanation claims the bug is fixed when it is not. ExplanationAuditAgent identifies this discrepancy through testing, then adds to the original explanation the following:

> *(important caveat): ... If FIXTURE_DIRS contains a pathlib.Path object equal to app_dir, the patch will not detect it and will not raise the ImproperlyConfigured error. I reproduced this behavior: (...)*

This refined explanation accurately describes the patch effect and intended behavior. As such, the end-to-end questions of ExplainBench are now accurately answered, improving explanation score.

Table 6: Improvement of explanations on ExplainBench score after ExplanationAuditAgent auditing. The numbers in parentheses represent the score gain relative to non-audited explanations as shown in Table 3.

Audited Agent	End-to-End		Local		Expl. Score
	Intent	*Effect*	*Intent*	*Effect*	
refact	0.768 (+6.2%)	0.837 (+2.3%)	0.359 (+1.1%)	0.504 (+2.9%)	0.617 (+3.4%)
Lingxi	0.768 (+9.1%)	0.813 (+1%)	0.362 (-1.6%)	0.501 (+1.8%)	0.611 (+3.2%)
OpenHands	0.774 (+7.1%)	0.806 (+1.9%)	0.366 (+3.7%)	0.516 (-1.1%)	0.616 (+3.1%)
trae-agent	0.782 (+23.0%)	0.826 (+2.6%)	0.361 (+8.7%)	0.512 (+12%)	0.620 (+11.3%)
mini-SWE-agent	0.718 (+56.1%)	0.731 (+22.0%)	0.362 (+19.9%)	0.510 (+37.8%)	0.580 (+33.5%)

To better interpret the gains from ExplanationAuditAgent, we sampled 60 explanations from the two agents most affected by ExplanationAuditAgent, namely mini-SWE-agent and trae-agent, and manually categorized the changes introduced by ExplanationAuditAgent. The results show that ExplanationAuditAgent generally added information about "*how the code behaves before and after the patch*" (46%), "*which parts of the code remained unchanged*" (26%), and "*edge cases*" (5%). Also, in 17% of the cases, ExplanationAuditAgent added example values. These findings suggest that ExplanationAuditAgent improves not only performance under the test-oriented scoring format but also the usefulness of explanations for developer comprehension.

9 Implications

Based on our empirical findings, we make the following recommendations to agent developers interested in improving the quality of their agent's explanations.

- **Structurally encourage explanations.** A key finding from ExplainBench is that agents such as OpenHands can generate better explanations despite having lower patch resolution rates. Upon analyzing OpenHands, we find that OpenHands architecturally enforces explanations through its `finish` tool, which must be run to submit a solution. The tool description states that `finish` should be provided with a `message` parameter which includes "a clear summary of actions taken and their results". In contrast, trae-agent, which has a generally similar architecture to OpenHands, critically does not include a `message` parameter in its `finish` tool. This leads to explanations being short or completely absent.
- **Clearly describe explanation desiderata in the system prompt.** The system prompt also plays an important role in determining the quality of explanations. The agent with the second-highest score on ExplainBench, refact, directs the agent to "post a concise, bullet-point summary that includes the suspected root cause". In contrast, mini-SWE-agent does not mention the need of explanations anywhere in its prompt. Along with the fact that mini-SWE-agent does not structurally encourage explanations, this leads to mini-SWE-agent having the worst explanation score among evaluated agents.
- **Employ an independent explanation auditing sub-agent.** As RQ3 shows, the explanations of all agents improved in quality when validated by ExplanationAuditAgent. Thus, especially for multi-agent systems such as Lingxi, it is worth considering including a sub-agent dedicated to auditing the natural language explanation of a coding agent.

10 Threats to Validity

External Validity. The benchmark and findings of this work are limited to a set of open-source projects from SWE-bench Verified, which contains repositories with regression test suites and reproducible failures. Nonetheless, the benchmark framework is not inherently limited to SWE-bench and can be extended easily.

Internal Validity. Explanation scores are influenced by stochastic variation. To minimize this threat, we run each experiment five times and report aggregated results. As shown in Table 3, the standard error of the mean is low, indicating that the results are stable.

Construct Validity. Explanation score captures only a subset of dimensions relevant to explanation quality, such as behavioral alignment, and does not assess qualitative aspects including readability or usefulness. While acknowledging these limitations, we adopt this design choice to enable easily scalable and reproducible evaluation.

11 Conclusion

As LLM agents play a greater role in software development, the explanations they generate will play an increasingly critical role in determining developer trust. Despite this, a systematic evaluation of their quality has not been done. In this work, we present ExplainBench, a benchmark to automatically evaluate agent explanations. ExplainBench is based on the intuition that informative explanations would help correctly answer questions about the bug and agent patch. Evaluation on ExplainBench revealed that agent explanation quality and patch generation efficacy do not always correspond, affirming the need to evaluate explanations. Furthermore, ExplainBench revealed the specific ways in which agent explanations could be improved, such as overconfidence in patch correctness. With this in mind, we implement ExplanationAuditAgent to autonomously run tests and calibrate overconfident explanations. ExplanationAuditAgent improved agent scores on ExplainBench by 10.9% on average, suggesting a mechanism through which explanation quality can be automatically improved.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, under NRF Investigatorship Program, Award ID: NRF-NRFI11-2026-0001, "Agentic AI based Software of the future: from Scale to Trust".

Data Availability Statement

Our replication package is available at [29]. We provide a leaderboard website of ExplainBench at https://explainbench.github.io.

References

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.

[2] Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. 2025. Otter: Generating Tests from Issues to Validate SWE Patches. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=b0jYs6JOZu

[3] L. Applis, Y. Zhang, S. Liang, N. Jiang, L. Tan, and A. Roychoudhury. 2026. Unified Software Engineering Agent as AI Software Engineer. In *International Conference on Software Engineering (ICSE)*.

[4] Islem Bouzenia and Michael Pradel. 2025. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA047 (June 2025), 23 pages. doi:10.1145/3728922

[5] Alexey Braver. 2022. *Functions-When Developers Use Them and Why*. Ph. D. Dissertation. Hebrew University of Jerusalem.

[6] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 335–336.

[7] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1869–1881. doi:10.1109/ICSE55347.2025.00012

[8] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. Retrieved 27 March 2026 from https://openai.com/index/introducing-swe-bench-verified/

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

[10] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[11] Saffron Huang, Bryan Seethor, Esin Durmus, Kunal Handa, Miles McCain, Michael Stern, and Deep Ganguli. 2025. *How AI Is Transforming Work at Anthropic*. Retrieved 27 March 2026 from https://anthropic.com/research/how-ai-is-transforming-work-at-anthropic/

[12] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[13] Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2025. DSBench: How Far Are Data Science Agents from Becoming Data Science Experts?. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=DSsSPr0RZJ

[14] Thomas Joshi, Shayan Chowdhury, and Fatih Uysal. 2025. SWE-Bench-CL: Continual Learning for Coding Agents. *arXiv preprint arXiv:2507.00014* (2025).

[15] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2025. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering* 30, 2 (2025), 1–28.

[16] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, 165–176. doi:10.1145/2931037.2931051

[17] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[18] Xuan-Bach D Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 524–535.

[19] Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E Gonzalez, and Ion Stoica. 2024. From Crowdsourced Data to High-Quality Benchmarks: Arena-Hard and BenchBuilder Pipeline. *arXiv preprint arXiv:2406.11939* (2024).

[20] Tianle Li, Evan Frick Wei-Lin Chiang, Lisa Dunlap, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. 2024. From Live Data to High-Quality Benchmarks: The Arena-Hard Pipeline. https://lmsys.org/blog/2024-04-19-arena-hard/

[21] Zhixing Li, Yue Yu, Tao Wang, Yan Lei, Ying Wang, and Huaimin Wang. 2022. To follow or not to follow: Understanding issue/pull-request templates on github. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2530–2544.

[22] Jingjing Liang, Yaozong Hou, Shurui Zhou, Junjie Chen, Yingfei Xiong, and Gang Huang. 2019. How to explain a patch: An empirical study of patch explanations in open source projects. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 58–69.

[23] Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2025. Towards Explainable Vulnerability Detection With Large Language Models. *IEEE Transactions on Software Engineering* 51, 10 (2025), 2957–2971. doi:10.1109/TSE.2025.3605442

[24] Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. 2024. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=9Y8zUO11EQ

[25] Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, Sen Hu, Ronghao Chen, Bo Li, Xin Li, Chen Hu, Binxing Jiao, Daxin Jiang, and Pin Lyu. 2025. GitTaskBench: A Benchmark for Code Agents Solving Real-World Tasks Through Code Repository Leveraging. arXiv:2508.18993 [cs.SE] https://arxiv.org/abs/2508.18993

[26] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2228–2240. doi:10.1145/3510003.3510040

[27] OpenAI. 2025. GPT-5 mini Model. Retrieved 27 March 2026 from https://developers.openai.com/api/docs/models/gpt-5-mini

[28] OpenAI. 2025. GPT-5.2 Model. Retrieved 27 March 2026 from https://developers.openai.com/api/docs/models/gpt-5.2

[29] Zhiyuan Pan, Sungmin Kang, Imam Nur Bani Yusuf, and Abhik Roychoudury. 2026. Artifact for paper "ExplainBench: Evaluating Code Explanations from Agents"'. https://doi.org/10.5281/zenodo.19230708

[30] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. 2025. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703* (2025).

[31] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR* abs/2403.05530 (2024).

[32] Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 388–410.

[33] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 963–974. doi:10.1109/ICSE55347.2025.00080

[34] Trae Research Team, Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. arXiv:2507.23370 [cs.SE] https://arxiv.org/abs/2507.23370

[35] The Terminal-Bench Team. 2025. Terminal-Bench: A Benchmark for AI Agents in Terminal Environments. Retrieved 27 March 2026 from https://github.com/laude-institute/terminal-bench

[36] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering*. 2389–2401.

[37] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.

[38] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, and Mingchen Zhuge et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=OJd3ayDDoF

[39] Ratnadira Widyasari, Jia Wei Ang, Truong Giang Nguyen, Neil Sharma, and David Lo. 2024. Demystifying Faulty Code: Step-by-Step Reasoning for Explainable Fault Localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 568–579. doi:10.1109/SANER60148.2024.00064

[40] Tao Xiao, Hideaki Hata, Christoph Treude, and Kenichi Matsumoto. 2024. Generative AI for pull request descriptions: Adoption, impact, and developer interventions. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1043–1065.

[41] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://arxiv.org/abs/2405.15793

[42] Xu Yang, Jiayuan Zhou, Michael Pacheco, Wenhan Zhu, Pengfei He, Shaowei Wang, Kui Liu, and Ruiqi Pan. 2025. Lingxi: Repository-Level Issue Resolution Framework Enhanced by Procedural Knowledge Guided Scaling. arXiv:2510.11838 [cs.SE] https://arxiv.org/abs/2510.11838

[43] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. Predictive models in software engineering: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–72.

[44] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.

[45] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. 2025. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=MhBZzkz4h9

[46] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2025. SWE-bench Goes Live!. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=OGWkr7gXka

[47] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. doi:10.1145/3650212.3680384

[48] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *NeurIPS*.