AutoCodeSherpa: Symbolic Explanations in AI Coding Agents

SUNGMIN KANG*, National University of Singapore, Singapore
HAIFENG RUAN*, National University of Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Large language model (LLM) agents integrate external tools with one or more LLMs to accomplish specific tasks. Agents have rapidly been adopted by developers, and they are starting to be deployed in industrial workflows, such as their use to fix static analysis issues from the widely used SonarQube static analyzer. However, the growing importance of agents means their actions carry greater impact and potential risk. Thus, to use them at scale, an additional layer of trust and evidence is necessary. This work presents AutoCodeSherpa, a technique that provides explanations of software issues in the form of symbolic formulae. Inspired by the reachability, infection, and propagation model of software faults, the explanations are composed of input, infection, and output conditions, collectively providing a specification of the issue. In practice, the symbolic explanation is implemented as a combination of a property-based test (PBT) and program-internal symbolic expressions. Critically, this means our symbolic explanations are executable and can be automatically evaluated, unlike natural language explanations. Experiments show the generated conditions are highly accurate. For example, input conditions from AutoCodeSherpa had an accuracy of 85.7%. This high accuracy makes symbolic explanations particularly useful in two scenarios. First, the explanations can be used in automated issue resolution environments to decide whether to accept or reject patches from issue resolution agents; AutoCodeSherpa could reject 2x as many incorrect patches as baselines did. Second, as agentic AI approaches continue to develop, program analysis driven explanations like ours can be provided to other LLM-based repair techniques which do not employ analysis to improve their output. In our experiments, our symbolic explanations could improve the plausible patch generation rate of the Agentless technique by 60%.

CCS Concepts: • **Software and its engineering → Software defect analysis**; **Software testing and debugging**; *Empirical software validation.*

Additional Key Words and Phrases: LLM, Agent, Property-Based Testing

ACM Reference Format:
Sungmin Kang, Haifeng Ruan, and Abhik Roychoudhury. 2018. AutoCodeSherpa: Symbolic Explanations in AI Coding Agents. In *Proceedings of ('XX).* ACM, New York, NY, USA, 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Agents are software systems in which a large language model (LLM) autonomously invokes external tools to achieve user-specified goals. Coding agents, which help developers perform coding tasks, have attracted particular attention from both industry [19, 34] and academia [12, 36], due to their strong efficacy [24]. Since their emergence, a substantial body of work on agents has rapidly

*Equal contribution, ordered alphabetically

Authors' Contact Information: Sungmin Kang, sungmin@nus.edu.sg, National University of Singapore, Singapore; Haifeng Ruan, haifeng.ruan@u.nus.edu, National University of Singapore, Singapore; Abhik Roychoudhury, abhik@nus.edu.sg, National University of Singapore, Singapore.

'XX,

https://doi.org/XXXXXXX.XXXXXXX

been built, especially on those that resolve software issues [36, 40, 44]; these agents operate on natural language issues and perform program improvements such as bug fixes or feature additions. Industry adoption has also been rapid, with examples including Microsoft's Copilot agent [19] and AutoCodeRover [45], which is used to fix static analysis issues in the SonarQube analyzer.

The rapid deployment of coding agents and their autonomous nature also means there is a greater potential for undesirable behavior. This makes trustworthy, evidence-based explanations for coding agents increasingly important. Indeed, prior work demonstrates explainability and transparency are primary factors for human trust in coding agents [35]. The need for trustworthy explanations is also evident in real-world usage. When the Copilot agent proposed a fix for an issue in the public `.NET` repository, the lead developer was not satisfied with the patch alone and instead requested an explanation of the underlying bug, asking "what causes us to get into this situation in the first place?" [38]. More broadly, the demand for explanations is also strongly supported by prior literature on automated debugging. Noller et al. [32] find that, beyond the patch itself, the most helpful artifact an automated program repair technique can provide is an explanation of the bug. Similarly, Kochhar et al. [28] report that over 85% of developers consider it important that a debugging technique be able to provide its rationale.

With this need in mind, we propose AutoCodeSherpa, a technique that, given an issue description, generates a *symbolic explanation* for why a bug occurs. Acting like an automated Sherpa[1], the symbolic explanation provides guidance through the key factors that lead to a bug. Specifically, the explanation consists of three interrelated conditions: an *input condition*, which specifies the input space under which the bug occurs; an *infection condition*, which is an internal program state resulting from the bug; and an *output condition*, which captures the observable symptoms of the bug. Together, these conditions help developers answer the question "what causes us to get into this situation?".

To generate a symbolic explanation, AutoCodeSherpa employs a multi-agent pipeline. In the first step, AutoCodeSherpa characterizes the input and output conditions of the bug by generating a property-based test (PBT), a generalization of failing executions. The PBT captures the input-output properties observed in buggy executions, providing a black-box view of the circumstances under which the bug occurs. To understand the program's internal behavior, a second agent explores the repository's code, and a third agent synthesizes infection conditions—symbolic formulae that distinguish buggy states from normal ones. While these conditions are generated by LLMs, we apply refinement steps at each stage to heighten accuracy.

The symbolic explanations generated by AutoCodeSherpa offer several unique benefits. First, as the explanation in the form of PBT is executable, it can be executed against suggested patches to filter for likely correct ones. This enables us to determine whether a given patch resolves the issue represented by the symbolic explanation, thereby increasing trust in the patch. In contrast, existing explainable LLM techniques [26, 30] produce only natural language artifacts, which cannot be executed. Second, as future software development increasingly features agent-agent interactions, on top of human-agent interactions, the explanations generated by our approach could help other agents reason about the bug and improve the likelihood of generating useful artifacts.

We evaluate AutoCodeSherpa in five aspects: (i) the accuracy of its generated explanations, (ii) automatically assessing patch correctness, (iii) assisting other coding agents in resolving bugs, (iv) generalizing across LLMs, and (v) its scalability. We find that the input, infection, and output condition achieve high accuracies of 85.7%, 79.7%, and 79.0%, respectively, providing a basis for trust. Running its PBTs on agent patches, AutoCodeSherpa rejected 123% more incorrect patches than the best baseline. Furthermore, the explanations were informative enough to help the issue-resolving

[1]A Sherpa is a mountain guide who assists trekkers in climbing mountains.

technique Agentless [43], improving its plausible patch generation rate by 60.7%. Finally, the explanation accuracy of AutoCodeSherpa remained consistent across different LLMs, demonstrating the generality of the approach.

Overall, our contributions are:

- Introducing executable symbolic explanations in coding agents, which symbolically describe a software issue in terms of program inputs and states and can be executed.
- AutoCodeSherpa, an agentic tool that automatically generates symbolic explanations.
- Experimental evaluation showing that the symbolic explanations generated by AutoCodeSherpa are highly accurate, can distinguish correct from incorrect patches more accurately than baselines, and can improve the issue resolution efficacy of other techniques.

2 Background

2.1 Bug Characterization

Our symbolic explanations are loosely inspired by the reachability, infection, and propagation (RIP) model of failure observation [4]. The RIP model notes that for a software bug to be observable, the fault should be reached during execution (reachability), the state of the program should be incorrect (infection), and the infected state should be propagated to a statement making the state observable. This model of bugs has prominently been used in the analysis of mutation testing [16, 25]. Our tripartite formulation of input, infection, and output conditions has a rough correspondence to the RIP model.

Meanwhile, understanding bugs is an integral part of developers' day-to-day work and the first step to issue resolution. Despite this importance, there is little research on associating natural language descriptions of software issues with symbolic conditions. One naive approach to issue explanation is to prompt an LLM to generate a natural-language explanation from the issue description. Although an explanation so generated may appear coherent and help understanding to some extent, it is prone to errors from LLMs, even for issues in simple programs such as introductory-level programming assignments [9]. To the best of our knowledge, our work is the first to produce a *symbolic* explanation from natural-language issue reports.

2.2 Hoare Triple

An explanation for a bug is a description of how the bug affects the program state. A formal way of describing the propagation of program states through code is the *Hoare triple*, whose standard notation is $\{P\}\ C\ \{Q\}$, where P and Q are both properties about the program state, and C is a program. A Hoare triple is *partially correct*, if, whenever the program C starts with a state satisfying P (the *precondition*) *and* terminates, its terminal state will satisfy Q (the *postcondition*). In contrast to partial correctness, total correctness requires reasoning that the program C always terminates when the precondition P is met. As deciding program termination is beyond the scope of this paper, all Hoare triples in our work are partially correct. In our work, we formalize the relationship between conditions with Hoare triples. For example, supposing the input condition is I, output condition is O, and program is P, we want to check that the partially correct Hoare triple $\{I\}\ P\ \{O\}$ holds.

2.3 Property-Based Testing

Property-based testing (PBT) is a possible way of checking whether a Hoare triple holds. The PBT is a powerful software testing technique mainly used to find logical bugs. Beginning with the QuickCheck [15] framework for Haskell, PBT frameworks have been developed in popular programming languages like Python [29] and Java [21]; PBTs are mature enough to have had

```python
from hypothesis import given, assume
from hypothesis.strategies import floats

# PBT for a function `reciprocal()`
@given(floats(allow_nan=False, allow_infinity=False)) # generator
def test_reciprocal_property(x):
    assume(x != 0) # precondition
    # lines 9-10 are the property
    result = reciprocal(x)
    assert abs(x * result - 1.0) < 1e-9
```

Fig. 1. An example PBT

successes in uncovering real bugs in industrial contexts [6, 11, 20, 22]. In PBT, there is a *property* to be checked, which is an executable specification of the program-under-test. The property often contains a *precondition* that specifies the valid domain of inputs. The PBT framework automatically checks the property on a large number of randomly or semi-randomly generated inputs, which are produced by a *generator* and filtered by the precondition. In our context, to check whether the Hoare triple $\{I\}\ P\ \{O\}$ holds, we use a PBT to repeatedly sample inputs that satisfy I, execute P, and check if O holds.

In Figure 1, we show a simple example PBT written in the Python Hypothesis [29] framework. In the example, the program-under-test is a function `reciprocal` calculating the reciprocal of a floating point number. The property being tested is that a number multiplied by its reciprocal is equal to one. The precondition for the test is that the number is non-zero, and thus the generator simply produces all non-zero floating point numbers except `NaN` and infinity.

PBTs stand at a useful middle ground between traditional software testing and formal methods. On one hand, because PBTs execute a large number of inputs, they are generally more rigorous than the most commonly used example-based testing, which only checks a few inputs picked by the developer. On the other hand, PBTs are usable on a wider range of programs and can be efficiently computed. PBTs have the further benefit of being syntactically similar to example-based tests, making them accessible to developers. This is in contrast to deductive verification methods, which require specialized mathematical knowledge and thus have a higher entry barrier.

2.4 Coding Agents

In this work, we highlight the use of symbolic explanations for coding agents. Coding agents perform software engineering tasks by leveraging LLMs to autonomously invoke external tools. Since 2024, a variety of coding agents have emerged, differing in their levels of autonomy and their choices of tools. Representative examples include:

- AutoCodeRover [45]. AutoCodeRover follows a two-stage workflow consisting of context retrieval and patch generation. During context retrieval, it autonomously invokes a tool to search the program's abstract syntax tree for relevant code snippets. In the patch generation stage, it produces a code change to resolve the issue.
- SpecRover [36]. SpecRover is an extension of AutoCodeRover which achieves high precision by writing a reproducer test and observing its execution result with its patch applied. In this work, we use SpecRover as a baseline in patch validation capability due to its emphasis on precision.
- Agentless [43]. Agentless is a coding agent with minimal autonomy. It employs a three-phase process consisting of localization, repair, and patch validation. In each phase, the prompts to the LLM follow fixed templates. Due to its simplicity, Agentless is sensitive to input variations. Therefore, we evaluate the quality of our symbolic explanations by providing them to Agentless and observing the resulting gains in issue resolution efficacy.

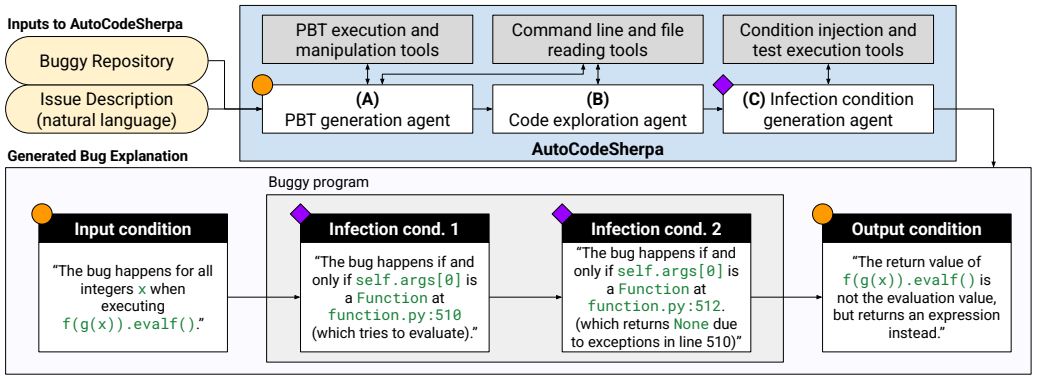


Fig. 2. Overview of AutoCodeSherpa with a real example simplified for clarity; PBT is for property-based test.

```
500 def _eval_evalf(self, prec):
501     # Lookup mpmath function based on name
502     fname = self.func.__name__
503     try:
504         if not hasattr(mpmath, fname):
505             from sympy.utilities.lambdify import MPMATH_TRANSLATIONS
506             fname = MPMATH_TRANSLATIONS[fname]
507         func = getattr(mpmath, fname)
508     except (AttributeError, KeyError):
509         try:
510             return Float(self._imp_(*self.args), prec)  # <- BUG IS HERE
511         except (AttributeError, TypeError, ValueError):
512             return
513 ...
```

Fig. 3. The buggy function involved in our running example.

- OpenHands [40]. OpenHands is a flexible agent capable of using a variety of tools to execute diverse tasks. For coding tasks, it primarily uses a bash tool to execute terminal commands and a text editor tool to view and modify code. OpenHands has no predefined workflow, and thus has the highest level of autonomy among the agents in this list.

While coding agents are becoming increasingly effective at resolving software issues, as demonstrated by open benchmarks [1], the code changes they produce may still be incorrect, even when validated against a limited number of test cases [42]. This challenge is long-standing and is known in the automated program repair (APR) community as the overfitting problem [33]. To make coding agents more trustworthy, we propose symbolic explanations that provide coding agents with additional information to improve their effectiveness in resolving software issues. Moreover, these symbolic explanations can be executed to validate the code changes produced by coding agents. We evaluate both of these use cases (patch validation and improving issue resolution) in this work.

3 Overview

Figure 2 presents an overview of AutoCodeSherpa, along with a running example used throughout this section. As illustrated in the upper part of the figure, AutoCodeSherpa takes as input a natural language issue description and the corresponding buggy program, and produces an explanation of

the issue. Our goal is to explain why a bug occurs in a symbolic manner, inspired by the reachability–infection–propagation model described in Section 2.1. To this end, AutoCodeSherpa employs LLM agents to identify three complementary conditions:

- Input condition, which specifies the set of program inputs that trigger the issue;
- Infection condition, which describes buggy internal program states;
- Output condition, which describes an observable fault.

AutoCodeSherpa also ensures that the conditions are connected by program execution: on the buggy program, execution on an input satisfying the input condition reaches a state satisfying the infection condition and then exhibits the fault specified by the output condition. This connection is checked by executing a PBT consisting of the input and output conditions while observing the values of the infection condition. A formal definition of the conditions is provided in Definition 1.

To illustrate our approach, we present a running example highlighting the high-level operations of AutoCodeSherpa. The issue references a Stack Overflow post titled "Why can't I evaluate a composition of `implemented_functions` in SymPy at a point?" The root cause resides in line 510 of SymPy's `_eval_evalf` function, shown in Figure 3, while the symbolic explanation generated by AutoCodeSherpa is shown in the lower half of Figure 2.

To explain the issue, AutoCodeSherpa first runs a PBT generation agent (Figure 2(A)). The agent encodes the input condition as a Python function that generates a set of inputs, and the output condition as an assertion statement. It then checks that all generated inputs trigger the assertion failure. In our example, the PBT asserts that for all integers `x` within a certain range, `f(g(x))` produces an `Expression` object instead of an expected concrete value.

Further, to explain how the bug propagates inside the program, AutoCodeSherpa generates infection conditions—first-order logic formulae evaluated at specific locations to distinguish buggy states from normal ones. The code exploration agent (Figure 2(B)) identifies these locations by searching the code based on the issue description and the generated PBT. In our running example, the bug report already names the buggy function (Figure 3), allowing the agent to quickly locate it in the repository as a candidate location. We note that while this example issue has a single buggy location, the symbolic explanation technique applies to multi-location issues as well.

Within the identified buggy function, the infection condition generation agent (Figure 2(C)) then proceeds to produce symbolic expressions that evaluate to true only for bug-triggering inputs from the PBT. In this example, the agent generated conditions at lines 510 and 512 of the buggy function (Figure 3). Both conditions indicate that the bug occurs when the argument of a `Function` is itself a `Function`. They accurately capture the cause of the bug: the buggy line 510 fails to handle the argument, triggering the exception-handling code in line 512 that should not have been executed.

The symbolic explanations produced by AutoCodeSherpa have several practical uses. First, the PBT in the symbolic explanation can be executed against candidate patches to filter out incorrect ones, a use case evaluated in Section 7.2. Second, the infection conditions can enhance other software engineering techniques, such as coding agents, by generalizing beyond the issue description and revealing the internal logic of the bug. This use case is evaluated in Section 7.3.

4 Symbolic Explanations: Definition

In this section, we present the formal definition of our symbolic explanations of software issues, consisting of input, output, and infection conditions, as follows.

Definition 1 (Symbolic explanation). *Given a program P whose input parameters are i, an issue X on the program, a symbolic explanation for issue X is a triple (I, F_L, O) that satisfies the following:*

- *Input condition I is a quantifier-free first-order logic formula over i, and for all inputs i that satisfy I, executing P with i triggers X;*

```
### Gist of the Report
The issue is that the `evalf` method in SymPy does not recursively evaluate the result of `_imp_`...

### Concrete Bug-Reproducing Inputs

1. **Input**: `f(g(2)).evalf()`
   **Expected**: `16.00000000000000`
   **Actual**: `f(g(2))` (fails, does not evaluate recursively)

2. **Input**: `g(f(2)).evalf()`
   **Expected**: `8.00000000000000`
   **Actual**: `g(f(2))` (fails, does not evaluate recursively)
...
```

Fig. 4. Output from the generalization phase of PBT generation.

- *Output condition O is a quantifier-free first-order logic formula over the terminal state of P that holds only if X is triggered, and for which the partially correct Hoare triple $\{I\}\ P\ \{O\}$ holds; thus by starting P with inputs satisfying I, O is guaranteed;*
- *Supposing $P = C_1; C_2$ where C_1, C_2 are sequences of program statements, and L is the program location immediately after C_1, infection condition F_L is a quantifier-free first-order logic formula over the program state at L, for which the partially correct Hoare triples $\{I\}\ C_1\ \{F_L\}$ and $\{\neg I\}\ C_1\ \{\neg F_L\}$ hold, i.e., F_L is the result of symbolically propagating I to L.*

By defining symbolic explanations as above, our explanations are able to provide the following information. First, input conditions describe the set of inputs triggering a software issue. Literature on bug reports shows that providing inputs to trigger the issue is a key factor in bug report quality [8]. Second, output conditions describe how the issue influences program output, so that readers may recognize the issue when it happens. Third, infection conditions detail how the issue propagates within the program. This helps developers, as they frequently inspect program states to better comprehend issues [3, 10]. Furthermore, the three conditions are connected via program execution: on the buggy program, if the input condition holds, the program must reach a state satisfying the infection condition, from which it then reaches a state satisfying the output condition.

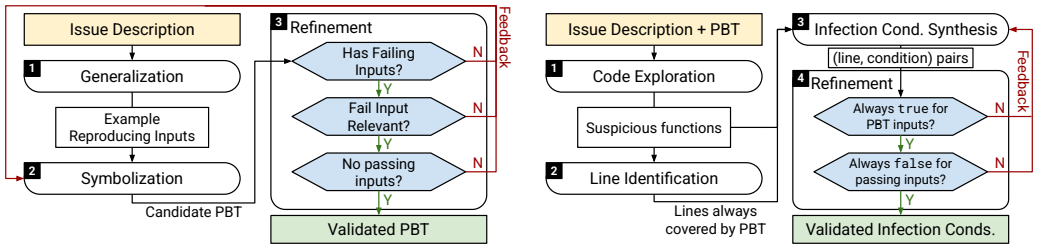
We implement each condition as executable artifacts, making our symbolic explanations executable, unlike natural language explanations. In this work, AutoCodeSherpa is implemented for Python. The input and output conditions I and O are implemented as a PBT. In particular, I is implemented as a function that generates a set of inputs, and O specifies an expected exception type. Infection conditions are implemented as first-order logic formulae, evaluated at specific lines in Python files via runtime monitoring. The PBT framework Hypothesis [29] introduced in Section 5.1 is used to repeatedly sample inputs from the input set, then check that the Hoare triples defined in Definition 1 hold.

5 Symbolic Explanations: Construction

In this section, we present our agentic approach AutoCodeSherpa for generating the symbolic explanation, which involves generating a PBT for input and output conditions and program-internal expressions for infection conditions.

5.1 PBT Generation

The first step in generating a symbolic explanation is to derive the input condition I and output condition O from an issue description, as illustrated in Figure 2. This process follows a three-step agentic workflow: generalization-symbolization-refinement, as shown in Figure 5a.



(a) PBT generation agent flowchart. (b) Infection condition generation agent flowchart.

Fig. 5. Detailed condition generation flowcharts for AutoCodeSherpa.

Generalization. To encourage generalization from the issue description, the PBT agent is first prompted to generate multiple bug-reproducing inputs based on the description, along with their corresponding actual and expected program outputs. Figure 4 illustrates the agent's response at this stage for the running example in Section 3. While these input-output pairs may be imperfect, as they are generated by an LLM, they serve as a reasoning step from the specific input in the description to the general conditions I and O, supporting the subsequent symbolization and refinement steps.

Symbolization. With example inputs generated, the PBT agent next abstracts over them to write a PBT to reproduce the issue, using symbolic expressions to represent the input and output conditions I and O. As shown in Figure 2, the agent is equipped with tools for command-line execution, file reading/writing, and PBT execution, enabling it to explore relevant files and perform trial-and-error before proposing a PBT. Once a PBT is proposed, it is executed to check if the exception specified by O is raised. If the exception is not raised, we prompt the PBT agent to retry. Note that this step only ensures the *existence* of an input i satisfying I that leads to O. To guarantee that *all* satisfying inputs lead to O, i.e., that $\{I\}\ P\ \{O\}$ holds, the PBT is refined in the next step.

Refinement. The refinement step serves to guarantee that $\{I\}\ P\ \{O\}$ holds as per Definition 1. To achieve this, two types of problems need to be detected and addressed. First, some failing inputs may fail for reasons unrelated to the described issue. To handle this, we either strengthen I to prevent the PBT from generating irrelevant inputs, or strengthen O so that these inputs do not trigger spurious failures. Concretely, we execute the PBT to collect failing inputs and their corresponding exceptions, which are then presented to the PBT agent for review relative to the issue description. If the agent determines that any failing input i is irrelevant to the described issue, the PBT is routed back to the symbolization step to generate a new PBT. Additionally, the PBT generation prompt is enriched to exclude these irrelevant failing inputs.

The second potential problem is that, even after ensuring that all failing inputs are relevant, the PBT may still generate passing inputs. To address the problem, we strengthen I to filter out these inputs. While one could also attempt to fix the problem by weakening O, we avoid doing so, as this could incorrectly reject a correct program P', reducing the usefulness of the symbolic explanation. Concretely, we execute the PBT to identify passing inputs—those that satisfy I but do not lead to O. Any passing inputs are then presented to the agent, which is prompted to revise the input condition I to exclude the inputs. The input condition can be revised in any of the following three ways: (1) adding additional `assume` statement filtering out passing inputs; (2) changing the parameters of the generator (e.g. the `allow_nan` parameter in Fig. 1), or (3) modifying the input generator function to change the sampling process itself. In practice, we find that parameter tuning and `assume` statement additions were the most common modifications. After revision, the PBT

is executed again to check for passing inputs, and the process repeats until all inputs lead to the target failure.

5.2 Infection Condition Generation

After generating the input and output conditions, AutoCodeSherpa produces infection conditions F_L to aid understanding of the program itself. As shown in Figure 5b, infection condition generation consists of four steps, namely code exploration, line identification, infection condition synthesis, and refinement.

Code Exploration. To generate infection conditions, which are defined at specific program lines, AutoCodeSherpa first finds functions relevant to the bug (suspicious functions). To do so, we reuse the context retrieval agent of AutoCodeRover [45]. This agent finds likely buggy functions by invoking a tool that searches the abstract syntax tree of the program, e.g., searching for a class or for a function in a certain class. The search starts by the agent identifying keywords from the issue statement. The result of the search would reveal relevant parts of the program, and the agent would analyze the result in relation to the issue and possibly launch another search for interesting elements in the result. This process is repeated until the agent decides that the suspicious functions have been found. At this point, the identified buggy functions and the intermediate search results are passed to the infection condition agent.

Line Identification. The infection condition agent receives the buggy functions identified by the code exploration agent and uses an LLM to select candidate code lines within these functions. The agent checks each candidate line to ensure it is executed by all inputs generated from the input condition I. Specifically, for each candidate line, the program is instrumented by adding a statement raising a custom exception just before the candidate line. After this instrumentation, the generated PBT is executed to see if the custom exception is triggered for all sampled inputs. If the custom exception is not always triggered, indicating a candidate line is not always covered, the agent provides feedback to the LLM, which proposes revised lines. After up to five iterations, the agent produces a final set of lines covered by all inputs from I.

Condition Synthesis. For each location L identified in the line identification stage, the agent presents an LLM with the buggy function and the issue description, through which the LLM generates a candidate symbolic expression F_L. Specifically, the concept of infection conditions is described to the LLM in its system prompt via the sentence "The conditions should evaluate to True for all bug-inducing inputs, and False for inputs that behave normally/are unrelated to the bug". During operation, the LLM is then prompted to "generate an infection condition at location [LOCATION]; you must generate general conditions that capture all bug-triggering inputs while excluding all passing tests." In response, the LLM generates a candidate infection condition; whether the candidate condition meets the definition of an infection condition is checked in the next step.

Refinement. A candidate infection condition can fail to meet Definition 1 in the following two ways:

- Too strong: The condition is false for at least one PBT-sampled input. That is, there exists an input i satisfying the input condition I such that the infection condition F_L does not hold at location L for at least one visit to L during execution.
- Too weak: The condition is true for at least one test unrelated to the bug. That is, there exists an input i that does not satisfy the input condition I such that the infection condition F_L holds at location L for at least one visit to L during execution.

The refinement step checks whether a candidate infection condition F_L is too strong or too weak. If F_L passes both checks, it is accepted. Otherwise, the agent attempts to refine it by incorporating feedback about the failure, together with the issue description and surrounding source code. This refinement is attempted up to five times, after which the agent moves on to the next code location. When condition synthesis has been attempted on all code lines from the line identification stage, the agent returns the successfully identified infection conditions.

This completes the construction of a symbolic explanation, consisting of a PBT (representing input and output conditions) and infection conditions. As a final step to achieve easier presentation, one can direct an LLM to convert the symbolic explanations to a natural language report. One can trace information in this report back to the conditions identified by AutoCodeSherpa, which have the properties outlined in Definition 1, unlike natural language explanations generated by LLM prompting alone. The report can be consumed by a human developer or another coding agent.

6 Experimental Setup

We evaluate the following research questions:

RQ1: How accurate are the input, output and infection conditions in the explanations?
RQ2: Are the generated PBTs useful for filtering incorrect patch candidates?
RQ3: Can the symbolic explanations improve the efficacy of issue resolution techniques?
RQ4: Does our symbolic explanation generation procedure generalize to different LLMs?
RQ5: How does symbolic explanation generation scale on repositories and bug complexity?

Benchmark. To evaluate our approach, we use the SWE-bench Verified benchmark [14]. SWE-bench Verified is a subset of SWE-bench [23] and consists of 500 issues from 12 open source repositories that were labeled as solvable by humans. Despite this labeling, we observe that some issues in SWE-bench Verified are under-specified, in that the expected program behavior is not fully described in the issue text. Each issue includes a natural-language description, which AutoCodeSherpa uses as input to generate explanations.
SWE-bench Verified provides two types of test cases: fail-to-pass (F2P) and pass-to-pass (P2P). F2P test cases are intended to reproduce the issue: they fail on the buggy program and should pass on the fixed version. These tests are not available to LLM agents (including AutoCodeSherpa) and are used only for evaluation. In contrast, P2P test cases are regression tests that pass on both the buggy and fixed programs and are available to LLM agents.

RQ1: Accuracy of Conditions. In RQ1, we report how often AutoCodeSherpa generates a symbolic explanation for issues in the benchmark. A symbolic explanation is not produced if the LLM fails to generate an input, output, or infection condition that passes the refinement steps described in Sections 5.1 and 5.2. For the explanations that are successfully generated, we evaluate the correctness of each condition using the following procedure:

- Input and output conditions are considered correct if the PBT they form fails on the buggy program and passes on the fixed program. If this is not the case, we manually examine the input and output conditions to decide their correctness.
- Infection conditions are evaluated using SWE-bench Verified test cases: an infection condition is considered correct if and only if it evaluates to true on all F2P tests and false on all P2P tests.

RQ2: Patch Validation Capability. In RQ2, we investigate whether the symbolic explanations from AutoCodeSherpa can be used to filter out incorrect patches produced by coding agents. As an example, we evaluate patches from the precision-focused agent SpecRover [36]. We report how frequently AutoCodeSherpa could correctly distinguish correct patches from incorrect ones.

Ground-truth. To determine patch correctness, we first run the P2P and F2P test cases provided by SWE-bench Verified. Patches that pass the tests are then manually checked for semantic equivalence with the human patch from the benchmark. To do this, two authors independently compared the agent-generated and human patches, then discussed and resolved disagreements to reach a final verdict on the agent patch's correctness. Both are experienced Python developers (10+ and 5+ years) familiar with the SWE-bench repositories, which aided the assessment.
Prior to discussion, the two authors achieved 92.5% agreement on the correctness of the 259 plausible SpecRover patches, with a Cohen's kappa of 0.81 (almost perfect agreement). Most comparisons were straightforward: 89 (34.3%) were syntactically identical or easily transformable to the developer patch, while most of the remaining 170 had small diffs (a median of 6 lines).

Baselines. We compare AutoCodeSherpa against three baselines: SpecRover, OpenHands [40], and OpenHands-PBT. SpecRover predicts patch correctness based on the issue description and an agent-generated example-based reproducer test. Patches produced by SpecRover, along with their validation results, are taken from its latest public data [1]. OpenHands is a general-purpose coding agent with state-of-the-art efficacy in generating bug-reproducing tests in SWT-bench Verified [2] among publicly available data. In particular, the best publicly available results from OpenHands on the SWT-bench were obtained using GPT-5-mini, which we also use throughout RQ1–RQ3 for a fair comparison. An OpenHands test is deemed to validate a patch if it passes after the patch is applied. The third baseline, OpenHands-PBT, modifies the system prompt of OpenHands by adding the instruction "The tests you write must be property-based tests using the Hypothesis framework. If `hypothesis` is not already installed in the repo, `pip install hypothesis` before running the tests.", which led it to consistently generate PBTs. PBTs from OpenHands-PBT are executed the same way as those from AutoCodeSherpa. The classification accuracy of all three baselines is then evaluated against the ground-truth correctness labeled above.

RQ3: Efficacy Gains in Issue Resolution. In RQ3, we evaluate an agent-agent interaction scenario, which may become more common as coding agents grow in prevalence. Specifically, we examine whether the symbolic explanations generated by AutoCodeSherpa improve the efficacy of Agentless in fault localization and issue resolution. For bugs where input, infection, and output conditions were all generated, an LLM converted the symbolic explanation into a natural language report, as described in Section 5. This report was then supplied as additional information to Agentless, and efficacy was compared against default Agentless (without explanations) on the same set of bugs. We choose Agentless because its simple structure makes it more sensitive to the quality of explanations than other techniques. Issue resolution efficacy is measured via the plausible patch rate, defined as the proportion of patches that pass the developer-written reproducer test. We evaluate all ten patches that Agentless generates during its patch generation process.
To assess the impact of explanations, we append the explanation to the bug report as follows: "In addition, a trustworthy process has provided the following explanation for the bug: {AutoCodeSherpa explanation}". Agentless efficacy under this setting is compared with a no-explanation setting.

RQ4: Generalization across LLMs. In RQ1–RQ3, we reported results for AutoCodeSherpa generated with GPT-5-mini, the same LLM used by our best baseline, OpenHands. In RQ4, we repeat the experiments from RQ1–RQ3 using several other LLMs to assess the generalizability of AutoCodeSherpa across different models. In particular, we evaluate AutoCodeSherpa on Claude-3-5-Sonnet, a closed-weight LLM from a different provider, and DeepSeek-v3.2, an open-weight model.

RQ5: Scalability across Repositories and Issue Complexity. In RQ5, we evaluate whether AutoCodeSherpa performs consistently across repositories of different sizes (by lines of code) and issues of different patch complexities (by the number of changed lines and functions in the developer

patch). Analyzing the RQ1 results, we track three indicators—PBT correctness, infection condition correctness, and total cost (USD)—and report the 95% confidence interval of each mean. For the binary indicators (PBT and infection condition correctness) we use the binomial standard error $\sqrt{\hat{p}(1-\hat{p})/n}$, and for continuous indicators (cost and runtime) the Bessel-corrected standard error of the mean, each scaled by 1.96.

Parameters. The PBT agent was allowed at most 50 LLM calls to limit time and cost. For the code exploration agent, up to 50 invocations of the search tool were permitted, which is its default setting [7]. The infection condition agent could suggest lines at most five times, and for each suggested line, the infection condition was iteratively improved up to five times. For fair comparison with OpenHands in RQ2, the experiments in RQ1–RQ3 were performed on GPT-5-mini. A temperature of 0 is used for DeepSeek-v3.2 and Claude-3-5-Sonnet (temperature cannot be set for GPT-5-mini). The programs under test were set up using the official Docker images of SWE-bench Verified, with a memory limit of 6 GB per container.

7 Evaluation

7.1 RQ1: Accuracy of Conditions

In RQ1, we evaluate the accuracy of the generated input, infection, and output conditions, as a necessary step in understanding the usability of the approach.

Figure 6 presents the ratio of successful PBT generations, including both input and output conditions, alongside the accuracy of the infection conditions. We note:

- First, the PBTs generated by AutoCodeSherpa were highly accurate. A PBT specifying both the input and output conditions was generated for 433 of the 500 issues (86.6%) in SWE-Bench Verified. In these PBTs, 85.7% of the input conditions and 79.0% of the output conditions were correct. In 318 (73.4%) PBTs, both conditions were correct.
- The infection conditions generated by AutoCodeSherpa were similarly accurate. Out of the 433 issues with a PBT, a total of 1001 infection conditions were generated for 289 issues (an average of 3.46 infection conditions per issue), of which 798 (79.7%) were correct.
- The generated symbolic explanations are thus highly accurate, and would likely be sufficient for developers to trust AutoCodeSherpa's results. Prior studies on automated debugging suggest that roughly 75% accuracy is considered trustworthy by about 75% of developers [28].

Despite the high overall accuracy, we analyze the circumstances under which AutoCodeSherpa produces imperfect results. First, the infection condition generation rate leaves room for improvement: infection conditions were not generated for 33.3% of all issues with generated PBTs. One challenge is the opacity of objects in certain projects. For instance, the Django project—which accounts for 46% of the SWE-bench Verified benchmark—uses deeply nested objects, where buggy states can be subtle, making infection conditions harder to generate. Consequently, the infection condition generation rate for Django was lower compared to repositories such as matplotlib, which achieved a 90% generation rate. Nonetheless, AutoCodeSherpa still generated infection conditions for a non-trivial 62% of Django issues. Future work could improve infection condition generation by leveraging serialization libraries[2] to expose program state and guide condition synthesis.

Second, output conditions are the most difficult to generate accurately and have the lowest accuracy of 79.0% among the three condition types. Some issue descriptions only cover the buggy behavior, providing hints about the input and infection conditions but leaving the intended behavior implicit or incomplete, which makes it difficult to generate correct output conditions.

[2]https://github.com/explosion/srsly

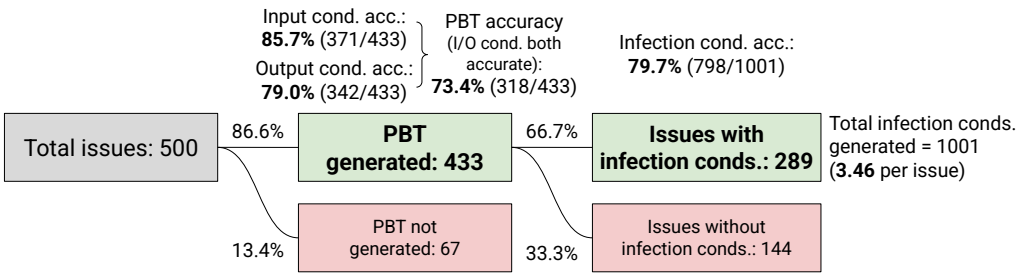
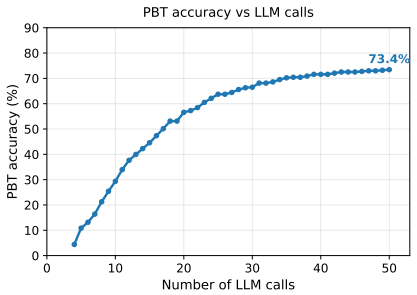
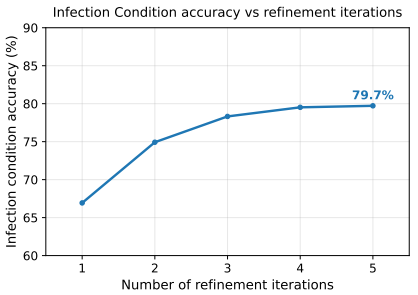


Fig. 6. A plot of generation ratio and condition accuracy.

Impact of LLM Calls and Refinement Iterations. We also examine how the number of LLM calls and refinement steps affects the accuracy of generated PBTs and infection conditions. As shown in Figure 7, the accuracy of generated PBTs increases steadily and plateaus after about 40 LLM calls, while the accuracy of generated infection conditions plateaus after 4 refinement steps. These results suggest that the chosen number of LLM calls and refinement steps for the PBT agent and the infection condition agent in Section 6 are sufficient for ensuring accuracy of the symbolic explanations.



(a) PBT accuracy across LLM calls, starting from the minimum number of calls the PBT agent needs to terminate.



(b) Infection condition accuracy across refinement iterations.

Fig. 7. Effect of LLM calls and refinement iterations on explanation accuracy.

Answer to RQ1: AutoCodeSherpa is capable of generating accurate symbolic explanations for a large proportion of issues.

7.2 RQ2: Patch Validation Capability

Patch Validation Procedure. RQ2 evaluates the capability of our symbolic explanations to validate agent-generated patches. To do so, a patch is first applied to the buggy program, and the PBT from the explanation is executed. The PBT framework repeatedly samples bug-triggering inputs and runs the PBT; if a test failure still occurs with the patch applied, the patch is considered invalid.

Metrics. The symbolic explanation can be viewed as a binary classifier—patches that pass the PBT are classified as valid (positive), while those that fail are classified as invalid (negative). To assess its efficacy, we evaluate the confusion matrix of the classifier— i.e., true positive (TP), false positive

(FP), true negative (TN), and false negative (FN)—along with derived metrics such as precision and recall. As mentioned in Section 6, evaluation is performed on patches generated for SWE-bench Verified [14] by SpecRover [36], an LLM agent designed to improve patch precision.

	AutoCodeSherpa	SpecRover	OpenHands	OpenHands-PBT
#TP ↑	132 (33.3%)	**147 (37.1%)**	136 (34.3%)	134 (33.8%)
#TN ↑	**105 (26.5%)**	29 (7.3%)	47 (11.8%)	41 (10.4%)
#FP ↓	**140 (35.4%)**	216 (54.5%)	198 (50.0%)	204 (51.5%)
#FN ↓	19 (4.8%)	**4 (1.0%)**	15 (3.8%)	17 (4.3%)
Total	396			
False positive rate = FP / (TN+FP) ↓	**57.1**%	88.2%	80.8%	83.5%
Precision = TP / (TP+FP) ↑	**48.5**%	40.5%	40.7%	41.1%
Recall = TP / (TP+FN) ↑	87.4%	**97.4**%	90.1%	93.4%

Table 1. Confusion matrices of AutoCodeSherpa and baselines, where positive means correct patch.

PBT only checks for exception	53 (36.4%)
Issue description incomplete	51 (36.4%)
Developer patch goes beyond issue scope	36 (27.2%)

Table 2. Reasons for FP patch categorization

Table 1 shows the confusion matrix and related metrics for AutoCodeSherpa and the baselines. A total of 396 issues are included—those for which AutoCodeSherpa, SpecRover, OpenHands, and OpenHands-PBT all generated a test. Focusing on the 245 (= TN + FP) incorrect patches, AutoCodeSherpa invalidated 105 of the incorrect patches (TN), i.e., the PBT failed after their application. This number of true negatives is 262.0% higher than SpecRover, 123.4% higher than OpenHands, and 156.1% higher than OpenHands-PBT, indicating that AutoCodeSherpa also achieves a much lower false positive rate than all three baselines.

To understand the 140 false positives (FP)—i.e., incorrect patches not flagged by AutoCodeSherpa—we performed a manual analysis and summarized the reasons in Table 2. Among these 140 FPs, 53 PBTs only included checks for an exception, allowing any patch that resolved the exception to pass. This reflects the well-known overfitting problem in program repair, where weak test oracles can permit plausible but incorrect patches [33]. For 51 issues, the issue description incompletely specifies the expected behavior—it describes only the buggy behavior but not the precise expected outputs when the bug is fixed. The PBT could not invalidate these patches because it only checks for the buggy behavior. Applying the regression tests of the program under test could further restrict patch behavior and potentially filter out 11 of these 51 incorrect patches. Finally, for 36 issues, the patch correctly handles the reported issue (and is thus flagged as correct by the PBT), but the developer patch goes beyond the reported issue and makes additional changes to the program. In these cases, AutoCodeSherpa correctly allows the patches to pass according to the issue description.

Meanwhile, among the 151 (=TP+FN) correct patches, AutoCodeSherpa produced 19 false negatives (FN). An examination of their issue descriptions shows that 3 issues are uninformative: they provide little or no description of the buggy behavior and instead directly describe a bug fix. Due to the insufficient information, generating accurate PBTs for such issues is difficult. Another issue describes the buggy and expected behaviors using images, whereas our agent processes text only. For the remaining 15 FN issues, the descriptions are informative, but the generated PBTs are

incorrect. These errors are caused by LLM hallucinations and do not exhibit a common pattern. Despite these FN cases, AutoCodeSherpa achieves a recall comparable to that of the baselines.

Finally, we also calculate precision. Precision is important as it captures how much of developers' effort in reviewing agent-generated patches would be spent on actually correct patches. As shown in Table 1, AutoCodeSherpa has a clear margin in precision over the SpecRover, OpenHands, and OpenHands-PBT baselines as well.

Answer to RQ2: Compared to the SpecRover, OpenHands, and OpenHands-PBT baselines, AutoCodeSherpa is more effective at identifying incorrect patches while achieving higher precision.

7.3 RQ3: Efficacy Gains in Issue Resolution

In RQ3, we evaluate the efficacy gains in fault localization (FL) and issue resolution achieved by providing symbolic explanations to an example coding agent, Agentless.

	File top-1	Element top-1	Element top-any
Without explanations	65.7%	27.3%	64.4%
With explanations	**78.2%** (+19.0%)	**42.6%** (+55.7%)	**76.8%** (+18.9%)

Table 3. Agentless fault localization efficacy, with and without explanations

Table 3 shows how providing explanations from AutoCodeSherpa improves the FL accuracy of Agentless. At the file level, access to explanations increases top-1 localization accuracy (a true buggy file is ranked as most suspicious) by 19.0%. For *element*-level localization, where Agentless identifies suspicious classes and methods, top-1 accuracy improves by 55.7% with explanations. For the *top-any* metric (i.e., whether the buggy element is identified at all), explanations enable the agent to correctly identify 18.9% of elements that it would otherwise have missed.

Agentless also benefited from explanations when generating patches, as shown in Table 4. As described in Section 6, Agentless generates ten patches per issue, both when it is and is not given explanations. Considering all ten patches for all issues, access to symbolic explanations increased the number of plausible patches by 60.7%, indicating that the explanations help guide Agentless toward effective fixes. When considering whether an issue was resolved with at least one plausible patch, Agentless successfully resolved 57.8% of all issues, a 32.5% improvement compared to when no explanations were provided.

To further understand what was driving the increase in plausible patch ratio and resolved issues, we experimented with a variant explanation that only used the PBT. We found that the plausible patch % increased by 9% in this PBT-only setting, in contrast to the 60.7% increase when infection conditions were provided. This suggests the critical role that infection conditions play in bug explanation: by providing concrete locations and conditions distinguishing buggy program states, they can help other agents generate effective patches.

	Plausible Patch %	Resolved Issues
Without explanations	29.2%	43.6%
With explanations	**47.0%** (+60.7%)	**57.8**% (+32.5%)

Table 4. Agentless plausible patch and bug resolve rate, with and without explanations

Despite the efficacy gains, some issues remain unresolved. To study the reasons for failure, we investigate a sample of 30 bugs where Agentless could not generate a plausible patch despite being

given an explanation. We categorize the reasons for failure as follows. First, limitations in the benchmark itself can prevent Agentless from passing the developer test. In 30% of the sampled cases, the test checks behavior beyond what the issue describes. As a result, even when Agentless generates a patch consistent with the description, the developer test may still fail. For example, in `sympy__sympy-21930`, the issue description mentions only one class, but the developer test also tests other classes. For 46.7% of sampled cases, the explanation described how the bug occurred, but did not specify how the issue should be resolved. Remaining minor issues included (a) Agentless producing patches with syntax errors (10%); (b) the explanations being too vague (6.7%); and (c) the explanations being misleading (6.7%).

Answer to RQ3: Providing Agentless with explanations from AutoCodeSherpa considerably improved both its fault localization and patch generation efficacy.

Metric	gpt-5-mini	claude-3-5-sonnet	deepseek-v3.2
PBT Generation (RQ1) (%)	86.6	48.4	57.8
PBT Correctness (RQ1) (%)	73.4	74.4	72.2
Infection Condition Correctness (RQ1) (%)	79.7	70.3	85.1
Patch Validation Precision (RQ2) (%)	48.5	56.1	51.5
Agentless Plausible Improvement (RQ3) (%)	+60.7	+11.1	+50.2

Table 5. Evaluation of AutoCodeSherpa with different LLMs

7.4 RQ4: Generalizability Across LLMs

To demonstrate that AutoCodeSherpa generalizes to other LLMs, we ran it using different LLMs to evaluate performance, the results of which are shown in Table 5. AutoCodeSherpa could generate explanations with similar correctness, regardless of LLM: all models showed similar PBT and infection condition correctness, as well as in patch precision evaluated as in RQ2. However, the PBT generation rate differed between LLMs, indicating their capability difference. Taken together, these results suggest that even while LLMs have capability differences, the automated refinement of AutoCodeSherpa (illustrated in Figure 5) helps ensure similar quality for explanations.

To further understand the effect of the refinement steps, we analyze the difference in correctness between the *first* PBT suggested by LLMs in AutoCodeSherpa's execution trace, before being subjected to any refinement, and the *final* PBT which had been refined and passed the pipeline of AutoCodeSherpa, as shown in Table 6. Comparing the closed-weight LLM gpt-5-mini and the open-weight deepseek-v3.2, the first PBT correctness of the LLMs was different: 51.6% with gpt-5-mini, and 39.2% with deepseek-v3.2. From this lower accuracy, AutoCodeSherpa provided feedback to fix initially incorrect PBTs (Rectified in Table 6) while preventing the submission of incorrect PBTs (Discarded). Eventually, for gpt-5-mini, 433 (=477-44) final PBTs were generated; among these, 246+83-2-(44-35)=318 were correct, where 44-35 corresponds to the number of correct first PBTs that were wrongly discarded, raising the correctness rate to 318/433 = 73.4%. For deepseek-v3.2, although the correctness of the first PBTs were lower than that of gpt-5-mini, the refinement process discarded as many as 125 incorrect first PBTs and rectified 76. As a result, there were 462-167=295 final PBTs, among which 462+76-2-(167-125)=213 were correct, raising the correctness rate to 213/295 = 72.2%, the same level as that of gpt-5-mini.

Meanwhile, all three models improved the plausible patch generation rate of Agentless. deepseek-v3.2 notably showed a strong improvement of 50.2%, similar to gpt-5-mini. However, claude-3-5-sonnet showed lower improvement than the other two models. This is due to two factors. First, claude-3-5-sonnet produced less accurate infection conditions than the other models as in Tab. 5,

	First PBT		Refinement			Final PBT	
	Generated	Correct	Rectified	Degraded	Discarded	Generated	Correct
gpt-5-mini	477	246 (51.6%)	83	2	44 (35 incorrect)	433	318 (73.4%)
deepseek-v3.2	462	181 (39.2%)	76	2	167 (125 incorrect)	295	213 (72.2%)

Table 6. Effect of refinement in PBT agent. Refinement brings the correctness rate of different LLMs close. *Rectified* means the first PBT was incorrect but was changed into a correct one by the refinement step, while *Degraded* means a correct first PBT was changed into an incorrect one. *Discarded* means a PBT was initially generated but eventually not submitted, as it could not pass the refinement step.

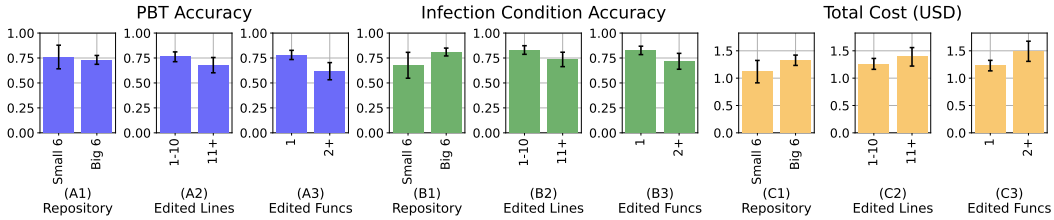


Fig. 8. PBT and infection condition accuracy over different repositories and issue complexities. The bar charts are shown with 95% confidence intervals.

increasing the likelihood of explanations misguiding Agentless. Second, when converting symbolic explanations from AutoCodeSherpa into natural language, claude-3-5-sonnet's explanations were the shortest, at half the length of gpt-5-mini's explanations on average.

Robustness against LLM non-determinism. We also reran AutoCodeSherpa to assess its robustness against LLM non-determinism. For cost reasons, we reran only gpt-5-mini and deepseek-v3.2. Table 7 compares the original and rerun results. For both LLMs, the rerun closely matches the original run across all metrics, indicating that AutoCodeSherpa can produce stable generation rates and explanation accuracy under LLM non-determinism.

Answer to RQ4 AutoCodeSherpa generalizes to different LLMs, generating symbolic explanations for all LLMs we evaluate. Furthermore, the refinement steps of AutoCodeSherpa ensure similar quality despite differences in LLM capability.

7.5 RQ5: Scalability Across Repositories and Issue Complexity

In this research question, we evaluate how AutoCodeSherpa scales across repositories of different sizes and different issue complexities. Results are shown in Figure 8. We observe the following:

AutoCodeSherpa scales across repositories of different sizes. Subplots A1 and B1 of Figure 8 show the condition accuracies when running AutoCodeSherpa on issues from the smallest six and largest six repositories respectively. There was no significant relationship between repository size and accuracy. As the code exploration agent of AutoCodeSherpa retrieves relevant context from the repository, the impact of repository size on accuracy is limited.

AutoCodeSherpa is generally robust to issues of different complexity. The A2 and B2 subplots in Figure 8 show the condition accuracies when running AutoCodeSherpa on issues with small edits of 1-10 lines and large edits of 11+ changed lines. There is little difference in accuracy, indicating that AutoCodeSherpa can find accurate conditions even when the issue involves more

buggy lines. Meanwhile, the A3 and B3 subplots of Figure 8 show the condition accuracies on issues on localized edits with 1 edited function and distributed edits with 2+ edited functions. The change in infection condition accuracy is not statistically significant. Analyzing the difference in PBT accuracy between issues with 1 and 2+ edited functions, we find the output condition is the primary culprit for the decline in accuracy: 75.9% of inaccurate PBTs have inaccurate output conditions. Issues where the developer changes many functions often have complex expected behavior which is not fully specified in the original issue report, leading to a decline in accuracy.

The cost of AutoCodeSherpa does not significantly change by repository or issue complexity. As the C1-C3 subplots in Figure 8 show, there is no statistically significant difference in cost observed in any category.

Answer to RQ5 AutoCodeSherpa scales to repositories of different sizes. AutoCodeSherpa is robust to issues requiring long patches.

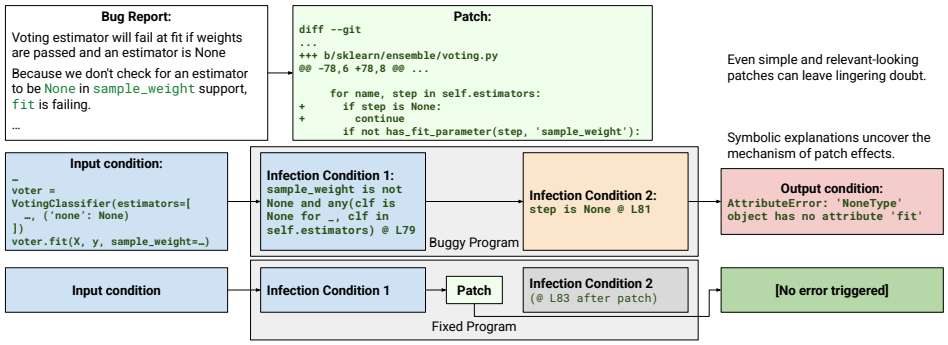


Fig. 9. A schematic of a correct and helpful explanation.

8 Case Studies

To illustrate the symbolic explanations generated by AutoCodeSherpa, we discuss two examples, one helpful and one unhelpful, and analyze them in the context of their corresponding issues.

Helpful Example. Figure 9 presents a schematic of the issue `scikit-learn-13779` from SWE-bench Verified and its symbolic explanation. The issue description, shown at the top-left, reports failure when the `estimator` parameter is `None`. A candidate patch, generated by SpecRover and displayed at the top, appears to handle the parameter correctly. However, in a real-world context, developers would likely need to inspect the code carefully to understand the patch's effects, as LLMs are known to produce plausible but incorrect patches [37].

On the bottom half of Figure 9, we illustrate how the symbolic explanation from AutoCodeSherpa can help understand both the issue and the patch. First, the input condition, shown on the left, accurately indicates that the issue arises when the `estimator` parameter is `None`. Second, infection condition 1, shown in the middle, further helps to understand the issue by indicating the values of key local variables in a buggy execution. Third, infection condition 2 is particularly helpful for understanding the patch by SpecRover: this condition indicates that the bug is triggered when the variable `step` is `None`, which is exactly the case handled by the patch. Finally, the PBT in the explanation can be executed to check automatically that the issue is resolved by the patch.

Metric	gpt-5-mini		deepseek-v3.2	
	Original	Rerun	Original	Rerun
PBT Generation (%)	86.6	87.8	57.8	57.6
PBT Correctness (%)	73.4	76.5	72.2	73.6
Infection Cond. Generated (%)	66.7	66.5	46.0	50.3
Infection Cond. Correctness (%)	79.7	80.2	85.1	85.1
Patch Validation Precision (%)	48.5	48.7	51.5	50.0

Table 7. Comparison of the original and rerun results of AutoCodeSherpa, showing its robustness.

Unhelpful Example. We present an unhelpful symbolic explanation generated for the issue `django-14539`. The issue description mentions that the `urlize` function has a bug. There are in fact two `urlize` functions in the `django` repository, and the bug report does not clarify which. In fact, only one of these two (defined in `html.py`) has real functionality; the second function (defined in `defaultfilters.py`) is simply calling the first. However, both the input condition and infection condition erroneously focus on the `defaultfilters.py` file. As a result, the symbolic explanation does not provide useful information about the root cause of the bug residing in the `html.py` file. This shows that our explanations can sometimes focus on correlated phenomena, rather than conducting an accurate analysis of the causal chain responsible for the bug.

9 Threats to Validity

While experimental results demonstrate the effectiveness of AutoCodeSherpa, we note the limitations of this study. Regarding threats to internal validity, in this study we performed manual analysis to assess the correctness of patches, which may contain mistakes. To mitigate this risk, two authors of the paper independently assessed the correctness of plausible patches, comparing them with the developer patch, and discussed to make a final decision. Meanwhile, there are threats to external validity: the results presented in this work may not generalize to other programming languages or bugs outside of SWE-bench Verified, and the Agentless performance improvement results may not generalize to other issue resolution techniques. Despite this, we note that the concept of bug characterization in the form of symbolic expressions, as well as its instantiation in the form of property-based tests and within-program first-order predicates, is general.

Another threat unique to LLM-based systems is the non-determinism of LLMs. This threat is reduced in three ways. First, we set the sampling temperature to 0 where applicable, which reduces output variance. Second, to empirically assess run-to-run stability, we reran the experiments with GPT-5 mini and DeepSeek-v3.2 and observed results similar to the original runs (Table 7). Finally, the refinements of AutoCodeSherpa filter out low-quality LLM outputs, as shown in Table 6. Beyond this, one could further reduce the variance of generated conditions by aggregating multiple samples through self-consistency [13, 41], which we leave for future work.

10 Related Work

10.1 Issue Reproduction

One line of work closely related to ours is issue reproduction, which aims to write a test case to reproduce a given issue. Since our symbolic explanation is executable, it is akin to a reproducer test. However, unlike existing works on issue reproduction [27, 31, 39], which only try to find example-based reproducing tests, our explanation concisely represents a large, potentially infinite number of buggy program executions with symbolic expressions. In other words, our symbolic explanation is unique in that it *generalizes* from the issue description.

10.2 Automatic Bug Explanation

Prior research shows developers are skeptical of automatically generated patches provided without evidence and that they seek explanations for the root cause of a bug [28, 32]. This observation has prompted researchers to propose techniques that explain bugs. Bugsplainer [30] uses a neural machine translation technique to generate natural language explanations for specific software lines. AutoSD [26] uses LLMs to follow the scientific process to generate and validate hypotheses about the bug. SpecRover [36] analyzes the intended behavior for software components and generates a natural language explanation based on these findings. In contrast to all such approaches, AutoCodeSherpa generates a *symbolic* bug explanation, which is executable and thus for which the veracity can be automatically evaluated.

10.3 Input Conditions

We are unaware of any techniques that jointly generate input, infection, and output conditions, in a similar manner to AutoCodeSherpa. However, there have been efforts to characterize the input space. Avicenna [17] seeks to identify and explain the inputs that cause a fault. However, it can only be applied to inputs conforming to a predefined grammar. This is unlike our approach which uses the expressiveness of PBTs to represent general input and output conditions. As most tasks in our benchmark involve the construction and use of complex objects, we assess that Avicenna would be difficult to compare against in this work. Daikon [18] generates invariants for program states, but requires these invariants to fit pre-defined property templates.

10.4 Coding Agents

Coding agents are autonomous software systems which allow LLMs to invoke tools to perform software engineering (SE) tasks. Early examples include SWE-Agent [44] and AutoCodeRover [45], which focus on resolving software issues. The capability of SE agents has expanded to test generation [31], bug finding [46], and more [5]. To tackle these tasks, SE agents typically make use of program analysis tools such as code search [45], code edit, test execution, and command-line execution [44]. Some agentic systems can employ multiple agents to collaborate on a complex task, with each agent specializing in one part or one step in the task [36]. In the present paper, we have proposed AutoCodeSherpa, a multi-agent system for the task of producing bug explanations.

11 Perspectives

Agents represent a promising new paradigm for the execution of software engineering (SE) activities. Agents use LLMs to invoke various file navigation and program analysis tools to autonomously carry out SE tasks. Despite their capabilities, human oversight and trust-building are still required for the successful deployment of SE agents. In this work, we show the promise of symbolic explanations of issue reports. By producing these symbolic explanations, we can enhance developer understanding, improve the quality of output produced by other agents, and most importantly produce evidence of correctness for patches generated by agents. As SE agents become more commonplace, and coding undergoes further automation, our work can be seen as a mechanism to enhance trust in auto-coding. In this sense, it contributes to the vision of *trusted automatic programming* and its integration into future workflows.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, under NRF Investigatorship Program, Award ID: NRFNRFI11-2026-0001, “Agentic AI based Software of the future: from Scale to Trust”.

Data Availability

Artifact is available: https://osf.io/wfyj4/overview?view_only=6c422aaca07d4c608d33d3e6f8345ee4.

References

[1] 2024. SWE-bench Verified Leaderboard. Retrieved 30 January 2026 from https://www.swebench.com/#verified

[2] 2025. SWT-Bench: Can your model write reproduction tests for real-world issues? Retrieved 30 January 2026 from https://swtbench.com/?results=verified

[3] Abdulaziz Alaboudi and Thomas D LaToza. 2023. What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering* 28, 5 (2023), 117.

[4] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

[5] 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)*.

[6] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–4. doi:10.1109/ICSTW.2015.7107466

[7] AutoCodeRoverSG. 2024. AutoCodeRoverSG/auto-code-rover. Retrieved 30 January 2026 from https://github.com/AutoCodeRoverSG/auto-code-rover

[8] Adrian Bachmann and Abraham Bernstein. 2009. Software process data quality and characteristics: a historical view on open and closed source projects. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. 119–128.

[9] Rishabh Balse, Viraj Kumar, Prajish Prasad, and Jayakrishnan Madathil Warriem. 2023. Evaluating the Quality of LLM-Generated Explanations for Logical Errors in CS1 Student Programs. In *Proceedings of the 16th Annual ACM India Compute Conference* (Hyderabad, India) *(COMPUTE '23)*. Association for Computing Machinery, New York, NY, USA, 49–54. doi:10.1145/3627217.3627233

[10] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. 572–583.

[11] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. (2021). https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3

[12] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[13] Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. 2023. Universal self-consistency for large language model generation. *arXiv preprint arXiv:2311.17311* (2023).

[14] 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 30 January 2026 from https://openai.com/index/introducing-swe-bench-verified/

[15] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

[16] Hang Du, Vijay Krishna Palepu, and James A Jones. 2024. Ripples of a mutation—An empirical study of propagation effects in mutation testing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[17] Martin Eberlein, Marius Smytzek, Dominic Steinhöfel, Lars Grunske, and Andreas Zeller. 2023. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 438–449.

[18] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*. 213–224.

[19] GitHub. 2025. GitHub Copilot: Meet the new coding agent. Retrieved 30 January 2026 from https://github.blog/news-insights/product-news/github-copilot-meet-the-new-coding-agent/

[20] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[21] Paul Holser. 2020. Junit-quickcheck: Property-based testing, junit-style. Retrieved 30 January 2026 from https://pholser.github.io/junit-quickcheck/site/1.0/

[22] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 135–145. doi:10.1109/ICST.2016.37

[23] 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

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

[25] René Just, Michael D Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 international symposium on software testing and analysis*. 315–326.

[26] 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.

[27] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2312–2323. doi:10.1109/ICSE48619.2023.00194

[28] 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

[29] David Maciver and Zac Hatfield-Dodds. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4 (11 2019), 1891. doi:10.21105/joss.01891

[30] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 640–652. doi:10.1109/ICSE48619.2023.00063

[31] 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

[32] 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

[33] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 international symposium on software testing and analysis*. 24–36.

[34] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating Agent-Based Program Repair at Google. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 365–376. doi:10.1109/ICSE-SEIP66354.2025.00038

[35] Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. 2026. Agentic ai software engineer: Programming with trust. *Commun. ACM* 69, 5 (2026).

[36] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 617–617. doi:10.1109/ICSE55347.2025.00080

[37] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence?. In *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group (PPIG 2022)*.

[38] GitHub User. 2025. Comment on PR #115733. Retrieved 30 Jan 2026 from https://github.com/dotnet/runtime/pull/115733#issuecomment-2892088377

[39] Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2024. AEGIS: An Agent-based Framework for General Bug Reproduction from Issue Descriptions. *arXiv preprint arXiv:2411.18015* (2024).

[40] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[41] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).

[42] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are" Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *arXiv preprint arXiv:2503.15223* (2025).

[43] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).

[44] 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. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[45] 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

[46] Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. 2025. An LLM Agent for Functional Bug Detection in Network Protocols. *arXiv preprint arXiv:2506.00714* (2025).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009