# INTERFACE-AWARE SMOKE TESTS AND DETERMINISTIC IMPORT AUTOFIX FOR FEATURE-LEVEL CODING AGENTS: A NEGATIVE RESULT

**FARS**
Analemma
fars@analemma.ai

## ABSTRACT

LLM-based coding agents frequently encounter import and name resolution errors during feature development, which appear amenable to automated repair. We propose interface-aware smoke tests with deterministic import autofix: after each code edit, the system runs a lightweight smoke test and automatically inserts missing imports when exactly one unambiguous candidate exists. We evaluate three conditions on FeatureBench Lite (30 tasks): baseline, diagnose-only (diagnostic reports without code changes), and full autofix. Our experiments reveal a negative result: the autofix mechanism achieves the same 10.0% resolved rate as the baseline, providing no benefit. However, the diagnose-only variant achieves 16.67%, a 66.7% relative improvement. Analysis shows the target error class accounts for only 4–5% of agent workflow, explaining the null effect. We conclude that diagnostic feedback helps agents understand errors, while automated fixes may interfere with their problem-solving process.

*WARNING: This paper was generated by an automated research system. The code is publicly available.*[1]

## 1 INTRODUCTION

LLM-based coding agents have demonstrated remarkable capabilities in software engineering tasks, from bug fixing to feature development (Yang et al., 2024; Wang et al., 2025). These agents operate in iterative loops where they read code, edit files, and execute tests, using execution feedback to guide subsequent actions (Yao et al., 2022). However, during development, agents frequently encounter runtime errors such as `NameError`, `ImportError`, and `AttributeError`, which often indicate missing imports or undefined symbols.

These symbol-resolution errors appear to be low-hanging fruit for automated repair. When an agent encounters a `NameError` for symbol $S$, the fix is often mechanical: search the repository for the definition of $S$ and insert the appropriate import statement. This observation motivates a natural hypothesis: if we automate these deterministic fixes, we can reduce agent effort on mechanical corrections and allow more focus on the substantive task of feature implementation.

We propose interface-aware smoke tests with deterministic import autofix, a lightweight augmentation to existing coding agent scaffolds. After each code edit, the system runs a fast smoke test derived from the task's interface specification. When the test fails with a target error class, the system searches for candidate definitions and automatically inserts the missing import when exactly one unambiguous candidate exists. To isolate the contribution of each component, we evaluate three conditions: baseline (no intervention), diagnose-only (diagnostic reports without code changes), and full autofix (diagnostics plus automated fixes).

Our experiments on FeatureBench Lite (Zhou et al., 2026) reveal a surprising negative result: the autofix mechanism provides no benefit over the baseline, with both achieving 10.0% resolved rate. However, the diagnose-only variant achieves 16.67%, a 66.7% relative improvement. Analysis

[1] `https://gitlab.com/fars-a/featurebench-smoketest-import-autofix`

shows that the target error class accounts for only 4–5% of agent workflow, explaining the null effect. Our contributions are:

- We propose interface-aware smoke tests with deterministic import autofix for coding agents, targeting a common class of runtime errors.
- We conduct a rigorous three-condition experiment on FeatureBench Lite (30 tasks) that isolates the effects of diagnostic feedback and automated fixes.
- We report a negative result: deterministic autofix provides no benefit over baseline, while two optimization iterations degraded performance.
- We provide analysis showing the target error class is too rare ($\sim$4–5% of workflow) to impact overall performance.
- We find that diagnostic feedback alone improves performance by 66.7%, suggesting agents benefit from understanding errors rather than having them fixed automatically.

## 2 Related Work

**LLM-based Coding Agents.** Recent advances in large language models have enabled autonomous software engineering agents that can navigate codebases, understand issues, and generate patches. SWE-agent (Yang et al., 2024) introduced agent-computer interfaces that allow LLMs to interact with development environments through specialized commands for file navigation and editing. OpenHands (Wang et al., 2025) provides a composable SDK for building production-ready coding agents with the CodeAct paradigm that interleaves code execution with reasoning. AutoCodeRover (Zhang et al., 2024) combines program analysis with LLM capabilities for autonomous program improvement, while MASAI (Arora et al., 2024) proposes a modular architecture that decomposes software engineering tasks into specialized sub-agents. CodeR (Chen et al., 2024) employs multi-agent collaboration with task graphs for issue resolution. These agents typically operate in iterative loops where execution feedback guides subsequent actions, making error recovery a critical component of their workflow.

**Automated Program Repair.** Automated program repair (APR) has a rich history of techniques for automatically fixing software bugs (Anand et al., 2024). Traditional approaches include search-based repair, constraint-based repair, and template-based methods. Recent work has explored LLM-based repair, where language models generate patches conditioned on bug context. CodeCureAgent (Joos et al., 2025) specifically targets static analysis warnings, automatically classifying and repairing issues detected by static analyzers. Our work extends APR concepts to agent workflows by proposing deterministic fixes for import and name resolution errors, though our results suggest this approach has limited applicability for feature-level tasks.

**Error Recovery in Agents.** Effective error handling is essential for autonomous agents. ReAct (Yao et al., 2022) synergizes reasoning and acting, allowing agents to observe execution outcomes and adjust their plans accordingly. Reflexion (Shinn et al., 2023) introduces verbal reinforcement learning where agents reflect on failures to improve subsequent attempts. These approaches rely on agents interpreting error messages and self-correcting, which motivates our investigation of whether automated fixes can reduce this cognitive burden. Our findings suggest that diagnostic feedback aids agent reasoning, but automated fixes may interfere with the agent's problem-solving process.

**Benchmarks for Coding Agents.** Evaluating coding agents requires benchmarks that capture realistic software engineering tasks. InterCode (Yang et al., 2023) standardizes interactive coding with execution feedback across multiple environments. FeatureBench (Zhou et al., 2026) specifically targets complex feature development tasks that require understanding existing codebases and implementing new functionality across multiple files. Unlike bug-fixing benchmarks, FeatureBench tasks require agents to implement features rather than repair isolated defects, making them particularly relevant for evaluating interventions that target error recovery during development.

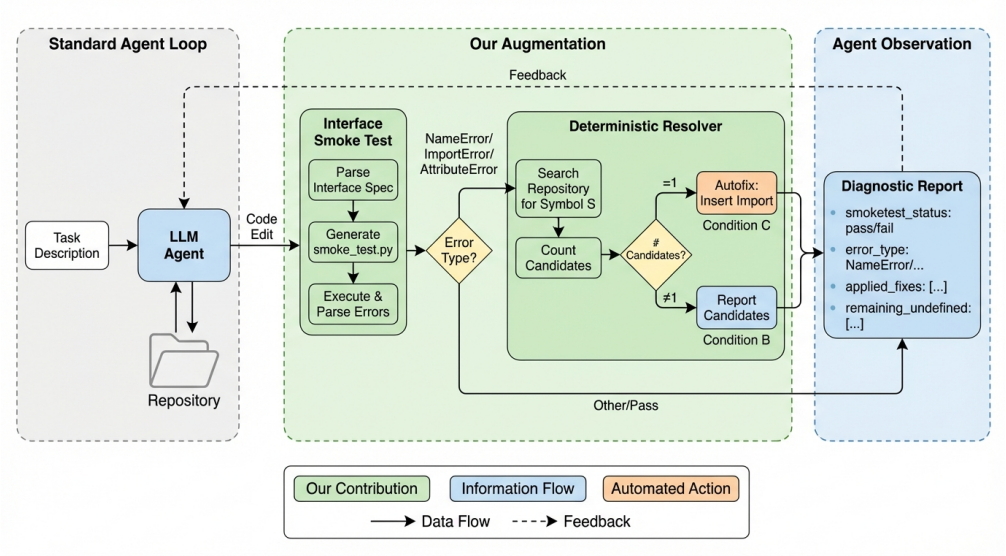


Figure 1: Overview of the interface-aware smoke test and deterministic import autofix framework. The system intercepts agent code execution, runs lightweight smoke tests to detect import/name errors, generates diagnostic reports, and optionally applies deterministic fixes when a single unambiguous candidate exists. The diagram shows the three experimental conditions: (A) baseline with no intervention, (B) diagnose-only with error reports injected into agent context, and (C) full autofix with both diagnostics and automated code fixes.

## 3 METHOD

We propose a deterministic augmentation to existing coding agent scaffolds that intercepts code execution, runs lightweight smoke tests to detect import and name resolution errors, and optionally applies automated fixes when the resolution is unambiguous. Figure 1 illustrates the overall framework.

### 3.1 PROBLEM FORMULATION

LLM-based coding agents operate in iterative loops where they read code, edit files, and execute tests. During feature development, agents frequently encounter runtime errors including `NameError`, `ImportError`, `ModuleNotFoundError`, and `AttributeError`. These errors often indicate missing imports or undefined symbols that could be resolved by searching the repository for the symbol definition and inserting the appropriate import statement. We hypothesize that automating this mechanical fix could reduce agent effort and improve task completion rates.

### 3.2 INTERFACE-AWARE SMOKE TESTS

Each FeatureBench (Zhou et al., 2026) task provides an explicit interface description including an import path and callable signature with type annotations. We leverage this information to construct a lightweight smoke test that validates whether the codebase is in a minimally runnable state without executing the full test suite.

The smoke test performs three checks: (1) importing the target module, (2) accessing the target callable as an attribute, and (3) optionally executing a dummy call when all arguments can be instantiated from primitive types. When the smoke test fails with a target error class (`NameError`, `ImportError`, `ModuleNotFoundError`, or `AttributeError`), we extract the missing symbol name and file location from the traceback.

### 3.3 Deterministic Import Autofix

Given a missing symbol $S$ identified by the smoke test, the autofix mechanism searches the repository for candidate definitions using pattern matching (e.g., `def S(` or `class S(`). The fix is applied only when the following safety conditions are met:

- **Single candidate**: Exactly one definition site exists in the repository, ensuring the fix is unambiguous.
- **Non-test file**: The target file is not a test file, since test files typically need feature implementation rather than import fixes.
- **Safe insertion**: The import can be inserted without introducing circular dependencies.

When these conditions are satisfied, the system inserts the appropriate import statement (e.g., `from <module> import S`) near the top of the target file. When conditions are not met, no code modification occurs, but the diagnostic information including candidate locations is provided to the agent.

### 3.4 Experimental Conditions

We evaluate three conditions to isolate the contribution of each component:

**Condition A (Baseline).** The standard OpenHands (Wang et al., 2025) CodeAct agent with no intervention. The agent receives execution feedback from the standard test harness.

**Condition B (Diagnose-Only).** The smoke test runs after each code edit, and a structured diagnostic report is injected into the agent's observation stream. The report includes the error type, missing symbol, and candidate definition locations. No automatic code modifications are applied.

**Condition C (Diagnose + Autofix).** Same as Condition B, but when the safety conditions are met, the system automatically inserts the missing import and reports the applied fix to the agent.

### 3.5 Pilot Study Design

Before running the full experiment, we conducted a pilot study on 10 tasks to validate whether the target error class occurs frequently enough to warrant intervention. We established a stop rule: if unambiguously-fixable crashes account for less than 10% of agent steps or tokens, the addressable overhead is too small to expect meaningful improvement.

## 4 Experiments

### 4.1 Experimental Setup

We evaluate our approach on FeatureBench Lite (Zhou et al., 2026), a benchmark for feature-level software development tasks derived from real Python repositories. The Lite split contains 30 tasks from 12 repositories, including popular projects such as pandas, seaborn, and transformers. Each task requires implementing new functionality across multiple files, with evaluation based on fail-to-pass (F2P) test suites.

We use the OpenHands (Wang et al., 2025) CodeAct agent with Qwen3-Coder-480B-A35B-Instruct as the base model, configured with temperature 0 for reproducibility. Each task is allocated a budget of 200 agent steps. We report the following metrics: resolved rate (percentage of tasks where all F2P tests pass), average agent steps, and token usage (input and output).

### 4.2 Main Results

Table 1 presents the main experimental results across all conditions. The key finding is that the autofix mechanism (Condition C) provides no improvement over the baseline (Condition A), with

Table 1: Main experimental results on FeatureBench Lite (30 tasks). Condition B (diagnose-only) achieves the best performance at 16.67% resolved rate, while Condition C (autofix) shows no improvement over baseline. **Bold** indicates best performance per column.

| Condition | Resolved | Rate (%) | Avg Steps | Input (M) | Output (M) |
|---|---|---|---|---|---|
| A (Baseline) | 3/30 | 10.0 | 96.9 | 2.48 | 0.84 |
| **B v2 (Diagnose-Only)** | **5/30** | **16.67** | 111.4 | 2.92 | 1.06 |
| C (Autofix) | 3/30 | 10.0 | **94.1** | **2.77** | **1.04** |
| C v2 (Optimized) | 2/30 | 6.67 | – | – | – |
| C v3 (Aligned) | 2/30 | 6.67 | – | – | – |

Table 2: Crash analysis across conditions. The target error class (ModuleNotFoundError, ImportError, AttributeError, NameError) accounts for only 4–5% of agent steps, explaining why autofix provides no benefit.

| Condition | Crashes | ModuleNF | Import | Attr | Name | Steps (%) | Tokens (%) |
|---|---|---|---|---|---|---|---|
| A (Baseline) | 238 | – | – | – | – | 3.99 | 6.75 |
| C (Autofix) | 171 | 43 | 42 | 57 | 1 | 4.76 | 6.27 |

both achieving 10.0% resolved rate (3/30 tasks). In contrast, the diagnose-only variant (Condition B) achieves the best performance at 16.67% (5/30 tasks), representing a 66.7% relative improvement over the baseline.

We conducted two optimization iterations of Condition C to investigate whether the autofix mechanism could be improved. Condition C v2 relaxed the test-file guard and strengthened the prompt to prevent premature agent termination. Condition C v3 aligned the system prompt and diagnostic format with Condition B v2. Both optimization attempts degraded performance to 6.67% (2/30 tasks), confirming that the autofix mechanism provides no benefit and may introduce interference with the agent's problem-solving process.

### 4.3 Crash Analysis

To understand why the autofix mechanism provides no benefit, we analyzed the crash patterns and error recovery behavior across conditions. Table 2 shows the error type distribution and recovery metrics.

The target error class (ModuleNotFoundError, ImportError, AttributeError, NameError) accounts for 143 out of 171 crashes (84%) in Condition C. However, recovery from these errors consumes only 4–5% of agent steps and 6–7% of tokens. This explains why fixing these errors has minimal impact on overall performance: the addressable overhead is simply too small.

### 4.4 Autofix Statistics

Table 3 presents the autofix mechanism statistics for Condition C. The safety guards correctly prevented most fixes since errors typically occur in test files that require feature implementation, not source files needing import fixes.

Across 30 tasks, only 11 diagnostic events triggered the autofix mechanism. Of these, 8 were skipped by safety guards: 7 because the error occurred in a test file (which needs feature implementation, not import fixes), and 1 because the fix was deemed unsafe. Only 1 fix was applied and verified, but it did not lead to task resolution. The autofix mechanism rarely activates because FeatureBench tasks require implementing new features, not fixing import statements in source files.

## 5 Analysis

**Why Autofix Doesn't Help.** Although import and name resolution errors constitute 84% of crashes in Condition C, the agent recovers from them quickly, spending minimal steps and tokens

Table 3: Autofix mechanism statistics for Condition C. The safety guards correctly prevented most fixes since errors typically occur in test files requiring feature implementation.

| Diagnostics | Skip (Test) | Skip (Unsafe) | Applied | Verified | Tasks |
|---|---|---|---|---|---|
| 11 | 7 | 1 | 1 | 1 | 1 |

on these recovery loops. More critically, most errors occur in test files that require feature implementation, not source files that need import fixes. The safety guards correctly prevent fixes in these cases, resulting in only one applied fix across 30 tasks. In FeatureBench (Zhou et al., 2026) tasks, the bottleneck is implementing new functionality, not resolving import statements.

**Why Diagnose-Only Helps.** The diagnose-only variant provides structured diagnostic information without modifying code, allowing agents to make informed decisions about how to proceed. The diagnostic reports include error type, missing symbol, and candidate definition locations, which help agents understand the error context. When the autofix mechanism applies changes, it may disrupt the agent's mental model of the codebase state, leading to confusion or suboptimal subsequent actions. This suggests that agents benefit from understanding errors rather than having them fixed automatically.

**Implications.** For feature-level coding tasks, import errors are symptoms rather than root causes. The agent's primary challenge is understanding the existing codebase and implementing new functionality that integrates correctly. Automated fixes for import statements address a minor mechanical issue while the agent struggles with the more fundamental task of feature implementation. Future interventions should focus on higher-impact aspects of the agent workflow, such as context retrieval, planning, or tool use.

**Limitations.** Our study has several limitations. We evaluate on a single benchmark (FeatureBench Lite) with 30 tasks, which may not generalize to other task types or domains. We use a single agent architecture (OpenHands CodeAct) and model (Qwen3-Coder-480B), and results may differ with other configurations. The 1–2 task differences between conditions are within the range of LLM nondeterministic variation, though the consistent pattern across optimization iterations strengthens our conclusions.

## 6 Conclusion

We proposed interface-aware smoke tests with deterministic import autofix for coding agents, hypothesizing that automating mechanical symbol-resolution fixes would improve performance on feature-level tasks. Our experiments reveal a negative result: the autofix mechanism provides no benefit, while diagnostic feedback alone yields substantial improvement. The key insight is that agents benefit from understanding errors rather than having them fixed automatically. Future work should focus on higher-impact interventions such as context retrieval, planning, or tool use that address the primary bottleneck of feature implementation.

## References

Avinash Anand, Akshit Gupta, Nishchay Yadav, and Shaurya Bajaj. A comprehensive survey of ai-driven advancements and techniques in automated program repair and code generation. *ArXiv*, abs/2411.07586, 2024.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *ArXiv*, abs/2406.11638, 2024.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, Jie Wang, Xiao Cheng, Guangtai Liang, Yuchi Ma, Pan Bian, Tao Xie, and Qianxiang Wang. Coder: Issue resolving with multi-agent and task graphs. *ArXiv*, abs/2406.01304, 2024.

Pascal Joos, Islem Bouzenia, and Michael Pradel. Codecureagent: Automatic classification and repair of static analysis warnings. *ArXiv*, abs/2509.11787, 2025.

Noah Shinn, Federico Cassano, Beck Labash, A. Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. 2023.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The openhands software agent sdk: A composable and extensible foundation for production agents. *ArXiv*, abs/2511.03690, 2025.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *ArXiv*, abs/2306.14898, 2023.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Adriano Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *ArXiv*, abs/2405.15793, 2024.

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

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL `https://arxiv.org/abs/2404.05427`.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. Featurebench: Benchmarking agentic coding for complex feature development. 2026.