Debug2Fix: Supercharging Coding Agents with Interactive Debugging Capabilities

Spandan Garg*
spgarg@microsoft.com
Microsoft
USA

Yufan Huang
yufanhuang@microsoft.com
Microsoft
USA

ABSTRACT

While significant progress has been made in automating various aspects of software development through coding agents, there is still significant room for improvement in their bug fixing capabilities. Debugging and investigation of runtime behavior remains largely a manual, developer-driven process. Popular coding agents typically rely on either static analysis of the code or iterative test-fix cycles, which is akin to trial and error debugging. We believe that there is a wealth of rich runtime information that developers routinely access while debugging code, which agents are currently deprived of due to design limitations. Despite how prevalent debuggers are in modern IDEs and command-line tools, they have surprisingly not made their way into coding agents. In this work, we introduce Debug2Fix, a novel framework that incorporates interactive debugging as a core component of a software engineering agent via a subagent architecture. We incorporate debuggers for Java and Python into our agent framework and evaluate against GitBug-Java and SWE-Bench-Live and achieve >20% improvement in performance compared to the baseline for certain models. Furthermore, using our framework, we're able to make weaker models like GPT-5 and Claude Haiku 4.5 match or exceed the performances of stronger models like Claude Sonnet 4.5, showing that better tool design is often just as important as switching to a more expensive model. Finally, we conduct systematic ablations demonstrating the importance of both the subagent architecture and debugger integration.

ACM Reference Format:
Spandan Garg and Yufan Huang. 2026. Debug2Fix: Supercharging Coding Agents with Interactive Debugging Capabilities. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Corresponding author.

1 INTRODUCTION

Coding agents [1–3] demonstrate impressive capabilities in a variety of software engineering tasks. Bug-fixing is among the most common tasks developers perform with agents [4], yet there remains significant room for improvement [5–7]. Taking a closer look at how agents approach bugs sheds some light on some of the limitations in their bug-fixing capabilities. When faced with bugs or failing tests, coding agents follow one of two strategies: making changes based on pure static code analysis, or entering extended print-debugging sessions, both of which requires the agent to guess the underlying state of the program during execution. In the latter, the agent cycles between examining the program output i.e. reading error messages, stack traces, console logs, etc. to guess the underlying behavior and then making changes based on its interpretation of the output. If the error persists, the agent continues to iterate through the debug and fix steps until the error goes away or it gives up. While this approach works for simpler bugs, the cycle of print-debugging and fixing is not only slow because it depends on repeatedly executing the program with small incremental changes that slowly approach the correct fix, but also unreliable as the agent is guessing runtime behavior rather than observing it directly. Without having access to the actual program state, the agent may make its decisions based on faulty hypotheses and make incorrect changes, which can even seep through human-written test suites [8].

We argue that this kind of print-debugging that agents primarily rely on, forms a major inefficiency in the debugging capabilities of today's agents. In contrast to this, expert human developers have long relied on debuggers to diagnose bugs effectively. Debuggers allow developers to pause the program at user-specified locations known as breakpoints, and inspect the values of variables, evaluate arbitrary expressions of code, examine the ongoing call-stack and step through code line by line. This is significantly more precise than inferring program behavior based on simply reading code or print output, where one wrong guess can compound and derail a whole debug session. Furthermore, this is also more efficient because converging on the right set of breakpoints requires fewer iterations than incrementally building an understanding of code based on print outputs alone.

Despite the prevalence of debuggers in modern IDEs and command-line (CLI) tools, they have surprisingly not made their way into coding agents. This is likely due to the fact that debuggers were designed with human-interaction in mind. They require careful orchestration of commands to interact with the program state. Command-line debugger commands are verbose by design as they're designed to provide as much diagnostic information as possible. Finally, debuggers involve asynchronous events and timing, which is not easy for an agent to manage programmatically. As a consequence, naively exposing a debugger to the agent can lead to brittle interactions and failures. Due to these limitations, an effective schema for debuggers is a non-trivial task. However, it's also possible that one might go through the exercise of creating a perfect suite of tools only for the agent to never use it [9]. In our experiments, providing debug tools directly to the main agent resulted in minimal usage. Despite all these difficulties, we believe that agents not leveraging debuggers represents a significant missed opportunity.

To address all these gaps we introduce **Debug2Fix**, the first specialized agent framework that incorporates debugging as a core component of the agent framework. It works by incorporating debuggers into coding agents via a novel tool design and crucially, a subagent architecture. Rather than exposing the debugger tools to the main agent directly, where they most likely go unused, we expose a unified Debug Subagent to the main agent, which is strongly instructed to use it. This subagent design encapsulates all the complexities associated with debuggers behind a simple high-level interface for the main agent to use. During inference, the agent offloads debugging tasks to the subagent, which then handles the debugger orchestration via its set of tools and returns a concise answer with all the findings. Our contributions in this work are as follows:

- **Debug2Fix**: We present the first integration of interactive debugging into a coding agent via the use of a subagent and a novel tool schema, enabling complex runtime debugging capabilities that complement static code analysis. Through our qualitative analysis, we show that the Debug Subagent follows similar workflows as an expert human developer.
- **Empirical Evaluation & Ablation Study**: We conduct an extensive evaluation of our approach by incorporating the Java Debugger (JDB) and Python Debugger (PDB) into a coding agent and evaluate on bug fixing benchmarks like GitBug-Java [10] and SWE-Bench-Live [11]. We show that adopting Debug2Fix framework improves performance over vanilla agent by >20% relative to baseline numbers in some cases. We also conduct ablation experiments showing that both our tool design and subagent architecture are essential.

2 BACKGROUND AND RELATED WORK

We build upon a rich foundation of research in Software Engineering and Agentic AI. Our work bridges a crucial gap in Software Engineering Agents and the typical developer workflow for fixing bugs.

2.1 Automated Debugging

Debugging is a core skill in software development. However it remains largely a manual and human-driven process. There have been works that try to automate debugging. AutoSD [12] prompts LLMs to automatically generate hypotheses and uses debuggers to interact with buggy code. ChatDBG [13] augments traditional debuggers into an AI-powered debugging assistant. While still a human-driven process, it integrates LLMs into debuggers to enhance the user-friendliness of conventional debuggers and allowing humans to have a dialogue with the debugger and pose complex questions. Zhong et al. [14] propose an LLM Debugger (LDB) that allows LLMs to refine their own generated programs with runtime information. It segments programs into blocks and tracks intermediate values after each block.

While these approaches demonstrate the value of runtime information and various forms of debugging, they differ from our work in key ways. Our work is specifically designed to introduce debuggers into automated coding agents that work on the entire repository. It encapsulates debugger complexity behind a high-level interface so that agents can utilize it effectively.

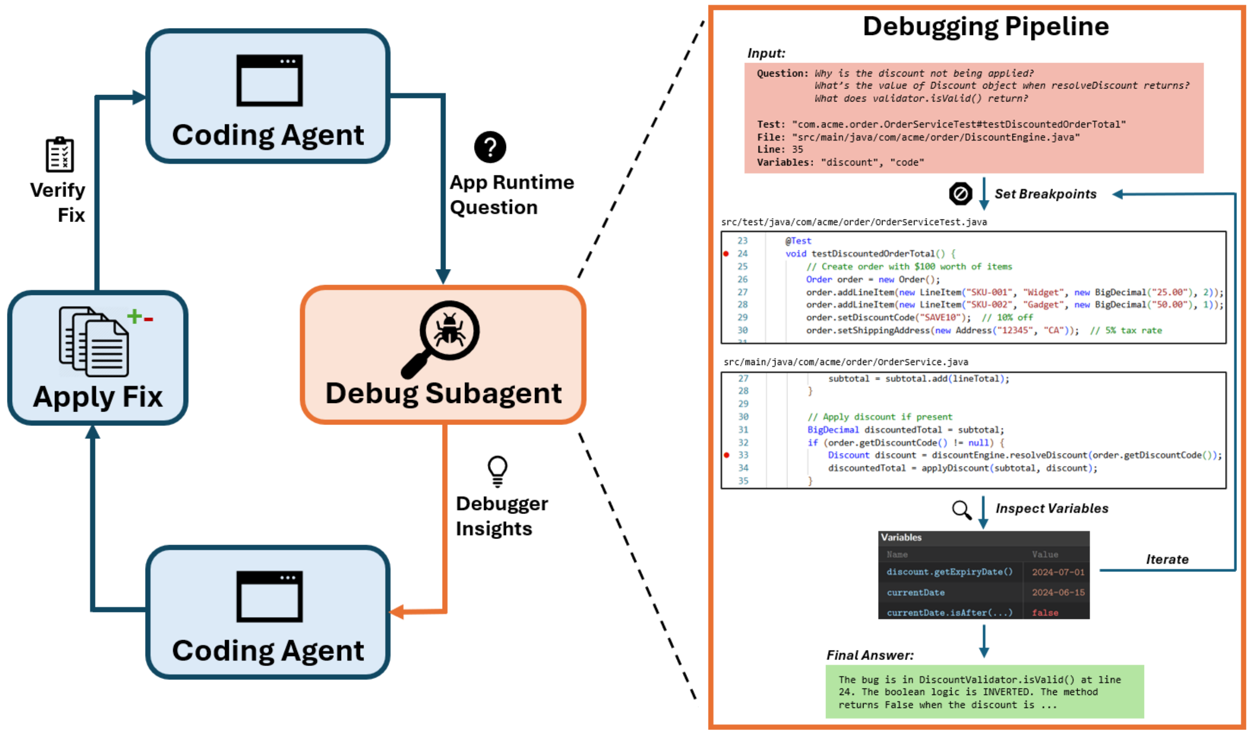
2.2 Software Engineering Agents

There have been significant advancements in the field of SE Agents. Starting with SWE-Agent [15], which proposed key design principles that demonstrated high-performance on repository-level tasks. Since then agent systems like OpenHands [3], Claude Code [1], Copilot CLI [16], VSCode Agent [2], Windsurf [17] have seen widespread adoption into the developer workflow and demonstrate impressive performance on various software engineering tasks such as test generation, bug fixing, code search, etc.

While existing agent frameworks have shown impressive capabilities on various benchmarks, gaps still remain when it comes to harder bug-fixing tasks. These agents leverage suboptimal techniques like iterative print-debugging cycles by inserting logging statements into code or making educated guesses based on static analysis of code. Our work addresses these core limitations with agents by incorporating interactive debugging into coding agents as a dedicated subagent.

2.3 Multi-Agent Architectures

The use of specialized agents has proven to be an effective pattern in AI Agent Systems. Rather than expecting a single agent / model to have all capabilities, multi-agent systems decompose the high-level problem and delegate responsibilities to smaller agents or subagents. Many works have improved problem-solving abilities of LLMs by integrating discussions among multiple agents. In their survey [18], He et al. systematically reviewed the landscape of LLM-based multi-agent systems for software engineering highlighting the current capabilities and limitations of these approaches. MASAI [19] proposes a modular architecture for software engineering agents where subagents are instantiated with well-defined objectives



```
Question: Why is the discount not being applied?
          What's the value of Discount object when resolveDiscount returns?
          What does validator.isValid() return?

Test: "com.acme.order.OrderServiceTest#testDiscountedOrderTotal"
File: "src/main/java/com/acme/order/DiscountEngine.java"
Line: 35
Variables: "discount", "code"
```

src/test/java/com/acme/order/OrderServiceTest.java

```java
    @Test
    void testDiscountedOrderTotal() {
        // Create order with $100 worth of items
        Order order = new Order();
        order.addLineItem(new LineItem("SKU-001", "Widget", new BigDecimal("25.00"), 2));
        order.addLineItem(new LineItem("SKU-002", "Gadget", new BigDecimal("50.00"), 1));
        order.setDiscountCode("SAVE10");  // 10% off
        order.setShippingAddress(new Address("12345", "CA"));  // 5% tax rate
```

src/main/java/com/acme/order/OrderService.java

```java
            subtotal = subtotal.add(lineTotal);
        }

        // Apply discount if present
        BigDecimal discountedTotal = subtotal;
        if (order.getDiscountCode() != null) {
            Discount discount = discountEngine.resolveDiscount(order.getDiscountCode());
            discountedTotal = applyDiscount(subtotal, discount);
        }
```

Variables

Name	Value
discount.getExpiryDate()	2024-07-01
currentDate	2024-06-15
currentDate.isAfter(...)	false

```
The bug is in DiscountValidator.isValid() at line
24. The boolean logic is INVERTED. The method
returns False when the discount is ...
```

Figure 1: A high-level view of the overall Debug2Fix pipeline with the Debug Subagent. We can see that the main agent's is to loop between querying the Debug Subagent, followed by making fixes based on the learned insights from runtime behavior. Internally, the Debug Subagent works by going through a cycle of setting breakpoints, stepping through code and inspecting variables / expressions until it has the answer to main agent's query or runs out of turns.

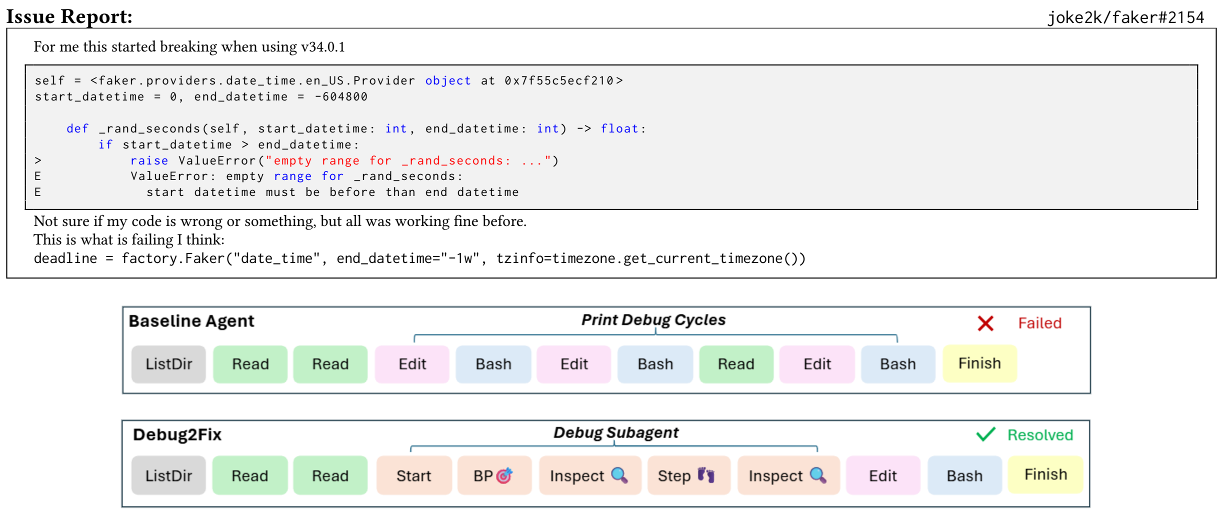
and strategies, achieving competitive performance on coding benchmarks. AutoDev [20] uses multiple autonomous AI Agents to achieve user defined objectives. MapCoder [21] is another such framework that consists of several LLM agents, which is designed to simulate the stages of the developer cycle.

Our Debug Subagent follows a modular multi-agent pattern, providing the main agent with a high-level interface for debugging queries while encapsulating the complexity of low-level debugger interactions. Unlike prior works that decompose tasks into code search, localization, repair phases, we introduce a specialized capability that complements static analysis. Through ablation experiments, we show that this architectural choice is essential i.e. when debugging tools are exposed to the main agent directly, they go largely unused. The subagent design is pivotal in making this otherwise low-level capability into a simple interface the main agent readily utilizes.

3 MOTIVATING EXAMPLE

In this section, we provide an overview of Debug2Fix with a motivating example. Figure 2 shows an example of a bug from a popular open-source python library where the user is trying to generate a random Unix timestamp from up to a week ago (`-1w` in code). The library method works by parsing the `end_datetime` string into a timestamp by computing a time delta of negative seven days compared to the output of the `_safe_now()` method, which is supposed to return the current timestamp. However, buried deep in the class hierarchy, its fallback is set to `datetime(1970, 1, 1)` in one of the code paths. Since this is 'Unix Time `0`', taking a negative timestamp from it results in a crash.

We compare the trajectories of both the baseline agent and our Debug2Fix agent when tasked with solving this problem given the corresponding repo. Looking at the high-level agent trajectories in the figure, we can see that the baseline agent engages in repeated edit and bash calls to do print-debugging, until it believes issue is fixed. Due to how deep in the codebase the issue is, the agent ends up having to do a lot of print debugging as it struggles to find where the bug is coming from. In this case, it actually ends up re-writing the unit test associated with this problem and submitting the wrong fix. Debug2Fix solves this problem with relative ease. Rather than guessing at the program's runtime behavior through print statements, Debug2Fix delegates the problem of finding the root cause to the Debug Subagent as seen in the figure. The subagent sets a breakpoint directly in the test script and inspects the local variables, while stepping through the code. Looking at the return value of `_safe_now()`, which returns `datetime(1970,`


Issue Report: joke2k/faker#2154

For me this started breaking when using v34.0.1

```
self = <faker.providers.date_time.en_US.Provider object at 0x7f55c5ecf210>
start_datetime = 0, end_datetime = -604800

    def _rand_seconds(self, start_datetime: int, end_datetime: int) -> float:
        if start_datetime > end_datetime:
>           raise ValueError("empty range for _rand_seconds: ...")
E           ValueError: empty range for _rand_seconds:
E             start datetime must be before than end datetime
```

Not sure if my code is wrong or something, but all was working fine before.
This is what is failing I think:
`deadline = factory.Faker("date_time", end_datetime="-1w", tzinfo=timezone.get_current_timezone())`

Figure 2: A bug from a popular open-source Python repository on GitHub. We see very different trajectories taken by the Baseline Agent and Debug2Fix. In the baseline, we see the agent doing repeated print-debug cycles and arriving at the wrong fix due to not being able to find the root cause of the issue, which is situated deep within the repo. With Debug2Fix, the agent uses the Debug Subagent which is able to find the root cause immediately using a debugger. This results in the agent arriving at the correct fix.

1, 1) instead of the current time immediately reveals the root cause of the issue.

This set of contrasting trajectories was one of many observed across our benchmark evaluations.

4 METHODOLOGY

In this section, we explain the approach behind Debug2Fix in detail and how we incorporate it into an existing coding agent. Figure 1 shows a high-level view of the overall approach. We first describe the schema and underlying architecture of the Debug Subagent. Let's begin by describing what a subagent is:

Subagent: A subagent is a secondary LLM-based agent that is invoked by the primary (or "main") agent to handle a specialized sub-task. Unlike the main agent, which is meant to solve a broad set of tasks, the subagent solves a smaller, often simpler, set of tasks using a specialized list of tools. Just like the main agent, the subagent has its own system prompt, context window and tool-set that it uses to achieve the goal delegated to it by the main agent.

4.1 Debug Subagent

The Debug Subagent is one such specialized subagent designed to answer questions about runtime behavior. In our case, it's exposed to the main agent as a simple tool, which can be invoked with the following set of parameters and arguments:

- **Runtime Question** (Required): The main agent describes what it wants to know about the program's runtime behavior. This is the only required field.
- **Test(s)** (Required): The main agent provides a script / test that fails.
- **Path** (Optional): This is a path to the code the main agent wants the subagent to exercise.
- **Lines** (Optional): These are the lines we want to set the initial breakpoints on.
- **Variable** (Optional): These are the variables the main agent has told us to pay particular attention to.

As it's output, the Debug Subagent returns a structured response containing a final answer to the question asked by the main agent and supporting evidence (observed variable values, stack traces, source locations, etc.).

This simple question-answer abstraction shields the main agent from the underlying complexity of the debugging orchestration. Rather than exposing low-level debugger commands to the main agent, this interface deliberately gives it the flexibility to ask questions in natural language while providing helpful metadata and the subagent to adapt accordingly.

4.2 Debug Tools

The Debug Subagent interacts with a set of tools that encapsulate the underlying debugger used for that language. These tools are designed to mirror the core actions a human developer would perform with a debugger, but in an LLM-friendly way. Below we describe each tool one by one.

Debug Start Session: This is the most critical tool that we add and gets called at the start of every Debug Subagent trajectory and performs the entire setup sequence atomically. It detects the underlying build system, builds the project, launches the test, waiting for a debug port to be available, attaching a debugger to it and setting any initial breakpoints. By default the tool also sets a breakpoint at the start of the test method provided by the main agent. Building all these steps into a single tool was a deliberate design choice. In our early experiments, exposing each step as a separate tool resulted in frequent failures, race conditions and timeouts. The atomic design is key to systematically eliminating all these issues.

Debug Control: Once the debug session is active and paused at a breakpoint, the subagent can use this tool to control the flow of execution. A debugger typically allows a user to take one of the following actions after a breakpoint is hit: continue, step over, step into or step out. These allow the user to let the execution continue to the next breakpoint, step over to the next line after executing the current one, step into the method being called on the current line or finally, exit the method to the calling method's context. Rather than one tool for each of these actions, we provide one central control tool that takes an enum for which action the LLM wants the subagent to take. Finally, the text output of this tool shows the surrounding context and the lines that have breakpoints.

Debug Inspect: Once the execution is paused, this tool allows the subagent to query program state. It supports examining local variables, evaluating arbitrary expressions, viewing call stacks and inspecting fields within objects. This is where the subagent gathers most of the information it needs to answer the main agent question and the evidence needed.

Debug Breakpoint: This tool simply allows the subagent to set or remove any breakpoints. It also allows for listing all the breakpoints. For toggling breakpoints, we allow line and method breakpoints, which set the breakpoints on a specific line or the entrance of a method respectively. This is to allow the LLM the flexibility to set breakpoints based on method names, which are less error-prone than line numbers.

In addition to these tools, the Debug Subagent is also equipped with tools for file navigation, grep search and reading files to allow the subagent to gather context before setting breakpoints, deciding execution control, interpreting call stacks, etc. We leave out tools for file editing because the goal isn't to fix the issue only to identify the root cause. Finally, Figure 3 shows the system prompt we use for our subagent.

```
You are a Runtime Oracle - a debugging assistant that
answers specific questions about program execution.

Assumption
The project is ALREADY BUILT. The main agent has compiled
the code before calling you. Do NOT attempt to build the
project yourself - go directly to debugging.

Your Role
You answer questions about runtime behavior by:
1. Starting a debug session with initial breakpoints
2. Inspecting variables when breakpoints hit
3. Stepping through code as needed
4. Returning factual, verifiable answers

Question Types You Handle
- Variable Inspection: "What is the value of X at line Y?"
- Reachability: "Does execution reach line Z during test T?"
- Condition Evaluation: "Why does condition X evaluate to true?"
- Exception Origin: "What causes the NullPointerException?"

Output Format (REQUIRED)
Always end your investigation with a <debug_answer> block:
  <debug_answer>
  **Question**: [The question you were asked]
  **Answer**: [Direct, factual answer]
  **Evidence**: [Variable values, stack frames observed]
  **Location**: [File:line where you observed this]
  </debug_answer>

Tools Available
- debug_start_session: Start session with test and breakpoints
- debug_inspect: Inspect variables, evaluate expressions, view stack
- debug_control: Step through code, continue, terminate
- debug_breakpoint: Add or remove breakpoints
- read_file: Read source code for context
```

Figure 3: System prompt for the Debug Subagent. The prompt explains the role of the subagent to the LLM along with descriptions of the kinds of questions, tools available and the output format required.

4.3 Main Agent Integration

Next, we describe how the Debug Subagent integrates with the main agent itself. We integrate the Debug Subagent as one of the tools available to the main agent, alongside tools like (Bash, Read, Grep, etc.). We also augment the main agent's system prompt (Figure 4) to describe when and how to invoke the Debug Subagent. This describes the kinds of tasks the main agent should delegate to the subagent: inspecting runtime values, root cause analysis, verifying fixes, etc. In our Qualitative Analysis, we see the Debug Subagent actually being used for these kinds of tasks.

These prompt changes describe the high-level workflow shown in Figure 1 to the LLM, where we want the LLM to call the Debug Subagent before making any changes to the code instead of doing print-debugging. This mirrors the natural workflow of an experienced developer, who reaches for the debugger when faced with a complex issue where static code analysis isn't enough. In principle, these changes would suffice but we face another interesting problem, which we describe next.

4.3.1 The Problem of Tool Under-utilization. In some of our experiments, simply giving the main agent the subagent wasn't enough. The usage of the subagent varied significantly by model [9] (Table 2). To solve this problem, we employ a two-part strategy: 1) rather than exposing the debug tools directly, we expose the subagent and only the subagent to the main agent, 2) for bug-fixing tasks, we disable all file-editing tools until the debug tool has been called at least once. Following the workflow shown in Figure 1, We believe that the main agent shouldn't have to modify any files until it has done root cause analysis with the Debug Subagent. It's allowed to navigate the codebase and view files, but it can only modify files after debugging the issue via the subagent. We justify this design choice with ablation experiments in our evaluation.

5 EXPERIMENTAL SETUP

In this section, we talk about our experimental setup to evaluate the impact of adding a Debug Subagent following the methodology described earlier.

5.1 Languages

We implement Debug2Fix for two languages: Java and Python. For Java, we integrate the Java Debugger (JDB), a command

```
You are a highly sophisticated automated coding agent
with expert-level knowledge across many different
programming languages and frameworks.
The user will ask a question, or ask you to perform
a task, and it may require lots of research to answer
correctly. There is a selection of tools that let you
perform actions or retrieve helpful context...

== Using debug_subagent for Bug Fixing ==

You have access to 'debug_subagent' - a debugging
tool that can inspect runtime values, trace execution,
and help verify fixes. Use it to understand bugs
before making changes.

Recommended Workflow:
Step 1: Build the project first
  mvn test-compile -q
Step 2: Understand the bug (before making changes)
  debug_subagent({question: "What exception
  occurs when running MyTest#testMethod?",
  test: "com.example.MyTest#testMethod"})
Step 3: Investigate root cause
  debug_subagent({question: "What is the value
  of [variable] at [location]?"})
Step 4: Apply your fix
Step 5: Verify the fix works
  debug_subagent({question: "Does the test pass
  now after my fix?"})

If you aren't sure which tool is relevant, you can
call multiple tools. You can call tools repeatedly to
take actions or gather as much context as needed...
```

Figure 4: Instructions added (green) to the main agent system prompt as part of the Debug2Fix framework. We inject a dedicated section that introduces the Debug Subagent and provides a recommended workflow for bug-fixing tasks.

line debugger that comes with the Java Development Kit (JDK). We add support for both Maven and Gradle build systems, which are automatically detected based on the files in the codebase. Since building the projects can be a long process, we use incremental build settings that are supported by these build systems.

For Python, we integrate the Python Debugger (PDB), which is part of the standard Python library and comes built in with the language. Unlike Java, Python does not require a separate compilation step and PDB can be invoked directly on any Python script. However, we added support for the more standard PyTest-based test execution as well.

5.2 Benchmarks

We use the following two benchmarks for our evaluation:

- **SWE-Bench-Live**: A benchmark [11] containing GitHub issues from 93 repos, filtered to have issues created after 2024 to minimize contamination. We use a subset of 400 python examples from their frozen Verified split.
- **GitBug-Java**: A Java benchmark [10] from 55 notable open-source repos on GitHub. We use a subset of 186 examples for which we were able to successfully execute the provided docker images and evaluation harness.

Both benchmarks use a test-based verification i.e. a fix is considered correct if all relevant tests pass after the agent fix has been applied.

5.3 Metrics

We evaluate our approach using the following metrics:

- **Pass Rate (%)**: We observe the change in pass rate of the agent across different configurations.
- **Call Rate (%)**: This is the % of instances in the run where the Debug Subagent was invoked. This helps us understand if the subagent is actually being used, which is important for our ablation study.
- **Avg. Step Count**: This is the average number of steps taken by main agent and the Debug Subagent per instance.
- **Avg. Token Usage**: This is the sum of average input and output tokens used by an instance in a given run. This along with number of steps captures the computational cost and latency of running the agent with our subagent architecture.

Along with all these metrics, we also report the change a configuration has on that metric as a percentage of baseline value, shown in brackets after the actual metric.

5.4 Models Configurations

We evaluate Debug2Fix over 3 popular frontier LLMs from OpenAI and Anthropic: GPT-5, Claude Sonnet 4.5 and Claude Haiku 4.5. For all our models, we use the same model for both our main agent and Debug Subagent. We leave experiments with different model combinations (stronger model for main agent and smaller finetuned model for subagent) up to future experimentation.

5.5 Ablations

We ablate over different design choices via the following configurations:

- **Baseline**: The agent without any modifications.
- **Debug Tools Only**: We add the debug tools defined in the Debug Subagent Tools section, but we expose them to the main agent directly. This is meant to evaluate the benefit of having a subagent architecture.
- **Debug2Fix**: We incorporate the Debug Subagent in the agent, as described in the Methodology.
- **Debug2Fix (w/ Tool Limit)**: We incorporate Debug Subagent and also disable Edit tools until the agent has called the Debug Subagent at least once.

For all the configurations that use the Debug Subagent, we limit each Debug Subagent trajectory to be at most 25 steps each. So, in other words, the Debug Subagent cannot take more than 25 steps at a time to investigate an issue. If the subagent hasn't finished by then, we prompt the agent to generate the final answer by injecting a user prompt into the trajectory and querying the LLM.

6 RESULTS

We evaluate Debug2Fix across two benchmarks and three frontier LLMs from OpenAI and Anthropic. Table 1 summarizes our findings on GitBug-Java and SWE-Bench Live. For GitBug-Java, we conduct a detailed ablation study across all configurations described earlier. We then use SWE-Bench Live to validate whether our approach generalizes to Python. In this section, we drill deeper into the results of our runs.

6.1 Main Results

For GitBug-Java, Debug2Fix improves the pass rate across all three models. GPT-5 sees the largest gains, jumping >20% over the baseline from 60.2% to 73.1%. Claude Sonnet also improves by ~13% over baseline, while Claude Haiku sees ~16% improvement.

We can also see that with the Debug2Fix configuration with the tool-limit in place, we see a >98% call rate for Debug Subagent, showing that our strategy is effective in encouraging debugger usage. In contrast, when not used without the tool limit, we only see a 60-70% call rate across the models. The Debug Subagent isn't provided to the Baseline and Debug Tools Only configuration, so the corresponding fields are left blank.

For SWE-Bench-Live, we evaluate the Debug2Fix agent without tool-limiting to understand the natural adoption patterns across models. GPT-5 shows the highest call rate of >60% and has the corresponding largest improvement (16%). Claude Haiku 4.5 demonstrates moderate usage (33.2%) and seems modest gains as well of 12%. Finally, we can see that when naturally prompted, Claude Sonnet 4.5 only calls the subagent only ~8% of the instances and sees minimal improvement. This makes sense because we haven't really changed the agent's workflow if the tool didn't trigger and it achieves a similar performance as the baseline. Further, this example also shows our approach generalizing to Python despite there being differences between the interfaces of JDB and PDB and how they're used.

6.2 Ablation Analysis

Our ablation study helps us isolate the contribution of the two key design choices: the subagent architecture and tool-limiting strategy.

Exposing debug tools directly is ineffective and even harmful: When we provide the main agent with the debug tools directly (start session, breakpoint, inspect, etc.), without a subagent wrapper performance either remains flat or degrades significantly. We see negligible change for GPT-5 and Claude Haiku, but for Claude Sonnet we see a drop of ~18% over baseline. Manual inspection of the trajectories revealed that the main agent rarely leverages the debug tools, calling at least one debug tool in only 17 (9%) of the instances. Interestingly, it resolves 14 (~82%) of those instances, while in the baseline only 11 (~64%) of those instances succeed. In the cases where it uses debug tools, the subsequences corresponding to debugging are very short (≤4 tool calls) compared to the Debug2Fix trajectories, where the agent can go into deeper debugging sessions to investigate complex questions posed by the main agent. Furthermore, the subagent doesn't get confused about tool orchestration because it has limited tool selection and has the sole purpose of debugging, unlike the main agent when presented with debug tools alongside its usual tool-set and is used for a wide range of tasks.

The subagent alone is often insufficient without tool-limiting Adding the Debug Subagent as a tool and updating the prompt to let the model know of its presence yields varying call rates across models. Based on our results on GitBug-Java (Table 1), GPT-5 seems to be the most suggestible when it comes to adding a new tool for the agent to call, while Claude Sonnet 4.5 seems the most unwilling. This is unfortunate given how its the stronger model of the group and would result in even better overall performance should it leverage the tools. For GitBug-Java, by enforcing debugging before editing, our approach was able to raise the call rates to ~99% for all models and produce performance gains across the models.

New tool's call-rate can vary by language Directly comparing call rates between GitBug-Java and SWE-Bench Live for all our models, we can see how the willingness of models to adopt a Debug Subagent changes by language. In Java, all three models show a similar call rate by default (60-70%), as seen in the Debug2Fix configuration. However, for Python the call rates diverge dramatically between the models. GPT-5 maintains it's >60% call rate, but Haiku and Sonnet the call rates drop considerably. We hypothesize that this may be because certain models are more confident in their ability to fix Python issues statically or via print-debugging without needing a debugger, which is what they were likely trained to do. However, this reluctance to adopt the Debug Subagent also reflects the smaller improvements seen in their performance (Table 2).

Computational overhead to adding such a subagent A natural concern with adding a new subagent with its own context and trajectory is the increased cost and added latency, not including the cost of running the debugger itself. LLM tokens and agent steps directly translate to cost and latency measures. Table 1 shows the breakdown of steps and tokens used by the main and subagent for GitBug-Java benchmark. This is so we can measure the change in the main agent's step count and token usage when it uses the subagent. For Claude Sonnet 4.5, we can see that we actually reduce the main agents token usage and steps when using our best configuration, while the subagent adds on average 33 steps and 400k tokens. For GPT-5 and Claude Haiku 4.5, the total tokens increases

Table 1: Comparison of success rate for each configuration. We also show the breakdown of steps and tokens used by the main agent and the Debug Subagent within each of our configurations. For each number, we also show the relative improvement or decrease in performance over baseline.

(a) GitBug-Java

Metric	GPT-5		Claude Haiku 4.5		Claude Sonnet 4.5	
Pass & Call Rate (%)						
	Pass %	Call %	Pass %	Call %	Pass %	Call %
Baseline	60.2	-	71.0	-	75.7	-
Debug Tools Only	60.8 (+1.0%)	-	70.4 (-0.8%)	-	64.5 (-14.8%)	-
Debug2Fix	64.0 (+6.3%)	64.5	76.1 (+7.2%)	69.4	78.0 (+3.0%)	70.4
Debug2Fix (w/ Tool Limit)	**73.1** (+21.8%)	99.5	**82.3** (+15.9%)	98.9	**85.5** (+12.9%)	98.9
Avg. Steps						
	Main	Sub	Main	Sub	Main	Sub
Baseline	15.7	-	42.8	-	35.7	-
Debug Tools Only	27.9 (+77.7%)	-	44.4 (+3.7%)	-	33.3 (-6.7%)	-
Debug2Fix	19.1 (+21.7%)	33.7	43.2 (+0.9%)	52.1	34.4 (-3.6%)	39.8
Debug2Fix (w/ Tool Limit)	23.6 (+50.3%)	25.9	47.5 (+11.0%)	44.5	33.7 (-5.6%)	33.1
Avg. Tokens (Input + Output)						
	Main	Sub	Main	Sub	Main	Sub
Baseline	347k	-	1.55M	-	1.22M	-
Debug Tools Only	802k (+131%)	-	2.64M (+70.3%)	-	1.15M (-5.7%)	-
Debug2Fix	442k (+27.4%)	479k	1.71M (+10.3%)	760k	1.12M (-8.2%)	478k
Debug2Fix (w/ Tool Limit)	645k (+85.9%)	350k	1.75M (+12.9%)	619k	978k (-19.8%)	396k

Table 2: Comparing the performances, token usages and steps used by Debug2Fix and Baseline agent on the Python subset of SWE-Bench Live dataset.

(a) SWE-Bench Live (Python)

Metric	GPT-5		Claude Haiku 4.5		Claude Sonnet 4.5	
Pass & Call Rate (%)						
	Pass %	Call %	Pass %	Call %	Pass %	Call %
Baseline	31.2	-	34.3	-	39.6	-
Debug2Fix	**36.2** (+16.0%)	61.4	**38.5** (+12.2%)	33.2	**40.4** (+2.0%)	8.1
Avg. Steps						
	Main	Sub	Main	Sub	Main	Sub
Baseline	20.0	-	55.4	-	53.0	-
Debug2Fix	21.9 (+9.5%)	24.6	57.5 (+3.8%)	28.6	54.4 (+2.6%)	31.4
Avg. Tokens (Input + Output)						
	Main	Sub	Main	Sub	Main	Sub
Baseline	550k	-	1.97M	-	1.96M	-
Debug2Fix	630k (+14.5%)	350k	2.08M (+5.6%)	371k	1.97M (+0.7%)	281k

modestly. Examining the steps taken by each model, we can see that they also go up in most cases except for Claude Sonnet.

This variability in tool adoption across models and languages presents a challenge for agent developers. We encourage model providers to improve instruction-following for novel tools so that these systems can be made extensible. Models that selectively ignore available tools would continue to demonstrate poor performance when faced with real-world problem solving.

Better tooling closes the gap between models An interesting consequence of us incorporating a Debug Subagent is that, Debug2Fix allows weaker models to match or even exceed the performance of stronger models' baseline performance. As one can see for GitBug-Java, **GPT-5 with Debug2Fix scores**

(73.1%) nearly the same as baseline Sonnet (75.7%), despite the 15% gap in their respective baselines. Similarly, **Debug2Fix helps Claude Haiku 4.5 outperform baseline Sonnet 4.5 by 6.6%**. This suggests that equipping agents with better tooling can be just as, if not more impactful than simply upgrading to a more capable model, which has been the belief in Coding Agent community. This finding has implications for cost-sensitive areas where smaller or cheaper models with better tooling may be preferably to an expensive larger model.

6.3 Qualitative Analysis

In addition to our Ablation Analysis, we conduct a qualitative analysis over the trajectories generated during our runs to better understand how agents leverage the Debug Subagent in practice. We first examine the kinds of runtime questions the main agent asks the Debug Subagent and whether this is in accordance with how it was instructed to use the subagent. Second, we identify common failure modes by inspecting the cases where the Debug Subagent was invoked, but the agent failed anyway. For all these analyses, we use the trajectories generated during the GPT-5 run over GitBug-Java under the Debug2Fix (w/ Tool Limit) configuration.

6.3.1 Debug Subagent Usage Patterns. We randomly sample 50 trajectories over GitBug-Java benchmark where the Debug Subagent was invoked by the main agent. We then look at all the questions asked by the main agent and manually classify them into 7 distinct categories shown in Table 3. For each category, we show how many instances had those categories of questions and the percentage of questions that fell under that category. Note that a given benchmark instance may ask multiple questions falling in different categories.

The most common use case is notably Exception Diagnosis, which covers ~28% of the questions asked. This is where the agent asks the subagent to identify which exception was thrown and where in code it originates. The second most common category is Root Cause Analysis, where the agent asks prying questions about the underlying behavior behind the bug. Local Variable Inspection and Attribute Value Inspection make up ~30% of the cases, where the main agent asks the subagent to observe runtime state and get values of variables and expressions. These categories mirror the core use cases developers use debuggers for in the real life and also what the main agent was asked to use the Debug Subagent for (Figure 4).

6.3.2 Typical Debug Subagent Workflow. Figure 5 shows the distribution of debug tool calls across step positions within the trajectories taken by the subagent. The plot shows a pattern to how the subagent orchestrates its debugging sessions. We can see that the first step is always Debug Start Session. This is by design of the subagent, where we execute the start session tool that atomically starts the program, attaches JDB / PDB and sets the initial breakpoints. We can see that the subsequent steps are dominated by Debug Inspect and Debug Control calls as the agent alternates between examining variables or

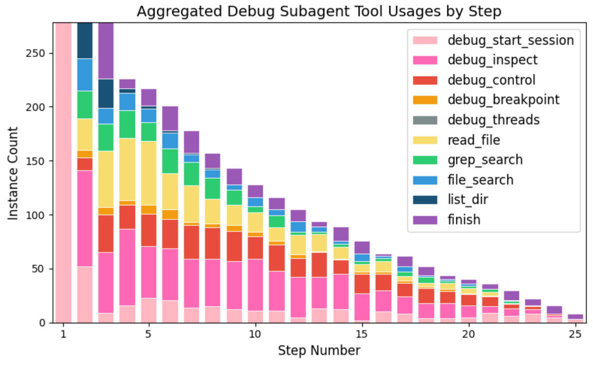


Figure 5: Plot showing an aggregated view of all the trajectories taken by the Debug Subagent. Each step shows a distribution of tools called within it. We can see the plot tapering to the right because more trajectories resolves as the subagent takes more steps.

expression values and stepping through execution. We also see the subagent reading files, grepping and setting breakpoints, which becomes less frequent in the later portion of the plot.

The plot itself tapers to the right because more and more trajectories resolve and terminate, as seen via the Finish calls throughout the plot. Most trajectories finish within 10-15 steps, though many still go on until Debug Subagent trajectory limit of 25 steps.

6.3.3 Common Debug2Fix Failure Modes. To understand the limitations of our approach, we manually analyze a sample of 50 failed instances in GitBug-Java run, where the agent invoked the Debug Subagent, but still failed to resolve the bug. Table 4 shows a categorization of the different failure modes we observed in the data.

We can see that a big chunk of failures stem from infrastructure limitations and issues with the main agent itself rather than the Debug Subagent. We observe that the most prevalent failure mode is due to the debug session failing, which manifests as Debug Start Session tool call timing out in the trajectory. This is caused by the Debug Subagent not being able to attach to the test process due to some build failure. This highlights how buildability is a prerequisite for our approach i.e. if the project doesn't compile, the Debug Subagent won't be able to provide runtime insights. The second largest category is the main agent implementing the wrong fix despite correct answer from Debug Subagent. Following these, we have issues having to do with bugs requiring more than 3 debug sessions, the request failing or the subagent resorting to static analysis in some cases. These findings suggest that there may be potential improvements we can make to the underlying build logic behind the subagent, add retry logic as well as improve the translation of Debug Subagent diagnostics into fixes from the main agent. We leave these explorations to future work.

Table 3: Categorization of the kinds of questions we observed the main agent posing to the Debug Subagent in a sample of 50 instances.

Category	Description	% Cases	# Instances
Exception Diagnosis	Identifying the type, origin, or cause of runtime exceptions.	27.8%	17
Root Cause Analysis	Understanding why a specific behavior or bug occurs.	25.3%	15
Local Variable Inspection	Querying the value of specific variables at a location.	15.2%	11
Attribute Value Inspection	Inspecting object attributes or map entries.	15.2%	9
Assertion Failure	Identifying which assertion fails and the actual vs expected values.	6.3%	5
Code Reachability	Checking if execution reaches a specific location or branch.	5.1%	4
Post-Fix Verification	Confirming that a code change resolves the issue.	5.1%	3

Table 4: Failure modes observed in failed Debug2Fix instances despite having the Debug Subagent.

Failure Mode	Description	% Cases	# Instances
Debugger Session Failed	Debug Subagent could not attach JDB due to failed build/test. This could be due to a variety of reasons like missing Gradle task, test process exited before debugger attached, etc.	36%	18
Wrong Fix Despite Correct Debugging	Debug Subagent successfully answered the runtime question, but the main agent applied an incorrect or incomplete fix.	34%	17
High Complexity Bug	Bug required >3 debug sessions, yet the agent could not converge on the correct fix despite these multiple attempts.	16%	8
Subagent API Error	Debug Subagent request failed due to server errors, which results in an empty response.	8%	4
Subagent Static Analysis	Debug Subagent ends up doing static code analysis, providing incomplete information without actual runtime values.	6%	3

7 LIMITATIONS

While our study provides a valuable framework for agents to follow, we would like to highlight several limitations that present opportunities for future explorations.

Language & Benchmark Coverage While our evaluation shows that our approach can we successfully applied to Python and Java, we have not tried this approach to other languages with mature ecosystems like C, C++, C#, Rust, etc. Furthermore, we only tried one benchmark for both languages studied. In future work, we would like to explore more languages and benchmarks like SWE-Bench Pro [6] and Multi-SWE-Bench [7] with greater language coverage.

Project Buildability Our approach seems to rely on whether the project can be built and has executable tests. As seen in our Qualitative Analysis, 36% of our failed instances stem from this issue. While this may be a reasonable constraint because developers do ultimately need to build and run the projects to do debugging themselves, this limits us to project with working build systems.

Using Different Models We use the same model for both the main and subagent across all our experiments. However, alternate configurations such as pairing different models for main agent and subagent may provide even better results. Potentially, one could even finetune a smaller model specialized for debugging, which may reduce cost of running such an agent. We leave these explorations to future work.

Tool-Limiting Trade-off While our tool limiting strategy worked for one of our benchmarks, it enforces a very rigid workflow on these models. This may lead to suboptimal performance or over-complication of simpler bugs that can be fixed without runtime information. A more adaptive strategy that selectively decides when to disable editing before debugging is needed to adapt this approach to real world agents.

Debugging Overhead While we measure the token usage and steps taken by the debugging subagent, we don't measure the computational cost of running the debugger itself. It may not always be possible to run a debugger depending on the environment.

8 CONCLUSION

In this work, we presented Debug2Fix, a framework that incorporates interactive debugging capabilities into coding agents through a specialized Debug Subagent. Our approach abstracts low-level debugger commands behind a unified interface that can be leveraged by the main agent without having to use debugger commands itself. Through extensive evaluation consisting of an ablation study and qualitative analysis of the trajectories, we show that Debug2Fix yields substantial improvements across all evaluated models and languages we study. Notably, we saw GPT-5 achieving a 21.8% relative improvement over baseline, Claude Haiku 4.5 improving by 15.9% and Claude Sonnet 4.5 improving by 12.9%.

One of the key findings of our approach is that better tooling can close the gap between models. With Debug2Fix, GPT-5 achieves nearly the same pass rate as baseline Claude Sonnet 4.5 on the same benchmark. Similarly, Claude Haiku 4.5 is able to surpass baseline Claude Sonnet 4.5 results when used with Debug2Fix. Through an ablation study we find that exposing debug tools directly to the main agent is ineffective and can even degrade performance i.e. subagent architecture is necessary. We also see models exhibiting varying degrees of reluctance in adopting new tools.

Debug2Fix represents a strong step towards coding agents that are on par with expert developers for tasks like bug fixing, by mirroring their workflows of using a debugger for tasks where static analysis proves insufficient. We think that this work will allow LLM-based coding agents to take on increasingly complex tasks. Finally, as a call to action, we implore LLM providers to support better instruction following, so that LLMs leverage novel tools more readily to enable more extensible and capable agent systems.

REFERENCES

[1] Anthropic, “Claude for Coding,” https://www.anthropic.com/claude-code, 2024, accessed: 2025-07-14.

[2] Microsoft, “VSCode Agent Mode,” https://code.visualstudio.com/blogs/2025/04/07/agentMode, 2025, accessed: 2025-09-28.

[3] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, “Opendevin: An open platform for ai software developers as generalist agents,” 2024. [Online]. Available: https://arxiv.org/abs/2407.16741

[4] S. Garg, B. Steenhoek, and Y. Huang, “Saving swe-bench: A benchmark mutation approach for realistic agent evaluation,” 2026. [Online]. Available: https://arxiv.org/abs/2510.08996

[5] S. Liang, S. Garg, and R. Z. Moghaddam, “The swe-bench illusion: When state-of-the-art llms remember instead of reason,” 2025. [Online]. Available: https://arxiv.org/abs/2506.12286

[6] X. Deng, J. Da, E. Pan, Y. Y. He, C. Ide, K. Garg, N. Lauffer, A. Park, N. Pasari, C. Rane, K. Sampath, M. Krishnan, S. Kundurthy, S. Hendryx, Z. Wang, V. Bharadwaj, J. Holm, R. Aluri, C. B. C. Zhang, N. Jacobson, B. Liu, and B. Kenstler, “Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?” 2025. [Online]. Available: https://arxiv.org/abs/2509.16941

[7] D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang, “Multi-swe-bench: A multilingual benchmark for issue resolving,” 2025. [Online]. Available: https://arxiv.org/abs/2504.02605

[8] Y. Wang, M. Pradel, and Z. Liu, “Are "solved issues" in swe-bench really solved correctly? an empirical study,” *ArXiv*, vol. abs/2503.15223, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:277113006

[9] G. Mo, W. Zhong, J. Chen, X. Chen, Y. Lu, H. Lin, B. He, X. Han, and L. Sun, “Livemcpbench: Can agents navigate an ocean of mcp tools?” 2025. [Online]. Available: https://arxiv.org/abs/2508.01780

[10] A. Silva, N. Saavedra, and M. Monperrus, “Gitbug-java: A reproducible benchmark of recent java bugs,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 118–122. [Online]. Available: https://doi.org/10.1145/3643991.3644884

[11] L. Zhang, S. He, C. Zhang, Y. Kang, B. Li, C. Xie, J. Wang, M. Wang, Y. Huang, S. Fu, E. Nallipogu, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, “Swe-bench goes live!” 2025. [Online]. Available: https://arxiv.org/abs/2505.23419

[12] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, “Explainable automated debugging via large language model-driven scientific debugging,” 2023. [Online]. Available: https://arxiv.org/abs/2304.02195

[13] K. H. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, “Chatdbg: Augmenting debugging with large language models,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, p. 1892–1913, Jun. 2025. [Online]. Available: http://dx.doi.org/10.1145/3729355

[14] L. Zhong, Z. Wang, and J. Shang, “Debug like a human: A large language model debugger via verifying runtime execution step-by-step,” 2024. [Online]. Available: https://arxiv.org/abs/2402.16906

[15] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “Swe-agent: Agent-computer interfaces enable automated software engineering,” 2024.

[16] GitHub, “GitHub Copilot Agent,” https://github.blog/news-insights/product-news/github-copilot-meet-the-new-coding-agent/, 2024, accessed: 2025-07-14.

[17] Windsurf, “https://windsurf.com/,” 2024, accessed: 2025-07-14.

[18] J. He, C. Treude, and D. Lo, “Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead,” 2025. [Online]. Available: https://arxiv.org/abs/2404.04834

[19] D. Arora, A. Sonwane, N. Wadhwa, A. Mehrotra, S. Utpala, R. Bairi, A. Kanade, and N. Natarajan, “Masai: Modular architecture for software-engineering ai agents,” 2024. [Online]. Available: https://arxiv.org/abs/2406.11638

[20] M. Tufano, A. Agarwal, J. Jang, R. Z. Moghaddam, and N. Sundaresan, “Autodev: Automated ai-driven development,” 2024. [Online]. Available: https://arxiv.org/abs/2403.08299

[21] M. A. Islam, M. E. Ali, and M. R. Parvez, “Mapcoder: Multi-agent code generation for competitive problem solving,” 2024. [Online]. Available: https://arxiv.org/abs/2405.11403