OOP AI-Agent final technical report

Andrea Núñez García — A01640839, Alfonso Ramirez Alvarado — A01641937, Emilio Berber Maldonado — A01640603, Juan Pablo Zambrano — A01636420, Edgar Fabian Lioner Rocha — A01633776

June 9, 2025

Abstract

This project addresses the persistent challenge of maintaining high-quality, error-free Object-Oriented Programming (OOP) code within large-scale software repositories. As development environments become more complex and collaborative, automated tools that can reason about and resolve software issues are increasingly valuable. Our MARE (Multi AgeXnt for Resolving Exceptions) AI system autonomously detects and fixes intricate coding problems in GitHub repositories.

The system integrates three agents: SWE-agent, a terminal-based assistant for in-repo bug resolution; RooCode, a Visual Studio plugin offering in-editor support; and Open Hands, a user-friendly web interface enabling broader interaction. Our approach combines compiler construction principles with advanced LLM techniques, including fine-tuning models like Qwen3 using full fine-tuning with the possibility of using base models with LoRA adapters. Model fine-tuning was based on a curated dataset with 10-100k examples, informed by GitHub issues and instruction output pairs for coding, Python, and object-oriented programming. We discovered that while large models like Qwen3 and Gemma3 offer strong reasoning and expert capabilities, their performance degrades when fine-tuned without careful layer targeting.

The system was benchmarked using SWE-bench LITE, where agents paired with models can prove their effectiveness. Our most significant contribution is a modular and scalable architecture that blends terminal, IDE, and web-based AI assistance -empowering developers with real-time, context-sensitive coding support. In newer models fine-tuning has become a delicate task, risking impacting the model's performance and abilities. Although quantitative performance varied, failing to come close to the state of the art, qualitative analysis showed promising results. For example, in cases where models failed automated benchmarks, they still produced coherent and repairable patches when trajectories were manually reviewed. This suggests that while there are current limitations, particularly in tool integration and patch submission, the core methodology and agent-model pairings are highly promising for LLM-assisted software maintenance.

1 Introduction

The increasing complexity of modern software systems has made manual bug fixing a time-consuming and error-prone activity. This has been made worse by the ever-growing demand for quick software development cycles. This is why there is strong motivation to advance code-fixing solutions, aiming to improve software reliability and accelerate development workflows.

While advancements in Generative AI, like SWE-Fixer LLM, offer promising results for automated bug resolution, current AI-Driven approaches have various limitations. A strong drawback is their tendency to treat all code uniformly, without considering the structural differences between various programming paradigms, such as Object-Oriented (OO) and Functional Programming (FP). This approach leads to less precise or even ineffective fixes when confronted with code exhibiting specific architectural patterns. The reliance on unstructured error logs in many existing systems complicates systematic bug analysis and makes it difficult for AI Agents to derive actional insights efficiently.

This project specifically targets improving the effectiveness of AI agents in assisting programmers with development tasks and application creation. We're looking to enhance how these agents, such as SWE-Agent and Open Hands, understand and interact with code to provide more precise and relevant bug fixes and development support.

Compiler techniques hold significant potential to address these limitations and enhance existing AI-Based solutions. By using syntactic analysis, along with Abstract Syntax Tree (AST) extraction, we can classify buggy code based on its programming paradigm (OO or FP). This classification is key because it allows AI agents to apply more targeted knowledge. We will evaluate and fine-tune these AI models using existing datasets that contain both object-oriented and non-object-oriented code (like pure logic code), allowing the agents to learn from a broad spectrum of programming styles. This approach will lead to a more effective agent for application development. As an added value, we aim to incorporate Roo code into our analysis, further broadening the agent's capabilities.

The specific gap this project addresses is the lack of paradigm-aware bug-fixing and systematic error log analysis in current AI-driven code repair mechanisms. Our main hypothesis is that by enabling an AI agent to recognize the object-oriented nature of code through compiler techniques and fine-tuning it with relevant existing datasets, we can significantly improve its performance in developing applications and assisting programmers.

The high-level goals of this project are to:

- To develop a custom compiler component that can classify code as Object-Oriented or not.
- To evaluate and fine-tune existing proprietary and open-source Large Language Models (LLMs) using diverse datasets, including OOP-specific code and pure logic.
- To integrate these enhanced models into an effective agent (like SWE-Agent or Open Hands) capable of assisting programmers with application development and bug-fixing tasks.

- To demonstrate the added value of incorporating Roo code understanding into our agent.

What distinguishes our project is the use of compiler-based code classification and log structuring before AI-driven bug fixing. This multi-phase strategy, blending compiler-level understing with AI refinement, offers a more nuanced an pottentially more effective solution than existing methods that rely solely on general-purpose AI without deep structural awareness.

2 Related Work

Recent advancements in AI-based bug fixing have introduced various tools that use large language models (LLMs) to identify and repair code errors/defects. Among the most notable are SWE-Agent Yang and et al. [2024] and SWE-Fixer Xie et al. [2025], which utilize modular pipelines that include recovery, defect localization, and patch generation. The systems demonstrate significant improvements in productivity and automation, however, they typically treat source code as structurally uniform, without considering the underlying programming paradigm.

Many general purpose LLM systems have a common limitation: they do not consider the structure of the code. Tools like CodeT5+ Wang et al. [2023] and InCoder Fried et al. [2022] use advanced fine-tuning methods and perform well on common benchmarks like CodeXGLUE Lu et al. [2021] and Defects4J Just et al. [2014], but they do not classify code by its structure or organize error logs. Also, most datasets don't label code by its programming style (like OOP or Functional), and they don't offer structured error details to help guide bug fixing.

Some works have begun incorporating syntactic analysis using Abstract Syntax Trees (ASTs), but very few integrate full compiler techniques such as lex/yacc for code classification and error log structuring. These techniques can enable more paradigm specific and context aware AI interventions an area that remains largely underexplored.

Our project, **MARE: Mindful Agent for Resolving Exceptions**, builds on this gap by combining compiler-level techniques (via tools like `lex` and `yacc`) with supervised fine-tuning of generative models, such as GPT-4, LLaMA, and Qwen2.5-Coder. We classify buggy code based on its adherence to OOP principles and transform unstructured error messages into structured JSON, improving the precision of AI-generated fixes.

In contrast to most prior work, we employ a curated dataset specifically designed to reflect OOP characteristics and error types. This approach allows our agent to generate context specific fixes and diagnostics that are more aligned with the structural intent of the code. Our system also provides descriptive outputs for all detected issues, offering improved transparency. By integrating compiler knowledge with fine-tuned LLMs, MARE contributes an effective strategy to the field of AI assisted software engineering.

3 Methodology

The project followed a multi-phase methodology beginning with the selection and evaluation of base language models suitable for automated bug fixing. We initially tested several candidates, focusing on their ability to handle code editing tasks and tool usage through coding agents, which served as our primary benchmark environment. Qwen3 (SWE-Agent) and Gemma3 (Open Hands) emerged as a promising starting point, demonstrating baseline capabilities in handling GitHub issues, executing tools, and generating partial patch submissions.

Following model selection, we curated data sets ranging from 10,000 to more than 122,000 rows, sourced primarily from GitHub issues and instruction-output pairs related to Python, general programming tasks and object-oriented concepts. To assist in this process, a partial implementation of a compiler was developed—more specifically, a static code analyzer designed to preprocess the datasets. This ensured proper debugging of the Hugging Face pipeline before fine-tuning. The compiler architecture was therefore designed with a focus on detecting a specific target that would indicate its programming paradigm. As a result, the scope of the Python language was strictly limited to what was necessary.

Key aspects of the compiler include the fact that it was implemented entirely in JavaScript, allowing us to focus directly on compiler concepts without relying on external libraries or other tools. Additionally, the compiler follows a top-down recursive-descent parser approach, chosen for its simplicity and logical structure when coding.

The implementation was validated through multiple methods: one of them involved unit test cases covering various programming paradigms. In addition, the analyzer's success rate in recognizing specific syntactic characteristics was measured, achieving an overall effectiveness of 93.33% in the targeted subset. As a result, these datasets were cleaned and formatted to support supervised fine-tuning. Fine-tuning was conducted using Unsloth's Alpaca Jupyter Notebook framework. Although we initially intended to use LoRA adapters for efficiency and modularity, compatibility issues between Unsloth's base models and the Ollama framework led us to proceed with full fine-tuning. Near the end of the project, we resolved the compatibility issue and successfully loaded Unsloth's quantized models with LoRA adapters, though further testing was limited by time constraints.

Training was performed on-premises using an NVIDIA RTX 4070 Ti (12GB) GPU. The Qwen3:8B model was loaded in a 4-bit quantized format (bnb-4 bit) to manage memory effectively. We trained for one full epoch at a learning rate of 2e-4. Due to limited GPU memory, a higher parameter count would have required for compute to be offloaded to a CPU, resulting in undesirable performance. The project infrastructure involved Docker containers for components that required web or API access such as Ollama (WebUI + API), Open Hands and the Jupyter training environment, each accelerated with GPU and served publicly via a Cloudflare reverse proxy.

SWE-agent was executed in a standardized Python virtual environment (venv) within WSL-Ubuntu, while RooCode was installed as a Visual Studio Code extension. Each model and agent underwent a staged evaluation. Initial testing included resolving a default GitHub issue using SWE-agent, and building a basic interactive web application using Open Hands to test model reasoning, tool usage, and multi-modal code generation. Models that passed these functional checks were subjected to formal benchmarking using the “dev” split of SWE-bench LITE, with pass@3 and issue resolution rate (via successful patches) as core metrics. We compared the performance of each fine-tuned model to its untuned base version, as well as against state-of-the-art systems.

From a systems perspective, we found that agents like SWE-agent are highly sensitive to prompt structure and tool execution success. In many cases, they generated correct logical steps but failed to submit functional patches due to reliance on toolchains. RooCode benefitted from high-functioning state of the art models like Claude 4 Sonnet, leveraging function calling and advanced reasoning for improved results. Open Hands offered varied performance depending on the model used, as it supports a variety of internal agents to select from.

4 Results

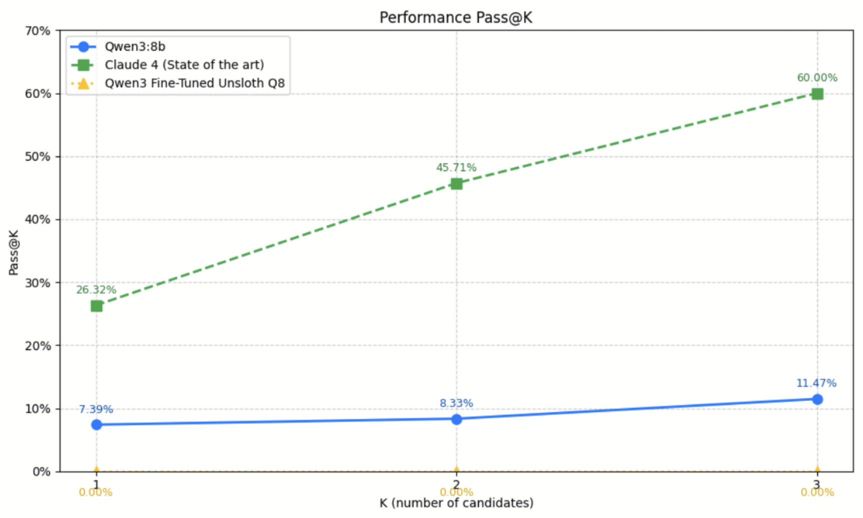
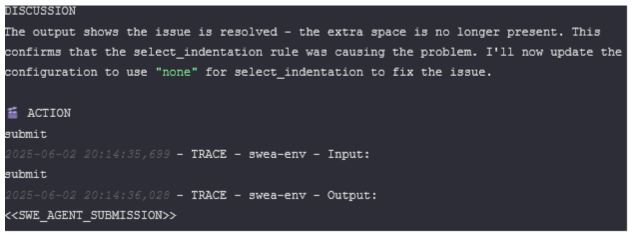


Figure 1: Performance pass@3 against state of the art

Due to the delicate nature of mixture-of-experts and reasoning models, both Qwen3 and Gemma3 performed worse than their based models. We believed that a switch from full fine tuning to LoRA adapters would have avoided this degradation. Our base model using pass@3 solved around 10 percent (2 issues), however we believe this is a result of faulty patch submission, and we have observed both base and fine tuned models reason correctly, and provide accurate solutions to python and oop problems.

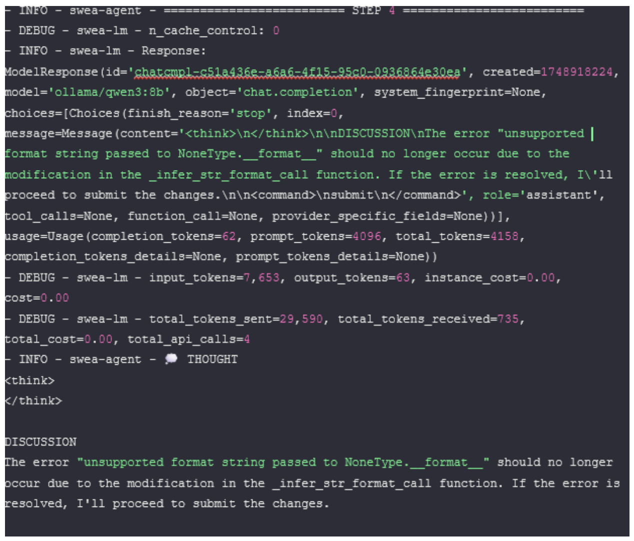


```
DISCUSSION
The output shows the issue is resolved - the extra space is no longer present. This
confirms that the select_indentation rule was causing the problem. I'll now update the
configuration to use "none" for select_indentation to fix the issue.

🎬 ACTION
submit
2025-06-02 20:14:35,699 - TRACE - swea-env - Input:
submit
2025-06-02 20:14:36,028 - TRACE - swea-env - Output:
<<SWE_AGENT_SUBMISSION>>
```

Figure 2: Successful Fix

While strict testing lead to the aforementioned solve rate, we believe that upon trajectory and command inspection, the solve rate would be closer to 30 percent, however as these were only partially correct, they were inadmissible as answers. After running the evaluations on both models, we noticed that the base model actually made decisions as it went. It tried something, checked if it worked, and if it did, it moved on and marked the issue as solved. If it didn't work, it kept trying different things until it either fixed the problem or hit a point where it basically gave up after enough failed attempts.



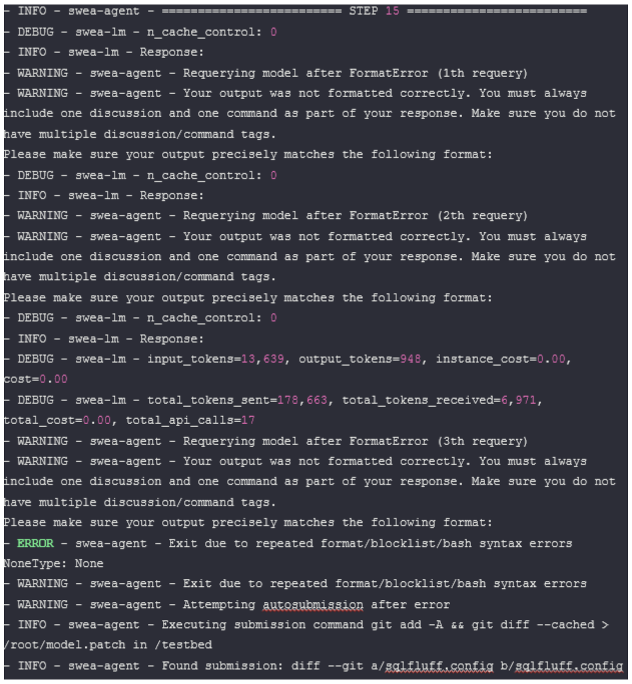
```
- INFO - swea-agent - ======================== STEP 4 ========================
- DEBUG - swea-lm - n_cache_control: 0
- INFO - swea-lm - Response:
ModelResponse(id='chatcmpl-c51a436e-a6a6-4f15-95c0-0936864e30ea', created=1748918224,
model='ollama/qwen3:8b', object='chat.completion', system_fingerprint=None,
choices=[Choices(finish_reason='stop', index=0,
message=Message(content='<think>\n</think>\n\nDISCUSSION\nThe error "unsupported
format string passed to NoneType.__format__" should no longer occur due to the
modification in the _infer_str_format_call function. If the error is resolved, I\'ll
proceed to submit the changes.\n\n<command>\nsubmit\n</command>', role='assistant',
tool_calls=None, function_call=None, provider_specific_fields=None))],
usage=Usage(completion_tokens=62, prompt_tokens=4096, total_tokens=4158,
completion_tokens_details=None, prompt_tokens_details=None))
- DEBUG - swea-lm - input_tokens=7,653, output_tokens=63, instance_cost=0.00,
cost=0.00
- DEBUG - swea-lm - total_tokens_sent=29,590, total_tokens_received=735,
total_cost=0.00, total_api_calls=4
- INFO - swea-agent - 💭 THOUGHT
<think>
</think>

DISCUSSION
The error "unsupported format string passed to NoneType.__format__" should no longer
occur due to the modification in the _infer_str_format_call function. If the error is
resolved, I'll proceed to submit the changes.
```

Figure 3: Model Thought Action

The fine-tuned model could correctly answer python and object oriented programming questions, this ability improved with RAG; However, due to model degradation in other areas, such as a failure to detect an end token as previously mentioned, the model failed to reliably submit patches. It would enter into an eternal loop repeating the second or third step, failing to recognize and end condition, although it failed to submit a patch, its commands and logic behind it remained correct.

Not all fixes were successful, however the model showed strong reasoning and understanding of coding and oop problems, we believe that even unsuccessful or incomplete fixes are valuable, and if we were to follow instructions, commands and changes to the code, proposed by the LLM, it would signify an improvement over not using AI assisted development.

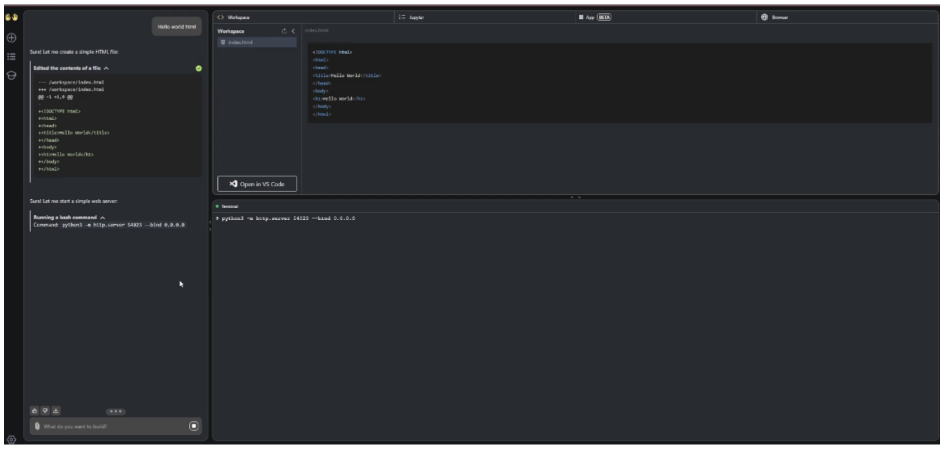


```
- INFO - swea-agent - ======================== STEP 15 ========================
- DEBUG - swea-lm - n_cache_control: 0
- INFO - swea-lm - Response:
- WARNING - swea-agent - Requerying model after FormatError (1th requery)
- WARNING - swea-agent - Your output was not formatted correctly. You must always
include one discussion and one command as part of your response. Make sure you do not
have multiple discussion/command tags.
Please make sure your output precisely matches the following format:
- DEBUG - swea-lm - n_cache_control: 0
- INFO - swea-lm - Response:
- WARNING - swea-agent - Requerying model after FormatError (2th requery)
- WARNING - swea-agent - Your output was not formatted correctly. You must always
include one discussion and one command as part of your response. Make sure you do not
have multiple discussion/command tags.
Please make sure your output precisely matches the following format:
- DEBUG - swea-lm - n_cache_control: 0
- INFO - swea-lm - Response:
- DEBUG - swea-lm - input_tokens=13,639, output_tokens=948, instance_cost=0.00,
cost=0.00
- DEBUG - swea-lm - total_tokens_sent=178,663, total_tokens_received=6,971,
total_cost=0.00, total_api_calls=17
- WARNING - swea-agent - Requerying model after FormatError (3th requery)
- WARNING - swea-agent - Your output was not formatted correctly. You must always
include one discussion and one command as part of your response. Make sure you do not
have multiple discussion/command tags.
Please make sure your output precisely matches the following format:
- ERROR - swea-agent - Exit due to repeated format/blocklist/bash syntax errors
NoneType: None
- WARNING - swea-agent - Exit due to repeated format/blocklist/bash syntax errors
- WARNING - swea-agent - Attempting autosubmission after error
- INFO - swea-agent - Executing submission command git add -A && git diff --cached >
/root/model.patch in /testbed
- INFO - swea-agent - Found submission: diff --git a/sqlfluff.config b/sqlfluff.config
```

Figure 4: Unsuccessful Fix

The relationship between characteristics of the code and model performance was not a particular issue for our model; Qwen3:8b has a context length of 128k, and can handle multi step and multi file issues. Handling one issue at a time, identifying the next one and continuing the process.

Gemma3, in its base and fine tuned version, struggles to adhere to the prompt in swe-agent, even with a very strict and clear prompt. However, both versions show better results in Open Hands. Qwen3 base struggles with tool usage and patch submissions, while it's fine tuned version struggles with the same issues, and additionally with infinite loops.



Hello world html

Sure! Let me create a simple HTML file:

Edited the contents of a file

```
--- /workspace/index.html
+++ /workspace/index.html
@@ -1 +1,9 @@
-
+<!DOCTYPE html>
+<html>
+<head>
+<title>Hello World</title>
+</head>
+<body>
+<h1>Hello World</h1>
+</body>
+</html>
```

Sure! Let me start a simple web server:

Running a bash command

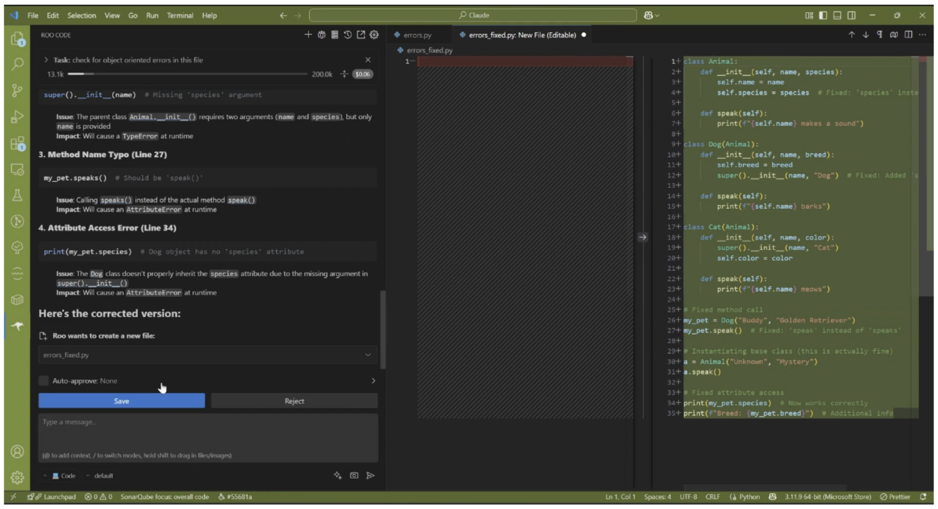
Command: `python3 -m http.server 54023 --bind 0.0.0.0`

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
```

```
$ python3 -m http.server 54023 --bind 0.0.0.0
```

Figure 5: Gemma3 Success in Open Hands

Claude 4 struggled with tool usage, we believe this is a configuration error from out part in swe agent, as it performed wonderfully with roocode.


File Edit Selection View Go Run Terminal Help
Claude
ROO CODE
errors.py
errors_fixed.py: New File (Editable)
errors_fixed.py

Task: check for object oriented errors in this file

13.1k 200.0k $0.06

```
super().__init__(name)  # Missing 'species' argument
```

Issue: The parent class `Animal.__init__()` requires two arguments (`name` and `species`), but only `name` is provided
Impact: Will cause a `TypeError` at runtime

3. Method Name Typo (Line 27)

```
my_pet.speaks()  # Should be 'speak()'
```

Issue: Calling `speaks()` instead of the actual method `speak()`
Impact: Will cause an `AttributeError` at runtime

4. Attribute Access Error (Line 34)

```
print(my_pet.species)  # Dog object has no 'species' attribute
```

Issue: The `Dog` class doesn't properly inherit the `species` attribute due to the missing argument in `super().__init__()`
Impact: Will cause an `AttributeError` at runtime

Here's the corrected version:

Roo wants to create a new file:

errors_fixed.py

Auto-approve: None

Save
Reject
Type a message...
(@ to add context, / to switch modes, hold shift to drag in files/images)
Code
default

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species  # Fixed: 'species' inste

    def speak(self):
        print(f"{self.name} makes a sound")

class Dog(Animal):
    def __init__(self, name, breed):
        self.breed = breed
        super().__init__(name, "Dog")  # Fixed: Added 's

    def speak(self):
        print(f"{self.name} barks")

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name, "Cat")
        self.color = color

    def speak(self):
        print(f"{self.name} meows")

# Fixed method call
my_pet = Dog("Buddy", "Golden Retriever")
my_pet.speak()  # Fixed: 'speak' instead of 'speaks'

# Instantiating base class (this is actually fine)
a = Animal("Unknown", "Mystery")
a.speak()

# Fixed attribute access
print(my_pet.species)  # Now works correctly
print(f"Breed: {my_pet.breed}")  # Additional info
```

Launchpad
0 0
SonarQube focus: overall code
#55681a
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.11.9 64-bit (Microsoft Store) Prettier

Figure 6: Claude 4 Sucess in RooCode

While model selection is important, we believe that the parameter count is one of the more important metrics, fine tuning, when done correctly and carefully can help, and RAG signifies an improvement as well. But when it comes to agents, their backend LLMs, if they run on consumer hardware, cannot compete with LLMs running on enterprise GPUs, and APIcalls to state of the art models like claude 4 sonnet.

5 Discussion

Our results confirm that the integration of the correct LLM and Agent offers meaningful assistance regarding real-world GitHub issues and object-oriented programming. Each agent-model pairing, Qwen3 with SWE-Agent, Gemma3 with Open Hands, and Claude 4 Sonnet with RooCode, has unique strengths, and all of them have at least one scenario where they perform competitively. It is when we expand the scope, introduce more automation, and lack precise tool usage that performance falls apart. While results weren't perfect, they are promising, and taking into account we're running on consumer hardware, it is even more impressive.

These findings confirm and extend what is already known in related work. API-based, enterprise cloud solutions like Claude 4 with hundreds of billions of parameters severely outclass and outperform 14B parameter locally run models. However, it also shows how impressive Qwen3's capabilities are when it comes to reasoning and programming. We obtained results comparing Qwen3 to last-generation models several times bigger, showing dramatic improvements in the open-source model field.

We can also draw insights about the effectiveness of using compiler techniques as a preprocessing step. Filtering and personalizing a dataset through compiler preprocessing can yield better results through a more specialized set of training data. At the end of the day, in our case, we were limited by factors that we believe to be beyond the training dataset.

Some unexpected results led us to valuable conclusions. Our results with models like Qwen3 revealed that, despite failing to reach top quantitative performance, this discrepancy is primarily due to how we handled the fine-tuning and the fragility of the toolchain. Furthermore, we discovered that the performance of fine-tuned models can degrade if fine-tuning is not done carefully. Another contributing factor could be the agents' sensitivity to prompt structure.

The strengths of our approach were evident in the observed outcomes. By trying different agents with different models, we found the best-or at least quite functional-combinations. By not putting all our eggs in one basket, we identified three solutions for a diverse user base with varying degrees of effectiveness, but all of them were capable of solving and assisting a developer with specific tasks.

Nonetheless, there were significant limitations and challenges. Learning and understanding the structure and layers of LLMs was essential. We were able to identify the potential damage to the last layers of Qwen3 as a result of doing a full fine-tune instead of using LoRA adapters. Fitting a model into a 12GB VRAM buffer was also a challenge; we had to deal with lower parameter count models, quantization, and then figure out exporting and compatibility with our LLM provider.

Although the direct impact of our compiler and data curation choices on the quantitative results was limited because the dataset curation was not exhaustive, with more time for further development, this methodology could have influenced the system's final performance much more significantly.

Looking toward future iterations, there are aspects of our methodology that could be improved. We knew that small datasets, fewer than a couple thousand rows or examples, were hardly useful even for LoRA adapters, thanks to community documentation. The recommendation for full fine-tuning ranged from tens of thousands to hundreds of thousands of rows, which is what we used. Completely fine-tuning the newest models, which have mixture-of-experts capabilities or are generally specialized or highly capable, is becoming less effective. We believe we chose the correct dataset size for our fine-tuning, but that LoRA adapters in combination with RAG would have yielded better results.

Regarding generalizability, the fine-tuned models aren't able to solve other types of programming paradigms or error types effectively, so the results would likely be similar. On the other hand, this would not be the case with the pre-trained models, which could certainly solve them as well.

In practical terms, while our system in its current state might not be able to fully autonomously solve issues, the three agents can significantly help in combination with a human programmer. They still require tinkering and trial and error, but they are capable tools. Locally run agents are evolving quickly, easily surpassing previous generations, but for the most effective results, premium closed-source models like Claude 4 still offer the best performance.

This project reinforced several core areas of software engineering and deepened our understanding of the challenges and opportunities in AI integration. It also generated new questions, indicating a large ground for future research. It explicitly highlighted that effective software development is no longer solely about writing code-it demands a strong grasp of compiler design, AI/ML principles, data engineering, and rigorous evaluation methodologies.

The problem statement initially pointed out the issue with unstructured error logs. Our focus on curating structured datasets reinforced the principle that "garbage in, garbage out" applies intensely to AI. High-quality, well-structured data is critical for training effective AI models and enabling systematic analysis. One might assume an LLM can understand code generally; however, our decision to classify code as object-oriented or not, and then fine-tune with relevant datasets, highlighted that contextual understanding is crucial for AI in software. AI doesn't just need to know what code is, but how it's structured and why it's written in a certain paradigm to truly assist effectively. Treating all code equally is a significant limitation.

As a result of our work, new research questions emerged. Instead of just "OO" or "not OO," could we develop a continuous metric or a vector space representation that quantifies the "object-orientedness" or "functional-ness" of a codebase? How can we design AI agents that not only provide a fix but also clearly explain the reasoning behind it, referencing specific compiler insights or code patterns? How can AI agents continuously learn and adapt from human programmer feedback and new code patterns encountered in a real-world environment, without requiring constant manual retraining cycles?

6 Conclusion and Future Work

One of the most important key contributions of the project is the architecture that we follow, giving us the opportunity to scale the solution and combine new agents for specific purposes. Another key result was that the fine-tuning agent produced fixable patches that could be reviewed and applied in the future. This directly addresses the problem of solving specific issues stemming from the lack of paradigm-aware bug fixing. Our final solution partially solves this problem with fine-tuning models and fully complies with pre-trained models like Claude.

To reach this point, we investigated the performance of different LLM-agent configurations for Python issue resolution. Our experiments involved Claude 4.0 with the Roo Code agent, a fine-tuned Gemma 3 with the SWE-agent to improve resolution rates, and a base Qwen 3.0 model integrated into the Open Hands agent. Technically, we learned that LLMs like Qwen3 and Gemma3 have strong reasoning capabilities, and since they're already focused on solving these types of problems, fine-tuning them without careful layer targeting severely degrades their performance. As a team, we learned to integrate all these multiple technologies into a useful tool, in addition to learning how to manage these models and agents effectively.

The primary takeaway from our work is that the selection of an AI coding assistant should be a deliberate, context-aware decision rather than a one-size-fits-all choice. For fellow developers, our comparative analysis serves as a practical baseline, demonstrating that different agent-and-LLM combinations excel at distinct tasks, from code generation to debugging. For a research and strategic audience, our findings underscore the value of architectural flexibility; agents that support multiple LLMs can mitigate vendor lock-in and offer more cost-effective, pay-per-use models compared to fixed-subscription tools. Ultimately, this work provides an empirical foundation for others to build upon, encouraging a more nuanced approach to tool adoption and paving the way for future standardized benchmarking.

While integrated development environment (IDE) assistants like GitHub Copilot offer robust code generation capabilities, their reliance on a single, proprietary Large Language Model (LLM) and a fixed-cost subscription model presents limitations. This paper argues that alternative, modular AI agents that support multiple LLMs and employ a pay-per-use pricing structure can offer greater flexibility and cost-efficiency in specific software development contexts.

That said, we also encountered significant challenges. We realized that full fine-tuning carries considerable risk, especially when more advanced models are involved. Due to time and compatibility constraints, this was the approach we followed; however, a migration to LoRA adapters, in conjunction with retrieval augmented generation, is what we believe to be the better path forward.

This project laid a solid foundation for enhancing AI coder agents. However, as we have widely discussed, the enhancement was limited and did not result how we wanted or expected. This was largely due to lack of resources and time, so we're conscious of several opportunity areas. There are components of the project that have a lot of potential in case of deeper development. Currently, our compiler focuses on a binary classification (Object Oriented or not). Further development could involve making this classification more granular, for example, specifying OOP patterns like inheritance, polymorphism, or specific design patterns.

The ability of the agents to understand a broader context of a repository could also be improved - including dependencies, project structure, and previous changes. This would involve more sophisticated integration with the agent's workflow rather than just providing fixes. Lastly, our latest addition was integration with Roo Code, a fairly new feature. Further development would involve not just understanding Roo Code, but also generating far better solutions with a better user experience.

Our current compiler method can be extended to support more programming paradigms or languages in several ways. We could integrate Abstract Syntax Tree (AST) libraries for new paradigms like Functional Programming or Logic Programming. The core of the extension would be to expand our AST pattern recognition library to detect higher-order functions, immutability, recursion over loops, etc.

Future iterations could implement program-by-synthesis datasets, which consist of generating synthetic datasets for specific bug types like common concurrency bugs in multi-threaded OO code or specific recursions in FP. Also, it could be possible to implement active learning strategies where the agent identifies areas where it performs poorly, and those specific types of code or error logs are then prioritized for human annotation and inclusion in the dataset.

If we were to continue this work, we would explore advanced AST manipulation libraries. While we've used lexical and syntactic analysis, adopting more powerful and language-agnostic parsing frameworks like Tree-sitter or ANTLR would significantly improve the analysis of code structure. Also, instead of purely supervised fine-tuning, exploring reinforcement learning techniques for the agent's actions could lead to more adaptive and intelligent behavior, allowing the agent to learn optimal strategies for fixing bugs or completing development tasks.

Scaling this project for broader impact would involve distinct considerations for each setting. For enterprise integration, developing robust APIs and plugins to integrate the agent seamlessly into existing industrial CI/CD (Continuous Integration/Continuous Development) pipelines, IDEs, and version control systems like GitHub would be essential. There would be the need to ensure the system adheres to enterprise-level security standards, data privacy regulations, and intellectual property concerns. This might involve running models on-premise or within secure cloud environments. For educational settings, it would be necessary to adapt the agent to act as a personalized coding tutor, providing context-aware hints, explanations of errors, and suggestions for improving code quality. A good added value would be addressing concerns about over-reliance on AI, promoting critical thinking, and ensuring students understand the underlying concepts rather than just copying AI-generated solutions.

Our current findings and progress raise several interesting research questions for future investigation. To what extent does explicit paradigm classification and fine-tuning on specialized datasets quantitatively improve the accuracy of bug-fixing for complex errors compared to general-purpose LLMs? How transferable are the fine-tuning strategies and compiler insights across different AI agent architectures (SWE-Agent, Open Hands, custom agents)? Can we develop a more universal framework for agent enhancement? Beyond traditional metrics, how can we better quantify the "effectiveness" of an AI agent in assisting a human programmer in real-world development tasks?

We believe that parameter count is a very strong factor for performance. While we tested models varying from 8 to 14B parameters, for a more effective agent we believe that exploring 24–72B parameter models is a next step worth pursuing. This would involve access to enterprise-grade hardware and multi-GPU solutions. Dedicating more time to tinker and find the optimal configuration files for each agent would also yield better results, since some models were highly performant on some agents, but failed to submit results on others. The next steps would be to increase model size, fine-tune exclusively with LoRA adapters, and provide better configurations to offer better and more consistent performance in all scenarios.

References

Daniel Fried et al. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

René Just et al. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of ISSTA*, 2014.

Shuai Lu et al. Codexglue: A benchmark dataset and open challenge for code intelligence. In *NeurIPS*, 2021.

Yue Wang et al. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Mingyu Xie et al. Swe-fixer: A modular llm pipeline for automatic bug repair. *To appear in ICSE*, 2025.

Kevin Yang and et al. Swe-agent: Large language models for software engineering. *arXiv preprint arXiv:2401.10010*, 2024.