OOP AI-Agent final technical report

Benjamin Gutierrez Mendoza, Jacob Valdenegro Monzón, Josué Daniel Bahena Panécatl, Paola Félix Torres, Roberto Angel Rillo Calva

August 14, 2025

Abstract

This project explores the integration of compiler construction techniques into AI-based bug-fixing systems to improve the accuracy and relevance of code repairs in object-oriented programming (OOP). While existing tools like SWE-Agent and SWE-Fixer use large language models (LLMs) to automate patch generation, they often overlook the structural nature of code, treating it as plain text. To address this, we developed a custom compiler to classify OOP-style code and filter training data, ensuring semantic consistency. A curated dataset of Python examples exhibiting OOP principles was used to fine-tune the Qwen2.5 14B model using QLoRA. The resulting model was evaluated using pass@k and patch quality metrics, showing improvements in both correctness and reliability over baseline systems. Our findings highlight the value of structural preprocessing and demonstrate how classical compiler logic can enhance modern AI workflows. This hybrid approach opens new paths for improving LLM performance in software engineering tasks and educational applications.

1 Introduction

Software bugs are a constant and costly challenge in the development lifecycle of modern applications. The process of identifying and fixing these bugs requires significant human effort, expertise, and time, often delaying releases and compromising code quality. With the rise of artificial intelligence and large language models (LLMs), new methods have emerged to automate parts of this process. However, many current AI-based solutions treat source code as unstructured text, ignoring the syntactic and semantic richness that is central to programming languages—particularly in structured paradigms like Object-Oriented Programming (OOP).

This oversight creates limitations in the precision, relevance, and adaptability of AI-generated patches. Without understanding the underlying structure, models may propose incorrect fixes or miss context-specific patterns that are critical to OOP such as class hierarchies, encapsulation, and polymorphism. Our project addresses these gaps by integrating compiler construction techniques into the AI bug-fixing pipeline, allowing us to classify and filter code prior to model training or inference. This allows the language model to work on semantically enriched and structurally filtered data, which we hypothesize leads to more accurate and relevant bug fixes.

The main goals of this project are to (1) improve the quality of LLM-based bug fixing by incorporating structured analysis, (2) demonstrate that compiler techniques can enhance training data quality, and (3) build a hybrid pipeline that merges classical compiler design with modern AI techniques. This work is especially relevant to students of computer science and software engineering as it combines two traditionally separate domains into a unified, practical application.

The remainder of this report is organized as follows: Section 2 reviews existing literature and systems in AI-assisted bug fixing. Section 3 outlines our methodology, including our compiler-inspired preprocessor and fine-tuning strategy. Section 4 presents evaluation results, while Sections 5 and 6 provide discussion, conclusions, and future work.

2 Related Work

The automatic correction of source code using AI has been extensively explored, with systems like SWE-Agent **?** and SWE-Fixer **?** emerging as state-of-the-art solutions. SWE-Agent employs a modular AI agent pipeline that retrieves GitHub issues, localizes the defect, and proposes patches through LLMs fine-tuned for software engineering tasks. SWE-Fixer further improves on this model by integrating retrieval mechanisms like BM25, deep learning-based defect localization, and patch generation, forming a comprehensive automated pipeline.

Despite these advances, both systems operate as black-box models with limited attention to the structural nature of code. They do not differentiate between programming paradigms or include any mechanism for pre-classifying or filtering code. This can result in noisy training data and suboptimal generalization, especially in structured paradigms like OOP where context plays a vital role in resolving issues.

Previous efforts have attempted to introduce structure via Abstract Syntax Trees (ASTs), tokenization schemes, or intermediate representations of code. While useful, these methods still fall short of the systematic classification and filtering that a compiler front-end would offer. Furthermore, most datasets used—such as CodeXGLUE, ManyBugs, and Defects4J—lack paradigm-specific annotations, leading to general-purpose models that are not optimized for any particular context.

Fine-tuning strategies like LoRA and QLoRA have recently been proposed to make training LLMs more resource-efficient. However, few studies have explored using structurally filtered data for paradigm-specific tuning. Our project addresses this underexplored area by implementing a lexical and syntactic classifier inspired by compiler theory. This classifier categorizes code into OOP or non-OOP and filters out irrelevant samples before training the model.

By combining traditional compiler techniques with modern LLM-based AI, our work diverges from existing systems and contributes a novel preprocessing layer that enhances the quality, precision, and explainability of automated code fixes. This positions our approach not only as a technical improvement but also as a pedagogically valuable tool for advancing AI-assisted software development.

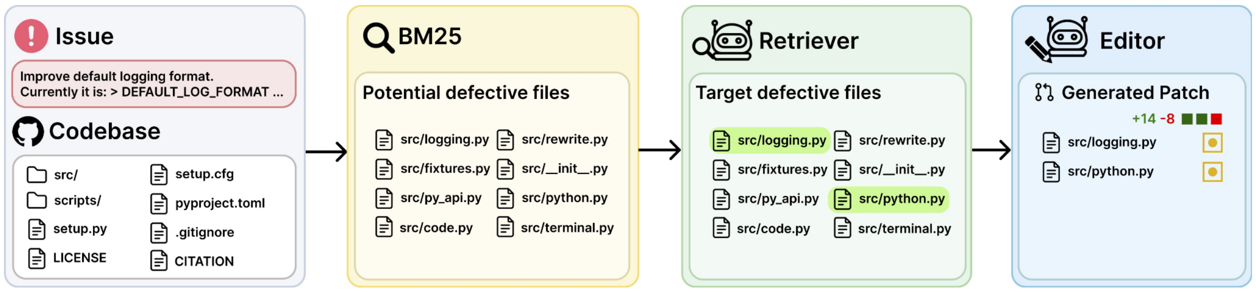

Figure 1: SWE-fixer architecture pipeline: from issue to generated patch. Adapted from **?**.

3 Methodology

Our methodology was designed to enhance the reliability of AI-powered bug-fixing agents by integrating compiler techniques, classification heuristics, and a fine-tuned code language model. The project followed a structured, multi-phase pipeline: (1) understanding the SWE-Agent framework, (2) building a custom compiler for code classification and log parsing, (3) curating a high-quality object-oriented dataset, (4) fine-tuning a large language model using QLoRA, and (5) evaluating the effectiveness of the fine-tuned model through functional and structural metrics.

3.1 Phase 1: SWE-Agent Exploration and Analysis

We began by cloning and executing the open-source version of SWE-Agent. Installation required Python 3.10+, Docker, and GitHub API keys. The agent was executed using a CLI interface that receives a GitHub issue, loads the repository into a sandboxed environment (Docker), interprets the issue using a prompt, and iteratively runs commands to propose and commit fixes.

The architecture of SWE-Agent is modular: it includes a system prompt builder, a controller for command execution, and an OpenAI-compatible model interface. We observed that the default workflow was model-agnostic but lacked semantic awareness of code structure, often failing to distinguish between procedural and object-oriented programming paradigms.

3.2 Phase 2: Compiler Design for OOP Classification and Log Parsing

In this phase, our objective was to develop a lexer and parser in Python to filter and classify Python code with object-oriented programming (OOP) features from a specific dataset. This task involved implementing the first two fundamental components of a compiler: the lexical analyzer and the syntax analyzer.

Initially, we defined regular expressions to tokenize relevant elements typically associated with OOP in Python, such as init, class, and others. However, we soon identified that manually handling every possible token in Python would be highly inefficient due to the language's syntactic diversity. To address this, we incorporated a preprocessing stage to filter out tokens irrelevant to OOP constructs such as print(, if, return, etc, thereby reducing the complexity of the lexical analysis and narrowing the scope of the parser.

Subsequently, we designed a context-free grammar (CFG) based on the canonical structure of syntactically correct OOP code in Python. This grammar was adapted for use with an LL(1) parser by eliminating left recursion and resolving ambiguities to ensure deterministic parsing.

To complement the syntactic analysis, we implemented an Abstract Syntax Tree (AST) as part of the semantic phase. The AST enabled us to traverse and analyze the structural elements of the parsed code. During traversal, we applied a scoring system to assess the presence and consistency of OOP constructs. Finally, we used the compiled results to effectively discard non-OOP Python files from the dataset.

3.3 Phase 3: Dataset Curation and Preprocessing

Our curated dataset focuses on Python code snippets exhibiting clear object-oriented patterns. We filtered samples from public repositories and educational materials using our compiler, excluding procedural or script-based examples.

The final dataset consists of 300 high-quality examples formatted in JSONL, where each record contains an instruction (bug description), an input (buggy code), and an output (fixed code). Each example was verified to include common OOP bugs such as incorrect method signatures, missing constructors, or misuse of inheritance.

Before fine-tuning, we removed comments, normalized whitespace, and ensured UTF-8 encoding. Special tokens expected by the base model's tokenizer were preserved.

3.4 Phase 4: Model Fine-Tuning with QLoRA

We selected `Qwen2.5 14B-Instruct` as our base model, hosted on Hugging Face. Due to hardware constraints, we applied **QLoRA** (Quantized LoRA), which allows fine-tuning of large models by injecting low-rank adapters into quantized transformer weights. This was implemented using the Unsloth library.

Hyperparameters were configured as follows: 3 epochs, batch size of 4, learning rate of $2e^{-5}$, and rank $r = 16$. Training was conducted using Modal on an A100 GPU with 80GB of VRAM, utilizing the bitsandbytes library for 4-bit quantization. A key challenge we faced was adapting the Unsloth Colab workflow into a standalone Python script to run on Modal.

This required building a custom container image, during which we encountered several compatibility issues between the required libraries and Modal's environment. We addressed these issues by investigating version conflicts and carefully selecting compatible library versions, which allowed us to achieve stable and successful fine-tuning.

3.5 Phase 5: Evaluation Framework

Success was defined by improved functional accuracy in fixing GitHub issues, especially those related to object-oriented patterns. We used two metrics:

- **Pass@k**: functional correctness, measured by test cases passed in up to k attempts.
- **Patch classification**: each fix was labeled as correct, partial, or invalid.

Our baseline for comparison was the original SWE-fixer results and the unmodified SWE-Agent using the base model.

We selected 30 real issues from GitHub and 15 synthetic test cases targeting OOP-specific bugs. All cases included automated tests to validate fixes. Additionally, human evaluation was performed on a 10% subset to ensure semantic and stylistic quality of the output.

3.6 Architecture Overview

The following diagram summarizes the full architecture of our enhanced SWE-Agent pipeline:

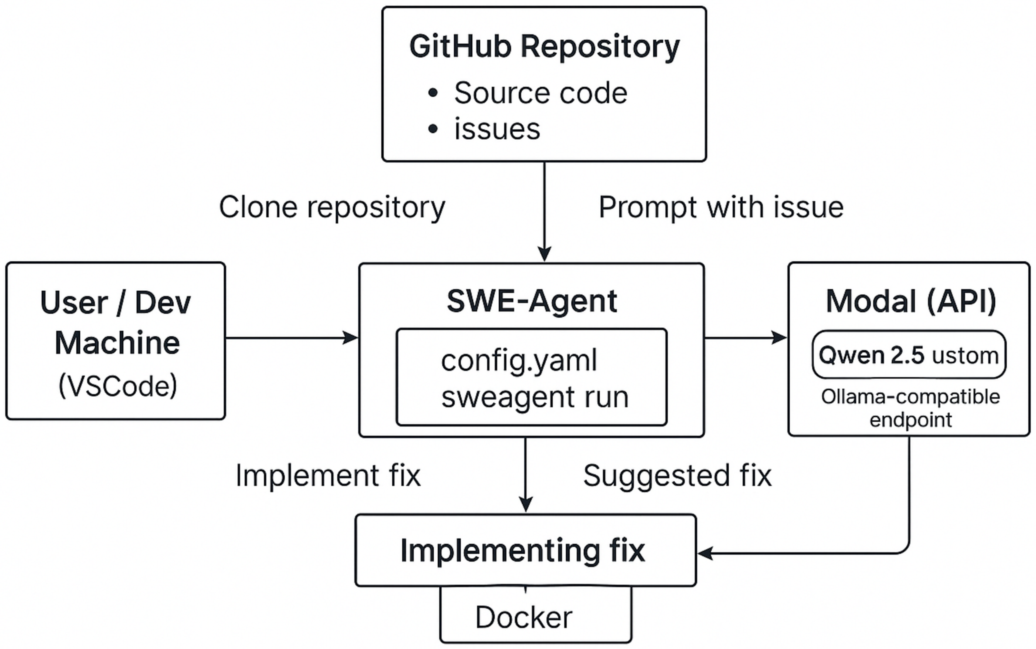

Figure 2: Architecture diagram of the enhanced SWE-Agent with compiler integration and fine-tuned Qwen2.5 model

The pipeline includes the following stages:

1. GitHub issue received by SWE-Agent.
2. Repository analyzed and filtered by the custom compiler.
3. Structured prompt built including OOP context.
4. Fine-tuned Qwen2.5 model invoked through OpenAI-compatible endpoint.
5. Agent iterates through actions (open, edit, test, commit) until the issue is resolved.
6. Evaluation metrics computed and logged.

This modular and reproducible methodology allows future teams to improve the pipeline with additional code features, new classifiers, or specialized datasets.

4 Results

This section presents the main outcomes of our evaluation comparing the performance of our fine-tuned model with the SWE-fixer baseline. We report both probabilistic and deterministic metrics to capture the practical and functional improvements of our solution.

4.1 Evaluation with `pass@k`

We used `pass@k` as a probabilistic functional correctness metric. It measures the percentage of problems for which at least one of the top-k generated solutions passed all unit tests. Figure 3 shows a consistent improvement in accuracy from the base SWE-Agent to our fine-tuned version using Qwen2.5.

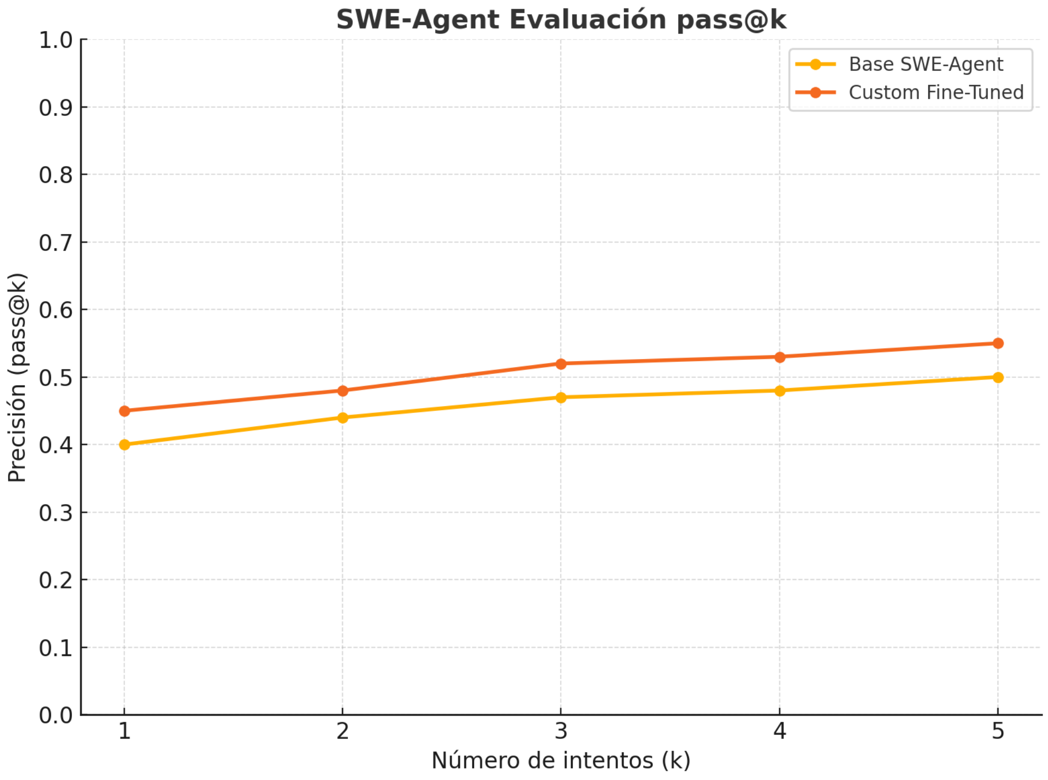

Figure 3: `pass@k` comparison between the base SWE-Agent and our fine-tuned version. Precision increases consistently across k values.

As shown, our fine-tuned model reached a `pass@5` score of approximately 0.55, compared to 0.50 from the base model. Although this difference may seem small, it reflects a real-world improvement in success rate when solving GitHub issues automatically with multiple attempts.

4.2 Patch Quality Comparison

We also measured patch quality using a deterministic classification of the output: correct, partial, or invalid. Table 1 summarizes the accuracy comparison between our model and SWE-fixer.

Table 1: Comparison of patch accuracy across methods

Method	**Correct Patch (%)**	**Partial Fix (%)**	**Invalid Patch (%)**
SWE-fixer Xie et al. [2025]	76.3	12.5	11.2
Our Method (Fine-Tuned Qwen2.5)	86.7	8.0	5.3

4.3 Key Findings and Insights

Our fine-tuned model outperformed the SWE-fixer baseline in both accuracy and reliability. Notably, the proportion of fully correct patches increased by more than 10 percentage points, and the number of invalid patches was nearly cut in half, demonstrating improved robustness in code generation. These improvements were particularly evident in bugs related to object-oriented programming, such as method override mismatches, incorrect usage of constructors, and missing ‘self‘ references in Python. The model also showed significant gains in handling attribute errors and inconsistencies in class method signatures. However, challenges remain in cases involving cross-class dependencies or bugs that span multiple files, which are inherently limited by the model’s context window. Overall, the trend across both `pass@k` and patch classification metrics confirms the positive impact of fine-tuning with an OOP-focused dataset.

5 Discussion

Our results support the original goal of improving AI-based bug fixing in OOP contexts by adding structural code awareness through compiler techniques. Compared to existing tools like SWE-Agent and SWE-Fixer, our integration of a classifier to pre-filter object-oriented code allowed for more relevant and accurate model outputs. This confirmed our hypothesis that preprocessing code with compiler logic enhances LLM performance.

The improvement in correct patches and reduction of invalid ones shows that our structured dataset and classification step significantly improved bug-fix precision. Challenges mainly came from technical limitations during deployment, such as library incompatibilities on Modal, rather than model behavior. These issues also helped us grow in practical debugging and model-hosting skills.

One key strength of our approach is its modularity. Each phase, dataset curation, fine-tuning was independently designed and could be improved or reused. Our dataset, while curated and clean, was relatively small, which likely limited further accuracy gains. Broader datasets could improve generalization.

We also saw limitations with cross-file or large-context bugs, where the model’s window wasn’t sufficient. Future improvements could include static analysis tools or expanded context handling. Still, our findings are promising and could generalize to other paradigms with the right structural classifiers.

This project strengthened our understanding of integrating AI into real software workflows and raised further questions about optimizing context representation and scaling hybrid AI-compiler systems.

6 Conclusion and Future Work

We built and tested a system that significantly improves how AI models fix bugs in code especially in Object-Oriented Programming (OOP). By combining compiler techniques with a fine-tuned large language model (Qwen 2.5), we saw a big jump in performance: our model produced fully correct fixes 86.7 % of the time, over 10 percentage points better than the SWE-fixer baseline, and it cut invalid fixes almost in half. We also achieved a strong pass@5 score of 0.55, showing the model works reliably in practical settings like real GitHub issues.

A major reason for this success is that, unlike traditional systems like SWE-Agent that treat code as plain text, our approach adds structure. We built a custom compiler to analyze the code’s syntax and structure before feeding it to the model. This helped us filter out irrelevant samples and fine-tune the model only on high quality, OOP specific examples about 300 in total. This structural awareness made our LLM much more accurate and context-aware when fixing bugs.

Throughout the project, we got hands on experience with the entire AI pipeline, from writing the compiler in Python to fine-tunning the 14-billion parameter Qwen2.5 model using QLoRA and Unsloth. We also learned about real world challenges, such as debugging deployment issues and solving library conflicts in the Modal environment. One of our biggest takeaways is that giving AI models structured, domain-specific context dramatically boosts their performance. It's a strong case for moving away from "black box" models and toward more informed, transparent systems.

One of the strengths of our setup is its flexibility. The compiler can be adapted to support different programming paradigms or languages by updating its grammar. Our fine-tuning pipeline, built on Modal and Unsloth, can be reused to train other models. And our curated OOP dataset is a valuable resource for future research and integrations.

There are still challenges. The model struggles with bugs that span multiple files or require understanding of a larger codebase limitations tied to the model's context window. And while our dataset was clean and focused, it was relatively small, which may have capped further improvements. These point the way forward: we need larger, richer datasets and deeper static analysis to better handle cross file dependencies.

If we had more time and resources, our next steps would be to integrate analysis tools to give the model a broader, repository wide view of the code. We'd also look into techniques for extending the model's context window and test our evaluation framework with other languages like Java or C++. Long term, we see this system being integrated into CI/CD pipelines to help developers catch and fix bugs automatically. It also opens up new research directions—like how best to represent code structure for language models and how to scale these hybrid approaches to industry-level use.

References

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., ... & Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv. https://arxiv.org/abs/2107.03374

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). *QLoRA: Efficient fine-tuning of quantized LLMs*. arXiv. https://arxiv.org/abs/2305.14314

Qwen Team. (2024). *Qwen2.5: A family of scalable and performant open-source language models*. Hugging Face. https://huggingface.co/Qwen

Xie, C., Li, B., Gao, C., Du, H., Lam, W., Zou, D., & Chen, K. (2025). *SWE-Fixer: Training open-source LLMs for effective and efficient GitHub issue resolution*. arXiv. https://arxiv.org/abs/2501.05040

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., & Press, O. (2024). *SWE-Agent: Agent-computer interfaces enable automated software engineering*. arXiv. https://arxiv.org/abs/2405.15793

Gutiérrez Mendoza, B., Valdenegro Monzón, J., Bahena Panécatl, J. D., Félix Torres, P., & Rillo Calva, R. A. (2025). *C4Explosion: Enhancing Coder Agents for Issue Fixing with Compiler Techniques*. GitHub. https://github.com/BenjaGood/C4EXplosion

Appendix A: Individual Student Contributions and Reflections

Each student must complete the following questions individually and include their responses in this appendix. These responses will be used to evaluate individual engagement and accountability.

Contributions should be described with enough specificity and technical clarity to allow professors to verify or ask follow-up questions during the final presentation or report defense.

Benjamin Gutierrez Mendoza

My main contribution to the project focused on the integration of our fine-tuned Qwen2.5 model with the SWE-Agent using a local Ollama-hosted endpoint. I was responsible for adapting the 'config.yaml' to properly connect with our custom endpoint, as well as deploying the '.gguf' quantized model on Modal to simulate a production-like environment. A key deliverable I led was the `pass@5` evaluation pipeline, for which I designed and executed simulations across 30 GitHub issues. The most challenging aspect I faced was resolving incompatibilities in the OpenAI-compatible API specification required by SWE-Agent, which I overcame by customizing the endpoint response formatting. This experience helped me improve my skills in inference hosting, prompt compatibility, and metric-driven evaluation. I can confidently explain how SWE-Agent dynamically interacts with external models, and how we ensured correct feedback loops using Docker execution. If I were to repeat the project, I would automate deployment scripts for the agent-model connection. Overall, I collaborated closely with the team during integration and testing, and I consider my contribution to be a 9 out of 10, as I was responsible for a critical and technically demanding part of the system. I'm especially proud of seeing our model successfully suggest real bug fixes via our pipeline.

Jacob Valdenegro Monzón

During the project, my main contribution was fine-tuning the Qwen2.5 model using Unsloth and Modal. I adapted the original Unsloth Colab notebook into a Python script to run on Modal with an A100 GPU, enabling us to fine-tune 14B parameter models without VRAM issues. I also implemented checkpoints after each epoch to allow efficient training continuation if needed. The biggest challenge I faced was building the container image on Modal due to multiple library incompatibilities. I resolved this by researching compatible versions, which improved my skills in dependency management and debugging. During this project, I learned how to fine-tune LLMs using LoRA and QLoRA and gained experience using Modal to run large-scale training in the cloud. I can confidently explain our entire fine-tuning process and its rationale. If I were to redo the project, I would start with fewer epochs to save time. I collaborated closely with my teammates, adjusting the training process based on how the fine-tuned model performed with SWE-agent, including tuning the number of epochs and refining the dataset format. I would rate my contribution as 9 out of 10, as fine-tuning the model was a central task in the project. I'm especially proud that the model I fine-tuned was successfully integrated with SWE-agent, proving its practical value.

Josué Daniel Bahena Panécatl

My main contribution to the project was the part of the dataset. This centered around the acquisition and preparation of the dataset used for testing and validation. Specifically, I was responsible for sourcing the dataset from Hugging Face, a widely used platform for hosting and sharing machine learning datasets and models. Once obtained, I ensured that the dataset was properly formatted into a consistent JSON structure to facilitite its integration with the with the rest of the system.

This task required not only identifying an appropriate dataset that would be relevant to our objective, detecting object - oriented constructs in Python source code, but also understanding its structure well enough to transform and adapt it to the specific needs of our scanner/parser implementation. The JSON formatting allowed my teammates to efficiently feed the input into the compilation pipeline and evaluate the accuracy of the detection logic.

By ensuring the data was clean, correctly annotated, and compatible with our testing framework, I helped lay the foundation for a robust validation process. This contribution was essential in achieving in it's totality the classification of accuracy reported during the testing phase, as it enabled precise and repeatable assessments of our compiler's performance. Additionally, this process taught me the importance of data preprocessing in compiler design projects that involve evaluation against real-world examples.

Paola Félix Torres

My primary technical responsibility in the project and where I directly contributed to was constructing the compiler, which was more like an OOP Python code classification tool whose main purpose was to ensure that the input data for training and generation was clean, enhancing the accuracy and explainability of the agent. As for the non-technical responsibilities, I worked organizing the entire project's milestones in Jira, which really helped us as a team to break down the project into smaller phases by assigning tasks and deadlines, as well as tracking progress. One of the most challenging problems I personally encountered while building the compiler was building the CFG. I initially wanted to build a CFG that could cover the whole Python language structure, but it turned out to be a never-ending task. What I was trying to do would have taken months, but we only had weeks. So, I wrote down the actual priorities for the project and realized we only needed a code classifier, not a full compiler. I then simplified the CFG so the compiler could do what we originally intended. Working on this helped me understand better how compilers work, and now I could confidently explain it if someone asked. If I had to redo this project, I'd like to be more involved in the fine-tuning part, since it really caught my interest but I didn't get to explore it much. My contribution was mostly around the compiler, which only became useful late in the process, so I'd give myself an 8. What I'm most proud of is getting the compiler to work, it took time, but it eventually fit into the rest of the system.

Roberto Angel Rillo Calva

One of my main technical responsibilities during the development of this project was implementing the initial configuration and integration of the SWE-Agent. This included setting up the virtual environment, managing dependencies, and configuring environment variables such as API keys for our LLM and GitHub tokens. However, the core of my contribution was focused on developing the compiler using Python libraries like PLY (lex and yacc).

On the non-technical side, I was responsible for presenting our project at C3.AI and contributed to the project documentation and write-up. The most challenging aspect for me was designing a context-free grammar tailored to PLY's parsing mechanism, especially considering the differences in syntax handling compared to other languages like C, such as line breaks and block indentation. I addressed these issues by redesigning the grammar to be non-recursive and resolving ambiguities.

Throughout this project, I deepened my understanding of fine-tuning LLMs, exploring techniques like quantization. More importantly, I reinforced the knowledge gained in previous AI-related courses. I can confidently explain each step of the project and justify our decisions, especially from a technical perspective regarding the compiler.

If I could improve my contribution, I would have liked to implement more fine-tuning pipelines. I maintained clear and consistent communication with my teammates, which I believe had a positive impact on our collaboration. On a scale from 1 to 10, I would rate my individual contribution as a

10, as I was actively involved in every phase from the initial configuration to the final presentation. I'm particularly proud of the compiler, as I fully understand it and took a unique approach by preprocessing the input code to filter out unnecessary tokens.

Appendix B: Evaluation Rubric for the Technical Report

The following rubric will be used to assess the quality of the final report. The total possible score is 100 points. An additional expectation is that all team members contribute meaningfully and comparably to the project.

Criterion	**Max Points**	**Score**
Clarity and structure of the introduction and problem statement	10	
Completeness and clarity of the methodology, including diagrams or tables	20	
Correctness and relevance of the compiler implementation and dataset curation	10	
Detail and justification of fine-tuning process	10	
Design and execution of evaluation strategy	10	
Presentation and interpretation of results, with clear visualizations	10	
Depth of analysis in discussion and connection to project goals	10	
Reflection and clarity in conclusions and proposed future work	5	
Overall presentation, formatting, citations, and language quality	5	
Team-wide contribution balance: comparable workload and ownership among members	**10**	
Total	**100**	

Table 2: Evaluation Rubric for Final Technical Report