AI Agents for Task Automation: Fixing Issues

Authors:

Nirjhar Nath
Vardhan Kumar Ray

August 19, 2024

Acknowledgement

We would like to express our sincere gratitude to our mentors, Sudhir and Rishabh, for their guidance and support throughout this internship. Our appreciation also goes to each other as co-authors, for our complementary skills and teamwork that drove this project forward.

Contents

1 Bug Fixing for a Repository

1.1 Abstract

We address the challenge of automatic code debugging within complex and extensive code repositories. This is crucial because software systems in real-world applications are often large, intricate, and prone to various bugs that are time-consuming to identify and resolve. To tackle this issue, we use the powerful abilities of Large Language Models (LLMs) and coordinate multiple specialized agents using CrewAI. Our system employs an architecture where these agents, each assigned distinct roles such as analysis, bug detection, code review, and execution, work in coordination to replicate the systematic approach of a developer. Powered by the llama3-8b-8192 model from Groq and integrated into a user-friendly interface built with Streamlit, these agents efficiently identify and fix bugs while providing clear and actionable feedback to the user. This coordinated effort ensures a seamless debugging process, significantly enhancing the reliability and maintainability of codebases in large-scale software development.

1.2 Introduction

In the realm of software development, debugging is a critical and often labor-intensive task. As software systems grow in complexity, the challenges associated with debugging become increasingly pronounced. Large codebases can harbor numerous subtle bugs that are difficult to identify and rectify manually. The traditional debugging process typically involves developers manually inspecting code, which can be time-consuming and error-prone. This method relies heavily on the developer's familiarity with the codebase and can often lead to inefficient debugging cycles.

The increasing complexity of modern software systems necessitates the development of advanced tools that can enhance the efficiency and accuracy of the debugging process. Automatic debugging tools are designed to address these challenges by leveraging automated methods to identify, analyze, and correct bugs. By integrating these tools into the software development workflow, it is possible to significantly reduce the time and effort required for debugging while improving the overall quality of the code.

1.3 Problem Statement

The primary objectives of our system are as follows:

1. **Identification and Localization of Bugs**: The system is designed to pinpoint buggy files within a code repository based on a given issue. This involves analyzing the repository and identifying the specific files or components that are likely to contain bugs.

2. **Automated Bug Fixing**: Once the problematic files are identified, the system generates corrected versions of these files. The goal is to automate the process of suggesting and applying code fixes, reducing the need for manual intervention.

3. **Validation of Bug Fixes**: After applying the proposed fixes, the system verifies whether the changes have successfully resolved the identified bugs. This validation step is crucial to ensure that the automated fixes are effective and do not introduce new issues.

1.4 Proposed Pipeline

The proposed pipeline for automatic code debugging leverages a systematic approach inspired by how developers typically diagnose and fix bugs. This pipeline integrates multiple specialized agents powered by Large Language Models (LLMs) and is designed to efficiently identify, diagnose, and resolve bugs within large code repositories. The key steps in the pipeline are outlined below:

1. **Issue Description Analysis**: The pipeline begins by analyzing the issue or bug description provided by the user. If the description is unclear or lacks sufficient detail, the system utilizes the LLM to rewrite the description into more actionable steps, often seeking clarification from the user. This step ensures that the system fully understands the nature of the bug before proceeding.

2. **Dependency Mapping**: The system next identifies dependencies and relationships between different files within the codebase. This is crucial for understanding how changes in one part of the code might affect others. The dependency information is typically stored in a JSON file, which the system uses to trace connections between different modules or files.

3. **Buggy File Identification**: Utilizing the LLM's analysis capabilities, the system scans through the codebase to identify files that are likely to be buggy based on the issue description. The system flags files that require fixes, distinguishing them from those that are not related to the bug.

4. **Code Fix Generation**: For the identified buggy files, the system generates code fixes. This involves suggesting modifications that address the root cause of the bug, often incorporating best practices and ensuring compatibility with the rest of the codebase. The fixes are generated in a manner that mimics the thought process of an experienced developer.

5. **Review and Validation**: After generating fixes, the system reviews the changes to ensure they align with the intended bug fix and do not introduce new issues. The validation process includes running tests or executing the fixed code to confirm that the bug has been resolved.

6. **Manual Review and Feedback**: Finally, the system presents the fixed code to the user for manual review. The user can either accept the fix or provide feedback for further refinement. If the user is not satisfied with the fix, the system can iterate through the process, refining its approach based on the feedback provided.

To achieve this pipeline, we employ a set of specialized agents, each equipped with Large Language Models (LLMs), which collaboratively handle different aspects of the debugging process. Figure 1.1 illustrates the overall workflow of these agents:

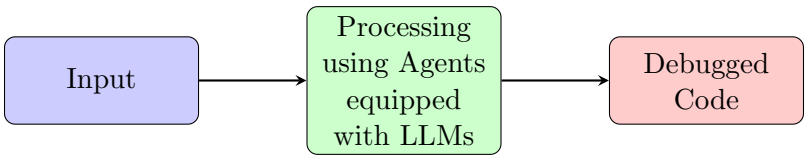

Figure 1.1: Overview of the LLM embedded pipeline

2 LLM Agents for Automation

2.1 LLM Agents

Large Language Model (LLM) Agents represent a significant advancement in artificial intelligence, designed to interpret and generate human-like text. These sophisticated systems utilize large-scale pre-trained language models, such as GPT-4, to perform a broad range of tasks including understanding natural language inputs, generating coherent and contextually relevant responses, and solving complex problems. LLM Agents are particularly valuable in automating and enhancing processes by analyzing user inputs, processing the information based on extensive training, and refining outputs through iterative improvements. The workflow involving LLM Agents typically begins with receiving an input, followed by processing the input with the language model, and concludes with producing a refined output. This process is often repeated until the output meets the desired criteria.

2.2 CrewAI

CrewAI is a universal multi agent framework that allows for all agents to work together to automate tasks and solve problems. It utilizes the capabilities of LLM Agents to facilitate communication, manage project workflows, and assist with decision-making processes. By combining the advanced text generation and comprehension abilities of LLM Agents with CrewAI's collaborative tools, users can streamline their operations, improve productivity, and foster effective teamwork. To implement our proposed pipeline, we will utilize several agents working in coordination to address and fix bugs effectively. This coordination can be seamlessly achieved through CrewAI, which streamlines the interaction between agents, ensures efficient workflow management, and supports collaborative problem-solving.

2.3 Implementation

The discussed pipeline is implemented through a coordinated system of specialized agents, each playing a distinct role in the debugging process. The system begins by taking in three essential inputs:

- The code repository in ZIP format.
- A detailed description of the issue that the user is encountering, i.e., the bug description.
- A JSON file containing dependency information about the files within the repository.

Before diving into bug identification, the system generates the dependency JSON file. This file is crucial as it maps out how different files within the codebase are interrelated, typically based on import statements. A script is executed across the code repository to analyze these dependencies, allowing the system to understand which files rely on others. This mapping is later used to trace the impact of changes and identify interconnected issues.

Once the dependencies are mapped out, the next phase involves identifying the buggy files. For this, two primary agents are employed:

1. **Analyzer:** The Analyzer Agent starts by examining the issue description in conjunction with the code repository. Its goal is to identify files that are likely to contain bugs. It does this by analyzing the code and checking for inconsistencies, potential logical errors, and any issues that align with the provided description. The output of this analysis is a

JSON file where each file in the repository is tagged with either 'Yes' (indicating it likely contains a bug) or 'No' (indicating it likely does not).

2. **Reviewer:** The Reviewer Agent then reviews the analysis provided by the Analyzer. It acts as a secondary check to confirm the findings and ensure no critical files are missed. The Reviewer Agent may also provide additional insights or suggest rechecking certain areas based on its own assessments.

Once the buggy files have been identified, the process of fixing these bugs begins. This involves a series of agents working sequentially:

1. **Analyzer**: Revisits the files flagged as buggy and generates a detailed analysis report, identifying specific issues such as syntactical errors, logical flaws, or runtime exceptions. This report serves as a blueprint for the next agent.

2. **Fixer**: The Fixer Agent takes the analysis report and applies necessary code fixes. These fixes are generated with consideration for best practices, code efficiency, and compatibility with the rest of the codebase. The Fixer Agent also annotates the changes with comments, thereby providing context and explanations for each modification.

3. **Reviewer**: The reviewer agent ensures that the code adheres to best practices and standards. It reviews the code for readability and provides feedback to ensure it is of high quality.

4. **Runner**: The runner agent executes the updated code to verify if the bugs have been resolved and to check for any new issues. It reports the outcome and confirms if the expected results are achieved.

Figure 2.1 illustrates how agents interact to process the input repository and issue description, ultimately producing debugged code.

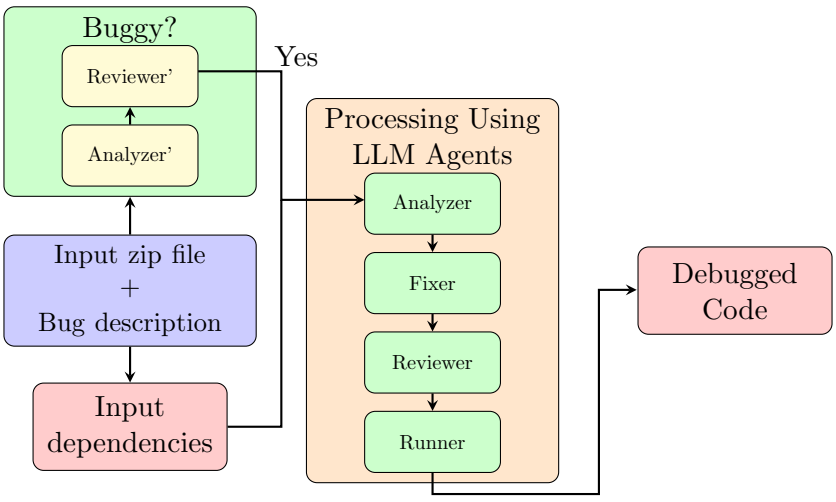

Figure 2.1: Debugging process using LLM agents.

3 Streamlit UI

When the user interacts with the interface, they are required to provide essential inputs that the system requires to analyze and fix the code. These inputs include a detailed description of the issue or bug, which serves as the foundation for the system's subsequent analysis. Additionally, the user is asked to upload the entire code repository in a ZIP format.

Once the user has provided all the necessary inputs, they can initiate the debugging process by clicking the 'Analyze and Fix' button. This triggers the system to process the provided data using the specialized agents powered by LLMs. The system then outputs a list of files that are likely to contain bugs based on the analysis. For each identified buggy file, the system not only generates a corrected version of the code but also provides detailed explanations for the changes made. These explanations help the user understand the reasoning behind the fixes. Figure 3.1 shows how the Streamlit UI looks like.

Figure 3.1: Streamlit UI showcasing the debugging interface

4 Conclusion and Further Improvements

In this report, we have presented a comprehensive approach to automating the debugging process in large code repositories using a pipeline powered by LLMs and coordinated through CrewAI. The system is designed to identify and localize bugs, apply automated fixes, and validate these changes to ensure that the bugs are resolved without introducing new issues. By integrating specialized agents that mimic the systematic approach of experienced developers, we provide a possible solution that could tackle the problem of bug fixing for a big repository.

We acknowledge that the testing aspect of our system has not been thoroughly explored. Rigorous testing is crucial to further validate the fixes and ensure the robustness of the system in diverse and complex scenarios. Enhanced testing mechanisms will allow us to more thoroughly validate the automated fixes. This will help ensure that the code changes do not introduce new bugs and that the overall system behaves as expected.

Also, implementing a feedback loop where user input and test results are used to continuously refine the LLMs and the debugging agents could significantly enhance the system's performance. This iterative approach would help adapt the system to various codebases and issue types, ensuring it remains effective across different contexts.

One more area where further work can be done is the scalability and performance optimization of the model. Further work can be done to ensure that the system remains efficient and responsive, even when dealing with extensive repositories.

Appendix

Tools Used: CrewAI, llama3-8b-8192 model from Groq, Streamlit.

References

1. OpenAI's GPT: https://chat.openai.com/
2. llama-3: https://groq.com/introducing-llama-3-groq-tool-use-models/
3. CrewAI: https://www.crewai.com/
4. Prompt Engineering: https://www.promptingguide.ai/