GLOBAL RESEARCH IMMERSION PROGRAM
FOR YOUNG SCIENTISTS

Exploring Agentless: Demystifying LLM-based Software Engineering Agents

Agentless Methods and Bug Sets Analysis

Yaoyu Chen & Junjie Jiang
University of Illinois Urbana-Champaign & University of Sheffield
`Junjiejiang203@gmail.com`
`Ericfengbai@gmail.com`

Abstract
Large language models (LLMs) like GPT-4 and Claude-3.5 excel in generating code snippets but struggle with repository-level tasks requiring comprehensive codebase understanding. The SWE-bench benchmark, particularly its Lite subset, evaluates tools on real-world bug fixing. Inspired by the Devin AI Software Engineer, many agent-based approaches have been developed but face issues with tool complexity and decision-making. We propose AGENTLESS, a simpler two-phase process for localization and repair. AGENTLESS outperforms SWE-agent[2] in both localization and repair, especially on complex tasks, as shown on SWE-bench Lite. This approach suggests a more effective, less complex path for autonomous software development.

Introduction

Large language models (LLMs) such as GPT-4 and Claude-3.5 have shown significant capabilities in generating code snippets based on user descriptions. However, their application to repository-level software engineering tasks remains underexplored. These tasks, including feature addition, program repair, and test generation, require an understanding of dependencies across an entire codebase. The SWE-bench benchmark addresses this gap by providing real-world GitHub issues and corresponding repositories for evaluation, with the recently released SWE-bench Lite subset focusing on bug fixing.

Inspired by the Devin AI Software Engineer, many studies have developed agent-based approaches, enabling LLMs to autonomously perform actions, observe feedback, and plan steps. However, these approaches face limitations in tool complexity, decision-making control, and self-reflection capabilities.

To address these issues, we propose AGENTLESS, which follows a straightforward two-phase process: localization and repair. In the localization phase, AGENTLESS identifies fault locations, and in the repair phase, it generates and selects the best patch. Our analysis of AGENTLESS's performance on GitHub compared to SWE-agent shows that AGENTLESS excels in both localization and repair. AGENTLESS successfully resolved several issues, while SWE-agent's weaker localization led to poorer repair performance. Overall, AGENTLESS demonstrated superior results, especially in complex tasks, on the SWE-bench Lite benchmark.

Given the limitations of current LLMs, complex agent designs are not always the best solution for repository-level software engineering tasks. Instead, a simpler, distributed approach like AGENTLESS can be more effective. By leveraging the more controllable aspects of LLMs and incorporating human intervention, AGENTLESS enhances efficiency and effectiveness, avoiding the pitfalls of fully autonomous agents. This approach offers a promising direction for future autonomous software development.

For the exploration of Agentless, this will be divided into two steps, since Agentless is composed of two approaches: localization as well as repair. Regarding localization we will analyze the code's localization approach, while for the repair part we use the methodology of comparing the characteristics of the SWE-agent generated by the agent-based code with the characteristics of the bugs repaired by Agentless, in order to analyze the characteristics of Agentless in terms of repairing bugs.

Background

A. Terminology.

Software debugging involves identifying and fixing issues in source code, encompassing processes like Fault Localization (FL) and Automated Program Repair (APR). FL aims to pinpoint buggy elements within a program using static or dynamic analysis, resulting in a ranked list of suspicious code elements. APR utilizes this list to generate and verify patches, ensuring plausible patches pass all test cases and manual verification by developers.

Unified debugging, a pioneering approach, integrates FL and APR by leveraging repair information to enhance FL accuracy. This method emphasizes the interconnection between FL and APR, which aligns with our approach that offers an end-to-end solution where FL and APR interact without being deterministic. This architecture enables the modification of code elements beyond those localized by FL, with FL results being adjustable based on repairs.

Rubber duck debugging, or rubber ducking, is a method where developers explain their code aloud to identify gaps between expectations and implementation. While traditional rubber ducking involves line-by-line explanation, breaking down the code and articulating it in natural language can be beneficial. .

B. Large Language Models

Large Language Models (LLMs) have made significant strides in natural language processing, including text generation, conversational engagement, and logical reasoning. LLMs predict tokens auto-regressively within a textual context, enabling unsupervised training on massive text corpora. Code LLMs, specifically trained on code-related data, excel in code generation tasks. For instance, DeepSeek-Coder is trained on a vast dataset from platforms like GitHub and StackExchange.

LLMs operate using prompts, which are instructions that guide the LLM to perform tasks until encountering a stop word or reaching a word limit. Prompt engineering allows researchers to utilize LLM capabilities for various tasks without retraining. In this work, we use GPT-4 for automated debugging through prompt engineering, leveraging its advanced understanding of both natural languages and code.

C. Agent-Based Approaches

With the advancement of LLMs, agent-based approaches have emerged to tackle complex software development tasks. These agents are equipped with tools to run commands, observe feedback, and plan future actions autonomously. Examples include the Devin AI Software Engineer and AutoCodeRover, which enable LLMs to perform tasks iteratively, such as editing files, running tests, and executing shell commands. Each action taken by an agent is based on previous actions and feedback from the environment.

However, the complexity of these approaches presents challenges. The usage and design of tools require careful abstraction and API design, which can lead to incorrect or imprecise tool usage. Moreover, agents often lack control in decision planning and self-reflection, making them prone to suboptimal decisions and difficulties in debugging.

In this poster, we propose an alternative to the complex agent-based methods: AGENTLESS. AGENTLESS employs a simplified two-phase process of localization and repair, avoiding the need for agents to make autonomous decisions or use complex tools. This approach not only simplifies the design but also enhances the interpretability and effectiveness of automated software development.

Methods

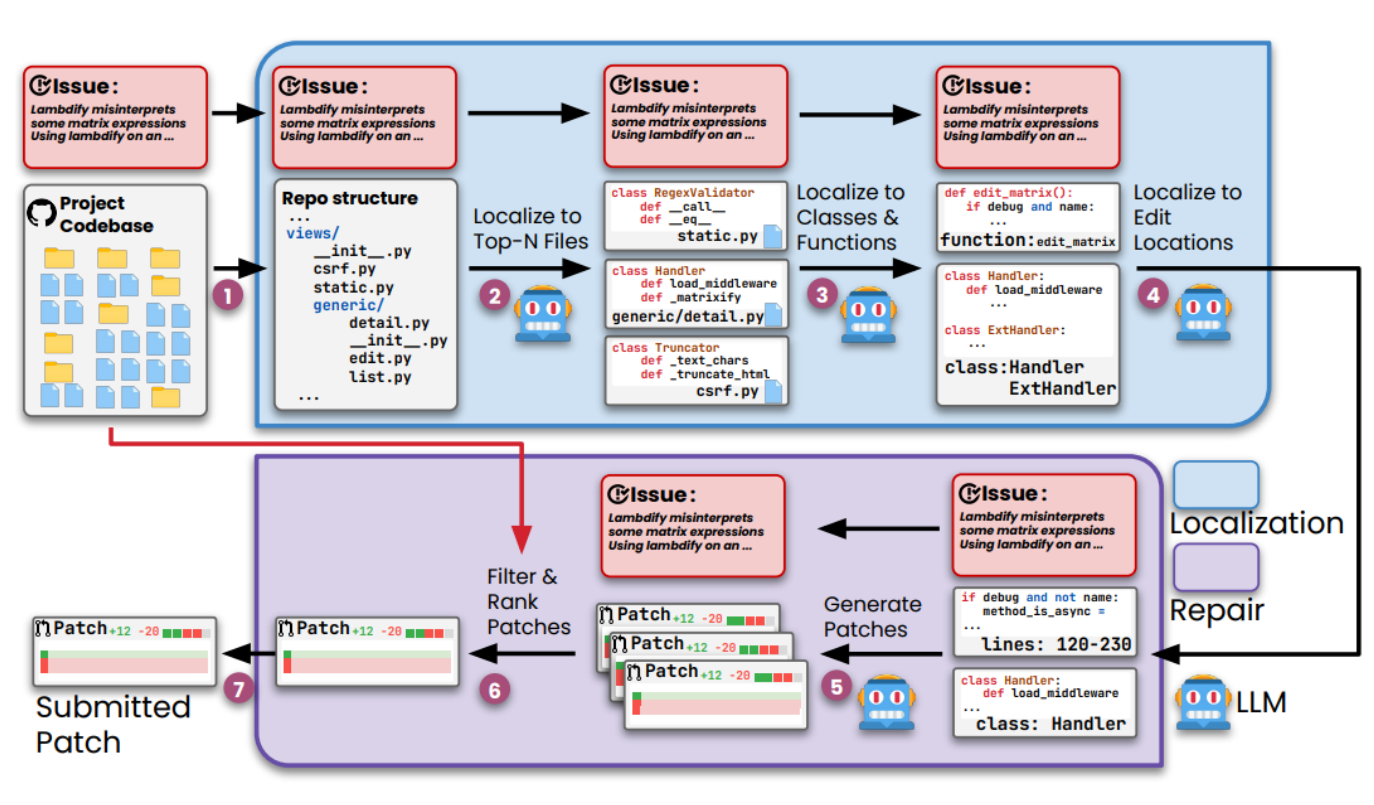

Figure 1: Overview of AGENTLESS[1]

Figure 1 shows the workflow on Agentless, and we start by understanding the workflow. Due to the open source nature of the project, we use the code and data provided by it for high reductivity. For the exploration of Agentless' localization features, we read the code provided in github and the output of the code run to analyze its localization features. For the exploration of Agentless repair function, we compare it with another SWE-agent with agent code generation repair, mainly to analyze the effect of the two for code repair and the characteristics of the comparison.

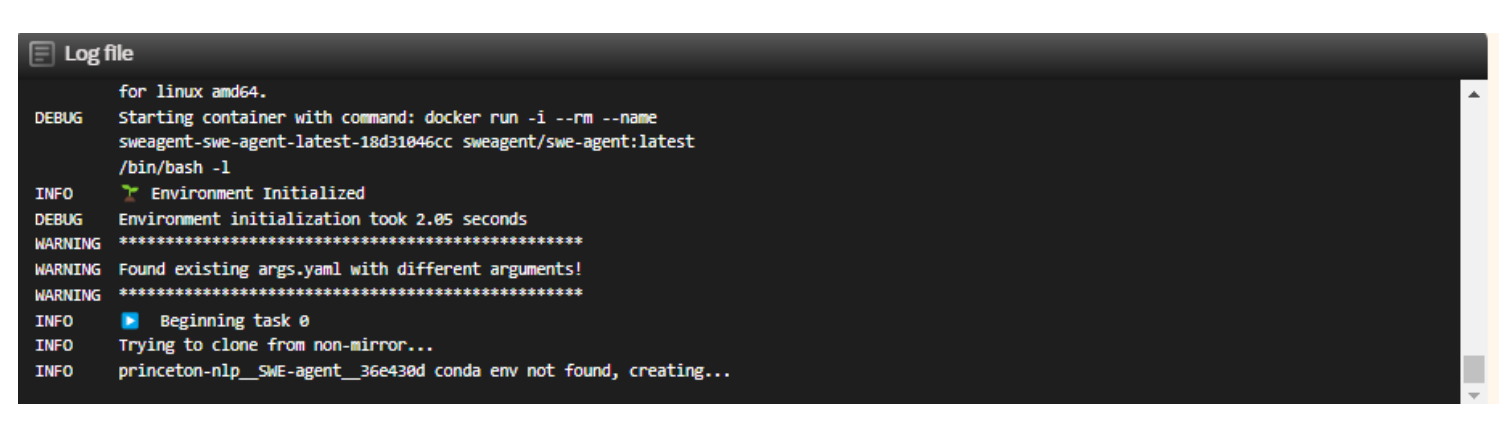
```
Log file
        for linux amd64.
DEBUG   Starting container with command: docker run -i --rm --name
        sweagent-swe-agent-latest-18d31046cc sweagent/swe-agent:latest
        /bin/bash -l
INFO    Environment Initialized
DEBUG   Environment initialization took 2.05 seconds
WARNING ***********************************************
WARNING Found existing args.yaml with different arguments!
WARNING ***********************************************
INFO    Beginning task 0
INFO    Trying to clone from non-mirror...
INFO    princeton-nlp__SWE-agent__36e430d conda env not found, creating...
```

Figure 2: logfile of SWE-agent

Results

For the localization part, SWE-agent's localization is weaker than that of Agentless, which is basically able to accurately locate the location of the bug. For the fixing part, using the same bug situation in the dataset, we sampled some Fail to pass samples, which Agentless was able to solve and left it to SWE-agent to fix the code, SWE-agent was not able to accurately and correctly change the incorrect code, usually because SWE-agent failed to locate it accurately. At the same time, we also found some problems on github for you to modify, for simple problems both of them can be solved, and there are also some cases with strong overall logic and large amount of code, some of which can not be handled by Agentless, but the overall result is better. Figure 3 shows how we handled integrating Agentless to fix bugs in the SWE-bench-lite dataset.

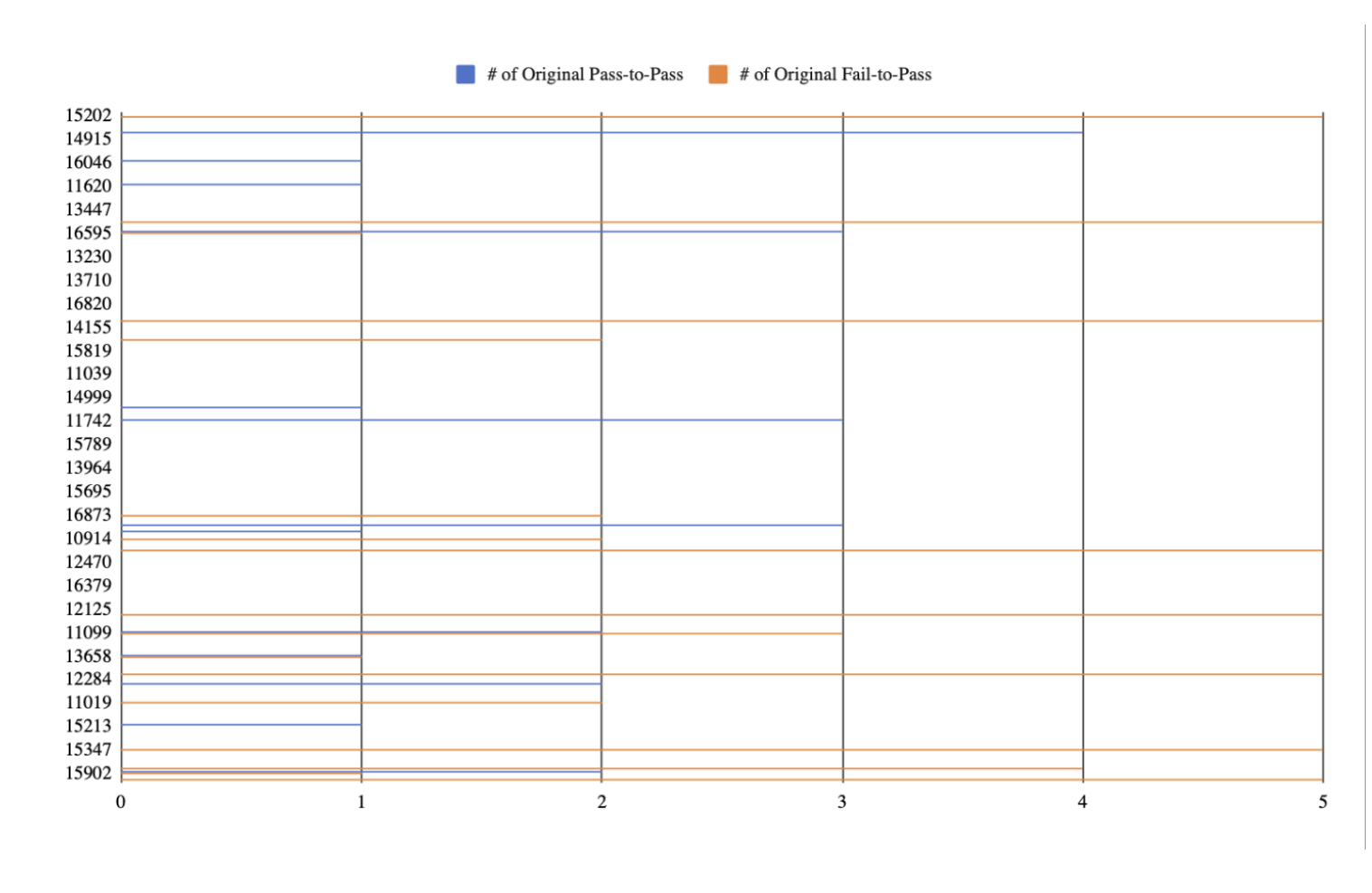

Figure 3: results of Agentless

Conclusions

Due to the current limitations of the large language model, complex agent design is not a good solution to the current an entire repository-level software engineering task; instead, it is better to use a distributed solution without using overly complex agents. The uncontrollability and robustness of the large language model still assists in the development of large-scale software. The idea provided by Agentless is to utilize the more controllable part of the large language model, and then through human intervention, in order to better improve the effect and efficiency, it should be improved step by step, rather than blindly using more complex and fully automated agents.

References

[1] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. Submitted on 1 Jul 2024.

[2] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. Submitted on 6 May 2024 (v1), last revised 30 May 2024 (this version, v2).