Improving OpenDevin: Boosting Code Generation LLM through Advanced Memory Management

Runyu He[1*], Anyu Ying[1], and Xiaoyu Hu[1]

[1]Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213

runyuh@andrew.cmu.edu

July 6, 2024

Abstract

OpenDevin, a code generation AI tool, has emerged as a powerful assistant for both technical and non-technical users, offering a practical approach to coding challenges. Unlike traditional code generators that merely output code, OpenDevin excels by executing code directly in a console, allowing for immediate testing and verification. This functionality not only streamlines the coding process but also enhances learning and troubleshooting, making it accessible to a broader audience. In this project, we address several key challenges to improve OpenDevin's effectiveness, especially in handling multi-round conversations and contextually relevant code generation.

Our team identified and tackled two main challenges faced by OpenDevin: variety of input, and multi-step conversations. Through incorporating a series of functions to parse, summarize, and organize LLM agent's memory logs, we significantly improved OpenDevin agent's capabilities among a variety of tasks. The integration of efficient memory management led to a notable increase in accuracy—from 44.4% to 88.9% —in multi-round conversations, highlighting the importance of effective memory management in AI-powered coding tools.

This report details our methodology, the challenges we faced, and the solutions we implemented, showcasing OpenDevin's potential to revolutionize the way users from various backgrounds engage with coding tasks.

Keywords

Code Generation, Large Language Models, Memory Management, AI Copilot, Multi-Round Conversations

Code and Resources

The source code for this project is available at ImprovingOpenDevin: An Experiment In Improving Domain Specific LLM Agent Through Effective Memory Management.

1 Introduction

As the development of large language models progresses, code generation has emerged as a notable trend. Recently, several high-quality code generation language models have been introduced. DeepSeek-Coder [1] excels in standard programming tasks, surpassing all existing open-source code LLMs across various benchmarks. StarCoder 2 [2], another model excelling in code generation, offers performance on par with its contemporaries and showing particular strength in data science-related tasks.

Embedded with evolving code generation models, various coding agents address a broad spectrum of coding scenarios. For instance, GitHub Copilot [3] integrates directly within IDEs like VSCode to provide code completion, thereby enhancing developer productivity. Similarly, the SWE-agent [4] specializes in solving software engineering problems, particularly those found on GitHub, and outperforms pure standard models like GPT-4 and Claude 3 on SWE-bench benchmark. Frameworks like Devin [5] and OpenDevin [6] are designed to function akin to software engineers, efficiently tackling programming issues.

As these agents evolve, understanding their technological foundations, the challenges they face, and their impact on software development becomes crucial. In this work, we explored the infrastructure of OpenDevin, identify some of its issues, and propose solutions.

OpenDevin is an advanced framework that leverages code generation models to tackle comprehensive software engineering problems. This system features a conversational user interface, utilizes large language models, and incorporates a robust frontend and backend architecture. Designed to handle full software development projects from planning to execution, OpenDevin goes beyond the capabilities of a typical coding assistant. A distinctive feature of OpenDevin is its ability to execute code directly through its user interface, providing immediate feedback to streamline learning and debugging processes. It harnesses large language models tailored with specific prompts to effectively address complex software issues. Additionally, OpenDevin is equipped with a suite of tools including internet browsing capabilities, further enhancing its functionality for comprehensive software engineering tasks.

However, OpenDevin, still under development, performs well for individual user requests but struggles with sequences of requests. Our analysis identified key issues in memory management affecting its performance. To address these challenges, we developed a test dataset and pipeline to serve as benchmarks for OpenDevin's ongoing development. This dataset focuses on evaluating the tool's accuracy across different coding challenges and its capacity for handling multi-round interactions. Based on this dataset, we improved OpenDevin's memory management by implementing a summarizer for relevant historical information, an indicator to differentiate tasks, and a classifier to distinguish between types of tasks. These enhancements are crucial for maintaining context awareness throughout multi-round interactions, significantly improving user experience by allowing the LLM to provide continuous, coherent assistance akin to a human collaborator.

In the following sections, this report will discuss: the infrastructure of OpenDevin, the specially designed test dataset and associated test pipeline, enhancements made to the monologue agent within OpenDevin, and the experimental results and analysis based on this dataset.

2 Background

2.1 OpenDevin Architecture

In this section, we first analyze the infrastructure of OpenDevin [6]. As identified in the graph, OpenDevin's message ingestion and generation process involves several key stages and components. User input, such as requests to "Write a python file to convert temperature from Celsius to Fahrenheit", are passed through the system's server. The agent controller oversees the operational loop of the agent, managing critical scenarios, such as preventing dead loops, and ensuring smooth transitions between stages. All past actions and observations are stored within the monologue, a dedicated memory container that preserves the context of each interaction. OpenDevin is compatible with various LLMs; users specify their choice of LLM and provide the corresponding API key to tailor the processing capabilities to specific tasks. The agent, leveraging historical data stored in the monologue, plans and executes tasks step by step, utilizing the accumulated knowledge and strategic plans formulated in previous steps. This infrastructure enables OpenDevin to handle complex software engineering tasks efficiently, providing immediate and context-aware responses to user requests.

2.2 OpenDevin Evaluations

In the development of OpenDevin, SWE-bench [4] serves as a critical roadmap. SWE-bench is an evaluation framework designed to assess the capabilities of language models in software engineering,

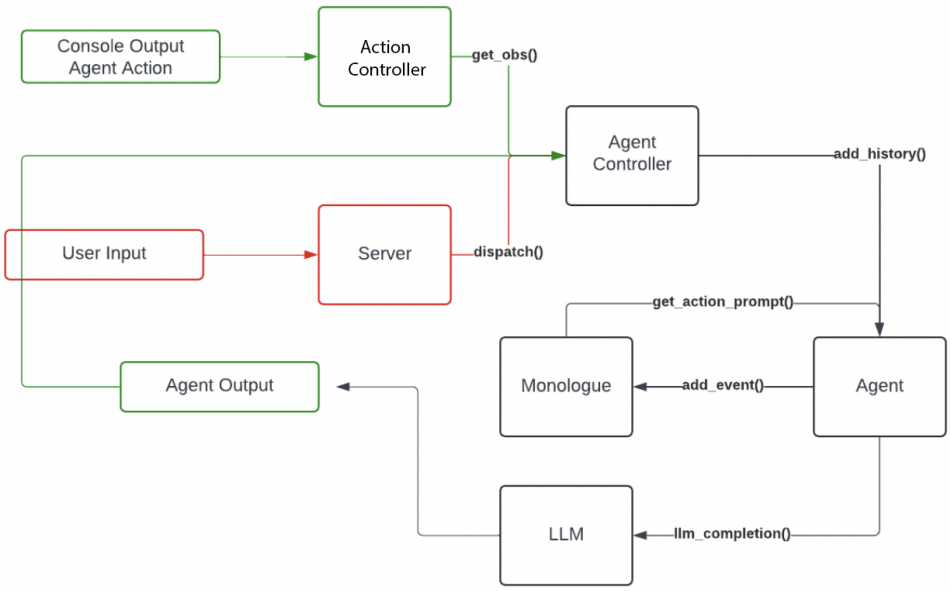

Figure 1: Infrastructure of the OpenDevin framework.

featuring 2,294 real-world problems from GitHub issues and pull requests across 12 popular Python repositories. This framework challenges language models to make complex edits across multiple code structures, testing their reasoning abilities and interaction with code execution environments. Despite its rigorous demands, current models, including advanced ones like SWE-Llama, show limited success. This underscores SWE-bench's role as an essential benchmark for developing more sophisticated coding agents.

While SWE-bench is a robust benchmark for developing AI software engineers like OpenDevin, it is not without limitations. In real-world scenarios involving complex software design, there should be continuous interaction between the agent and the user, which the benchmark does not currently accommodate. These interactions allow the user to clarify commands, highlight errors, and contribute new ideas based on previous progress. Moreover, the benchmark focuses on highly technical software queries, yet in practice, agents will encounter a diverse range of users, including those without coding experience. Recognizing these gaps, we propose our own dataset in the following section to further the development of OpenDevin.

3 Design

3.1 Testing Data

To supplement the ineffectiveness in SWE-bench [4], we have curated a new test set with four categories of questions:

1. Variant of coding request
2. Variant of non-coding request
3. Unrelated question series
4. Related question series

These categories are designed to rigorously test the OpenDevin agent [6] in a more realistic setting, encompassing tasks frequently posed by both programmers and non-programmers across both development and commercial environments.

Additionally, these question types highlight areas where the original OpenDevin model exhibited deficiencies during our preliminary experiments. For instance, when presented with coding requests in the

form of emails or product reports, the original model often lost focus on the core request. Similarly, when confronted with non-coding requests such as "write a calculator.py and write a calculator-user-guide.txt," the baseline model typically generated only the code, neglecting the accompanying non-coding directives. Furthermore, when processing a series of related or unrelated questions within a single session, the agent sometimes either ignored subsequent requests after the initial one or forgot all preceding requests.

Table 1: Examples of prompts are shown in the table in four categories.

Test Purpose	**Prompt 1**	**Prompt 2**	**Prompt 3**
Variant of coding request[7]	Python3: Write a function to find the longest common prefix string amongst an array of strings.	N/A	N/A
Variant of non-coding reques	Write a program that calculates subtraction of two arguments, called subtraction.py.	Write a txt file about the potential direction of improvements of your code	N/A
Unrelated question series	Write a python file that converts a temperature in Celsius to Fahrenheit.	Write a python file that converts a temperature in Fahrenheit to Celsius.	N/A
Related question series	Create a bash script that lists and counts files by type in a given directory.	Expand the script to include options for the user to specify which file types to count or exclude from the count.	Integrate a feature in the script that archives all files of a specified type into a single zip file.

3.2 Testing Pipeline

As OpenDevin is rapidly evolving due to the efforts of numerous developers, building a testing pipeline compatible with different OpenDevin versions presents a significant challenge. To partially address version control in testing automation, we have developed a Selenium Webscraper that can automatically interact with the frontend of OpenDevin. This approach ensures that our testing pipeline remains compatible with any backend modifications, provided that the frontend remains consistent with our version.

Our testing automation, powered by Selenium, systematically processes all test cases and the sub-tasks within each test, feeding them sequentially to the OpenDevin agent. It also captures and stores all chatbox outputs and coding results in folders organized by text index. Subsequently, we conduct a human evaluation of the recorded outputs from OpenDevin for each task.

4 Methodology

In this section, we explore the memory management strategies employed by OpenDevin, particularly focusing on the enhancements to the performance of the monologue agent. OpenDevin consists of several components, including a user interface, a comprehensive framework that features both backend and frontend systems. It utilizes LLMs for performing software engineering tasks through various agents, with the monologue agent currently being the primary one.

The monologue agent operates by using a memory container called "monologue" to store all historical information, such as actions and observations. For each user request, the agent will start a loop to process this task step by step. Between steps, it will ask itself to re-think current situation and log thoughts into the monologue[8]. To prevent looping issues, the agent incorporates prompts that prevent excessive introspection and an agent controller that intervenes if the process stalls. Moreover, when the monologue becomes overly saturated with data, it compacts older, less relevant information.

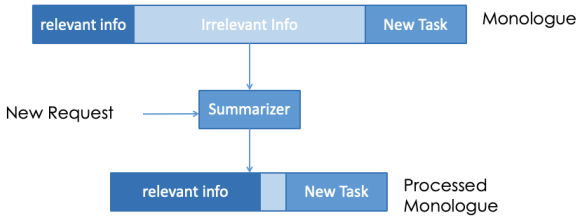

Figure 2: The summarizer processes monologue based on relevance.

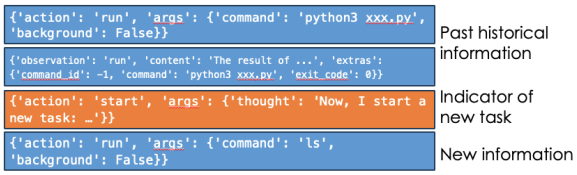

Figure 3: The blue lines are historical information, and the orange line is the indicator.

However, these logic flows are merely concatenated together in the monologue, which can lead to several issues. A primary concern is the accumulation of irrelevant historical information that impede the LLM's processing capabilities for new tasks [9]. On average, the monologue of coding agent would accumulate over 2400 words of messages (3200 tokens) in completing a single task. LLMs struggle more with extracting and utilizing information from extended inputs, particularly when key information is embedded deep within them [10]. Aiming to reduce the length of irrelevant histories and focus LLM's attention to the relevant information and request, we propose a summarizer in figure 2, which sifts through the monologue to retain only the most relevant information for the current task. Whenever it receives a new user request, this summarizer compares the request with each piece of history in the old monologue, and provides a processed monologue. This process engages LLM to assess the relevance using prompting and a few examples [11]. This approach is crucial, especially when the agent processes multiple tasks sequentially, ensuring that each task is handled with a clean slate.

Another issue is that it would be harder for the agent to figure out the logic flow among different tasks. To make it even worse, it is exacerbated by the agent's final statements at the end of tasks, such as "all completed", which could confuse the handling of subsequent tasks. We introduce an indicator as shown in figure 3 among histories of different tasks, which clearly denotes the start of new tasks and provide contextual information about them. This indicator resembles an action within the monologue and is accompanied by enhanced prompts that help the agent recognize the start of a new task and the conclusion of the previous one. Other instructions are also added in the prompts, including expressing the order of the historical information, explicitly explaining the indicator, and the relation between the current task and the historical information.

Additionally, another issue of the memory system is that it has a fixed pipeline to process the user request, which is efficient in handling coding request, but struggles with non-coding requests like summarization and question-answering. To better accommodate these types, we implement a classifier as depicted in figure 4 to discern between coding and non-coding requests by prompting the LLM to make this determination, and the default type is set to coding to prevent ambiguities in hard-to-decide cases. This allows for specialized handling strategies, by restricting coding-specific actions such as command execution for non-coding requests, and encouraging more explanatory interactions in the chat window.

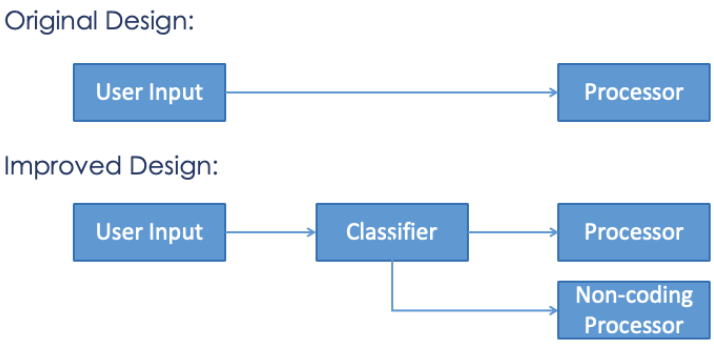

Figure 4: Classifier for coding and non-coding request.

By implementing these enhancements in the monologue agent within OpenDevin, we have successfully achieved continuous iteraction. For instance, a user might initially request OpenDevin to write a Python file and subsequently ask it to "explain the functions in the previous task." Owing to the agent's ability

to manage monologues across tasks, it effectively handles complex, sequential user interactions. Detailed qualitative analysis will be discussed in the following sections.

5 Results

Our experimental evaluation of the testing models involved four types of tests: 1) Variant of coding request, 2) Variant of non-coding request, 3) Unrelated question series, and 4) Related question series. For the Variant of coding request, we further divided the tests into two subcategories: Leetcode style and email style.

The results demonstrate a noticeable improvement in the performance of the improved model across most categories when compared to the baseline model. Specifically, for the Variant of coding request (Leetcode style and email style), the accuracy remained stable at 80% for the Leetcode style while it improved from 14.29% to 42.86% in the email style subcategory. In the Variant of non-coding request, the accuracy improved from 44.44% to 88.89%. The Unrelated/Related question series also showed enhanced performance, improving from 71.43% to 85.71% and 50% to 70%. All the tests were run on gpt-3.5-turbo agent.

Table 2: Accuracy of Baseline vs. Improved Models (in %)

Test Type	**Subcategory**	**Baseline Model**	**Improved Model**
Variant of coding request	Leetcode style	80	80
	Email style	14.29	42.86
Variant of non-coding request	-	44.44	88.89
Unrelated question series	-	71.43	85.71
Related question series	-	50	70

Our accuracy assessment relies on human evaluations of both chatbox and code outputs. A task is deemed a failure if the chatbox explicitly indicates that the task will not be completed, or if "All Done" is not output within 10 minutes. Conversely, if the task is marked as completed by the chatbox's "All Done" output, our annotators then review the output files of the specific task to determine its accuracy. The Inter-Annotator Agreement rate for this evaluation stands at 92%.

6 Discussion

Limitations of the Current Testing Pipeline

Our current testing pipeline primarily relies on frontend tests, which simulate the process of entering test cases into the OpenDevin chatbox and subsequently scraping the output code. This approach is inherently limited by its dependence on the UI maintaining a consistent appearance. Any modifications to the UI or frontend could potentially disrupt the test pipeline's functionality. To mitigate these limitations, we propose an enhancement to the testing pipeline that involves direct integration with OpenDevin's backend. By modifying the UI code and backend, we could enable the system to automatically process uploaded test case files, generate output files, and save these files to local computers. This enhancement would not only make our testing process more robust against frontend changes but also streamline the entire testing workflow, allowing for more efficient and reliable testing of OpenDevin's capabilities.

Challenges with Complex and Non-Explicit Coding Requests

The performance discrepancy observed in OpenDevin's processing of various coding request formats poses a significant challenge. While direct coding questions, such as those from LeetCode, are handled with high accuracy, the performance decreases when these questions are embedded in more complex contexts like emails or project briefs. For example, when tasked with interpreting a detailed project brief for a weather forecasting app, OpenDevin struggles to separate project planning and coding instructions.

To address these issues, we introduced an input_parsing function aimed at simplifying these complex inputs into clearer coding requests. Initially, this function used an LLM to reinterpret the inputs, but

the performance improvements were not as expected. For future enhancements, refining this function to better identify programming-related content is crucial. Incorporating advanced natural language processing techniques or training models on diverse complex inputs could significantly enhance the model's ability to discern and prioritize relevant information, thus improving accuracy in processing multi-faceted requests.

Future Directions and Enhancements

Beyond refining the input_parsing function, other potential strategies could involve incorporating structured data parsing techniques that recognize and organize input according to predefined templates or key phrases typically associated with coding tasks. Implementing machine learning models trained on a dataset of similar project briefs and coding requests might also improve the system's ability to understand and respond to multifaceted inputs. Another approach could be to develop a feedback loop within the system where OpenDevin asks clarifying questions when the input is ambiguous or contains multiple potential tasks, thus ensuring more accurate understanding and response generation.

7 Conclusion

In summary, this research showcases the significant advancements made in enhancing OpenDevin, a cutting-edge code generation AI tool. By addressing key challenges such as handling diverse input types and managing multi-round conversations, the research team has substantially improved the tool's effectiveness and accessibility for a wide range of users.

The implementation of a summarizer function, an indicator for contextual information, and a classifier for differentiating coding and non-coding requests has led to notable improvements in the AI's performance. Experimental results demonstrate the efficacy of these enhancements, with the improved model outperforming the baseline across most test categories.

Despite these advancements, the report recognizes the limitations of the current testing pipeline and proposes strategies for future enhancements, such as incorporating advanced natural language processing techniques and developing a feedback loop for clarifying ambiguous inputs.

The importance of this research lies in its potential to revolutionize the way users interact with coding tasks. By making programming more accessible and efficient for both technical and non-technical users, tools like OpenDevin can democratize the field of software development. As AI-powered coding tools continue to evolve, they have the capacity to transform the landscape of programming, empowering users from diverse backgrounds to engage with coding tasks and fostering innovation across various industries.

In conclusion, this research underscores the significance of continuous development and refinement in AI-powered coding tools. By pushing the boundaries of what is possible with tools like OpenDevin, researchers are paving the way for a future where coding is more intuitive, efficient, and accessible to all.

References

[1] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE].

[2] Anton Lozhkov et al. *StarCoder 2 and The Stack v2: The Next Generation*. 2024. arXiv: 2402.19173 [cs.SE].

[3] GitHub. *GitHub Copilot: Your AI pair programmer*. https://copilot.github.com. 2021.

[4] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL].

[5] Cognition Labs. *Devin: The First AI Software Engineer*. https://www.cognition-labs.com/introducing-devin. 2024.

[6] OpenDevin. *OpenDevin: Code Less, Make More*. https://github.com/OpenDevin/OpenDevin. Retrieved from GitHub. 2024.

[7] *LeetCode—The World's Leading Online Programming Learning Platform*. Retrieved May 1, 2024. 2024. URL: https://leetcode.com/.

[8] Ofir Press et al. *Measuring and Narrowing the Compositionality Gap in Language Models*. 2023. arXiv: 2210.03350 [cs.CL].

[9] Mosh Levy, Alon Jacoby, and Yoav Goldberg. *Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models*. 2024. arXiv: 2402.14848 [cs.CL].

[10] Nelson F. Liu et al. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172 [cs.CL].

[11] Oriol Vinyals et al. *Matching Networks for One Shot Learning*. 2017. arXiv: 1606.04080 [cs.LG].