Debugging like an Expert: Assertion-based Large Language Model Debugging

Anonymous Authors[1]

Abstract

Large Language Models (LLMs) have shown impressive capabilities in code generation, yet they frequently produce programs with subtle, hard-to-detect bugs. To address this challenge, I proposed an assertion-based debugging framework that emulates expert debugging strategies to identify and correct errors in LLM-generated code. Our approach leverages a multi-agent architecture: a divide-and-conquer agent decomposes complex problems into smaller, tractable subtasks; programmer agents generate code enriched with strategic assertions; and a debugger agent iteratively analyzes assertion failures to refine incorrect outputs. By incorporating assertions into the debugging process, our framework improves robustness and correctness, achieving a 3% performance gain on the HumanEvalPlus benchmark suite.

1. Introduction

Code generation and program synthesis have emerged as transformative fields within machine learning, aiming to automate the process of software development. By leveraging advanced neural architectures—particularly large language models (LLMs)—these approaches translate natural language descriptions or formal specifications into executable code. This automation not only accelerates development cycles but also opens avenues for reducing human error and enhancing productivity in complex programming tasks.

Recent advancements in LLMs have catalyzed transformative progress in automated code generation. While early approaches primarily focused on one-shot generation, emerging methodologies increasingly emphasize execution-based checking and multi-agent collaboration to address challenges such as subtle semantic errors and logical inconsistencies. In this context, a growing body of research explores mechanisms for self-debugging, testing, and multi-agent coordination to enhance both the reliability and efficiency of program synthesis.

Notably, (**?**) demonstrates that instructing LLMs to engage in self-reflection and error diagnosis can markedly improve the correctness of generated code. Complementing this, (**?**) introduces an execution-driven approach for assessing code accuracy, where the generation process is iteratively refined based on dynamic feedback obtained through code execution. This execution-based approach is further advanced in (**?**), which emulates human debugging by systematically verifying runtime behavior in a stepwise manner.

Parallel to these self-debugging and testing strategies, multi-agent frameworks have emerged as a powerful paradigm for code generation. Systems such as (**???**) proposes an multi-agent workflow that integrates software engineering methodologies into the program synthesis process, including role specialization, structured communication, and iterative refinement. Similar architectures are also proposed in (**?**) and (**?**). Collectively, these studies underscore a paradigm shift from static, single-shot code generation towards dynamic, collaboration-driven methodologies. By combining self-debugging techniques with execution-based feedback and multi-agent coordination, recent research provides a promising roadmap for developing more robust, reliable, and adaptive code generation systems, laying the groundwork for future advancements in autonomous programming.

Despite recent advances, several critical aspects of human debugging remain underexplored in current state-of-the-art methodologies. In practice, expert programmers routinely incorporate assertions into their code as a proactive measure to enforce correctness, thereby enabling the early detection and localization of bugs during runtime. These runtime checks not only streamline the debugging process but also enhance overall software reliability. For instance, languages like Java and Python automatically perform array boundary checks, offering a layer of protection that languages such as C do not inherently provide. In this project, I aim to investigate the efficacy of integrating assertion-guided strategies into LLM-driven code generation, leveraging these

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Purdue ECE 57000 (Liu, Purdue) Spring 2025. Do not distribute.

human-inspired debugging techniques to improve both the robustness and accuracy of synthesized code.

2. Motivation

2.1. Problem Definition

I follow the problem definition of code generation in (**?**). Basically, each sample can be represented as a triplet (Q, T_v, T_h) where Q is the natural language description of the task, T_v is the target code, and T_h is the code generated by the model. Q may include a description of the task, the signature of the function to be implemented, and the constraints that the code must satisfy. In standard code generation tasks, both Q and T_v are provided as inputs, and the objective is to generate a program P that meets the specifications of both T_v and T_h. The generated program P is evaluated against both T_v and T_h to check its correctness.

In this project, I specifically aim to improve the correctness of code generated by large language models (LLMs). Achieving correctness is particularly challenging for LLMs because, despite their impressive fluency and syntactic capabilities, they often lack a deep semantic understanding of the tasks they are solving. As a result, LLMs can produce code that appears syntactically valid and may even pass a limited set of test cases, yet still fail to fully capture the problem's intent or handle critical edge cases.

To address this issue, I explore multi-agent collaboration-based testing methods, which enhance reliability by introducing diverse perspectives into the validation process. However, LLMs frequently struggle to generate robust and accurate test cases, primarily due to a lack of sufficient task-specific data or reasoning depth.

To illustrate this limitation, I randomly selected one benchmark problem `HumanEval/118` from the HumanEvalPlus benchmark suite. Using the `gpt-4o` model, I generated a corresponding test function. As shown in Figure 1, although the model is capable of generating a correct solution for this problem, it fails to produce valid test cases.

In this project, I propose assertion-based debugging to address this challenge. By embedding runtime assertions throughout the code, I can expose hidden errors that might be overlooked by surface-level test cases. These assertions act as lightweight contracts that enforce input-output constraints, loop invariants, and other correctness properties during execution. When an assertion fails, it provides a precise signal to a debugger agent, which then locates and fixes the faulty function in an iterative loop. This structured approach mimics human debugging strategies and enhances both the reliability and interpretability of LLM-generated code.

2.2. Related Work

Program synthesis and code generation is certainly a wide-researched field, with a plethora of approaches and methodologies. In this section, I have selected several notable work both from top conferences and from state-of-the-art leaderboards (**?**).

SELF-DEBUGGING (**?**) introduces an innovative approach to enhancing the accuracy of code generated by large language models (LLMs). The core idea involves enabling LLMs to autonomously execute their generated code using external programs, analyze the outcomes, identify errors, and subsequently refine their code based on this self-assessment. This self-debugging process allows the model to iteratively improve its code without external feedback, such as unit tests or human instructions. However, the self-debugging approach may failed to utilizing runtime execution information to enhance the accuracy and reliability of code generation, as mentioned by other works (**?**).

Large Language Model Debugger (LDB) (**?**) is a framework designed to emulate human debugging practices by integrating runtime execution information into the debugging process. Unlike traditional methods that treat programs as monolithic units, LDB divides programs into smaller segments, allowing for more granular analysis and refinement. By adopting this step-by-step approach, LDB effectively mirrors human debugging strategies, focusing on smaller code units and utilizing runtime information to enhance the accuracy and reliability of code generation. However, LDB's approach is not generally applicable. It relies heavily on the availability of well-structured intermediate steps and predefined task decomposition, which limits its effectiveness on complex or poorly specified problems. Additionally, LDB does not incorporate assertion-based reasoning or active error localization, making it less effective in cases where silent logic errors occur or when test cases alone fail to capture edge-case behavior. To overcome these limitations, I propose a more flexible and robust framework that integrates assertion-based debugging with multi-agent collaboration, enabling dynamic task decomposition and more accurate error detection through execution-aware reasoning.

Another notable work is MetaGPT (**?**) which leverages large language models (LLMs) to facilitate multi-agent collaboration in complex software engineering tasks. By integrating standardized human workflows into LLM-based multi-agent systems, MetaGPT assigns specialized roles—such as product manager, architect, and engineer—to different agents, mirroring traditional software development pipelines. This role specialization enables the decomposition of intricate tasks into manageable subtasks, promoting coherent and efficient problem-solving. The framework employs an assembly line paradigm, where agents with domain-specific expertise collaborate systematically, reducing errors and

```python
def get_closest_vowel(word):
    """You are given a word. Your task is to find the closest vowel that stands between
    two consonants from the right side of the word (case sensitive).

    Vowels in the beginning and ending doesn't count. Return empty string if you didn't
    find any vowel met the above condition.

    You may assume that the given string contains English letter only.

    Example:
    get_closest_vowel("yogurt") ==> "u"
    get_closest_vowel("FULL") ==> "U"
    get_closest_vowel("quick") ==> ""
    get_closest_vowel("ab") ==> ""
    """
    def is_vowel(ch: str) -> bool:
        return ch in "aeiouAEIOU"
    for i in range(len(word) - 2, 0, -1):
        if is_vowel(word[i]) and not is_vowel(word[i-1]) and not is_vowel(word[i+1]):
            return word[i]
    return ""

def test_get_closest_vowel():
    assert get_closest_vowel("yogurt") == "u"
    assert get_closest_vowel("FULL") == "U"
    assert get_closest_vowel("planet") == "e"
    assert get_closest_vowel("ab") == ""
    assert get_closest_vowel("HeLLo") == ""  # INCORRECT !!!
    assert get_closest_vowel("ABcdEF") == "" # INCORRECT !!!
    assert get_closest_vowel("a") == ""
    ......
```

Figure 1. Example of incorrect test cases generated by `gpt-4o` while the program can be correctly generated.

enhancing the overall quality of the generated solutions. However, MetaGPT does not explicitly address the issue of runtime error detection and prevention, focusing primarily on role specialization and structured communication. In contrast, our assertion-based debugging framework aims to proactively prevent errors during the synthesis process by incorporating assertion-guided strategies, complementing existing multi-agent methodologies.

QualityFlow (**?**) is a recent state-of-the-art work on (**?**) that emphasizes the importance of quality assurance in code generation. By introducing a quality control mechanism that evaluates the correctness and robustness of generated code, QualityFlow ensures that the synthesized solutions meet predefined quality standards. The framework employs a feedback loop that iteratively refines the generation process based on quality metrics, enhancing the reliability and accuracy of the final output. While QualityFlow focuses on post-generation quality assessment, our assertion-based debugging framework aims to proactively prevent errors during the synthesis process by incorporating assertion-guided strategies. By integrating runtime checks and assertions into the code generation pipeline, I aim to enhance the reliability and correctness of synthesized code, complementing existing quality assurance methodologies.

3. Methodology

3.1. Overall Architecture

Figure 2 illustrates an assertion-based debugging framework for large language models (LLMs), which adopts a divide-and-conquer strategy for both problem-solving and debugging. The process begins with a divide-and-conquer agent that decomposes the overall problem specification into a set of smaller subtask specifications. Each subtask is represented as a pair (Q_i, A_i), where Q_i is a natural language description of the subtask, and A_i is a set of assertions that any valid solution must satisfy.

In addition to generating these subtasks, the divide-and-conquer agent also produces the code required to assemble the individual subprograms into a unified solution. Each subtask is then handled by a programmer agent, which implements a corresponding function enriched with detailed runtime assertions. These assertions are designed to catch potential errors and enforce the correctness of the solution.

To verify the correctness of the assembled program, a tester

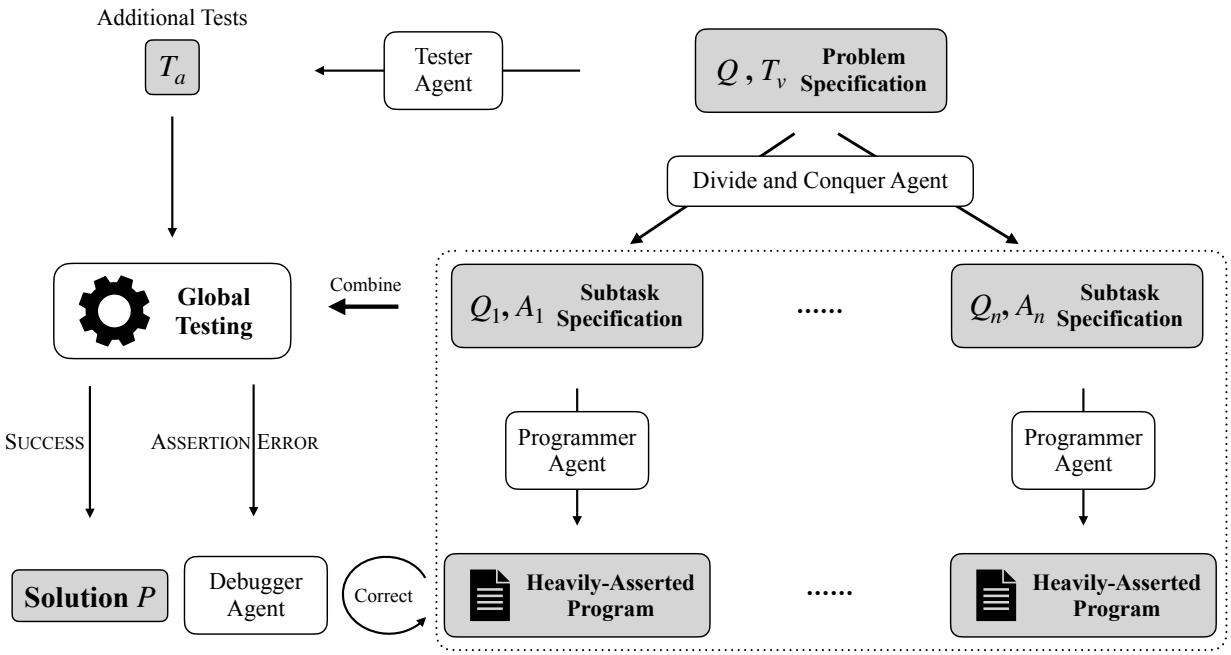

Figure 2. Overview of the proposed assertion-based debugging framework.

agent generates a single unit test function. This function combines predefined and automatically generated test cases to evaluate the entire solution in a global testing phase. If all tests pass, the solution is accepted. Otherwise, if an assertion fails, a debugger agent is invoked.

The debugger agent is the core of this framework. It analyzes the error message to locate the source of the bug by navigating the codebase. Once sufficient information is gathered, it proposes a fix by replacing one or more faulty functions with corrected versions. The testing and debugging cycle continues iteratively until a correct solution is found. This framework mirrors expert-level debugging practices by embedding assertions, systematically isolating faults, and refining the solution step by step.

In some cases, the debugger agent may be unable to localize the issue based solely on the error message—typically due to a misunderstanding of the original problem specification. When this occurs, the entire process is restarted from scratch to produce a new candidate solution.

3.2. Prompt Design

Since the proposed framework requires a large amount of assertions to generated by the programmer agent, the prompt used to guide the LLMs must be carefully designed to produces desired assertions. We must specify exactly what kind of assertions we want the LLMs to generate, and how to incorporate these assertions into the generated code. The prompt design should also consider the balance between the number of assertions and the complexity of the generated code, as an excessive number of assertions may lead to code bloat and performance degradation.

Here I list the important categories of assertions that I aim to specify in the prompt:

1. **Input Assertions** specify the constraints that the input data must satisfy. These assertions are especially useful for checking the correctness of the combination program generated by the divide-and-conquer agent.

2. **Output Assertions** specify the expected output of the program. A lot of engineering efforts need to put into those assertions, as they are the most important part of the program. Thus I force the model to generate such assertions, and recommand the model to create helper functions to check the output match the specialization exactly. For example, in a sorting algorithm, an output assertion may specify that the output list must be sorted in ascending order, which may involves a helper function `isSorted` to ensure that list is completely sorted.

3. **Loop Invariants** can potentially help the model to

understand the loop structure and the expected behavior of the loop without using information from long-distance context, and thus generate more accurate assertions.

Similarly, the prompting strategies used for both the tester agent and the debugger agent play a crucial role in the success of the framework. As illustrate in Figure 1 one common issue arises when test cases generated by the tester agent are themselves flawed. To mitigate this, I instruct the tester agent to rely primarily on random testing, which helps reduce overfitting to a narrow interpretation of the problem. In this setup, the program specification is effectively written twice: once in the implementation prompt and again in the test generation prompt. This redundancy encourages coverage diversity, but also introduces the potential for misalignment between the two representations of the same task.

Given this setup, the debugger agent must be carefully guided to reason across multiple potential sources of error: the program implementation, the embedded assertions, and the test script itself. It must not only identify the component responsible for the failure but also avoid unnecessarily altering correct parts of the system. To support this, the debugger agent is carefully designed to analyze error messages and assertion failures in a structured manner, navigating the codebase using introspection tools to locate faults. By explicitly including the possibility of errors in any part of the development pipeline—including faulty tests or incorrect assertions - the framework ensures that debugging is not biased toward assuming code is always the source of failure.

4. Experimental Result

Now I present the results of our assertion-based debugging framework on the HumanEvalPlus benchmark. I use a simplified version of the architecture in Figure 2 with OpenAI `gpt-4o`. The result is shown in Table **??**. I compare our framework with the original `gpt-4o` model and a version of our framework without the assertion-based debugging. The results show that our framework outperforms both baselines, achieving a pass@1 rate of 88.27% on the HumanEvalPlus benchmark, which is close to 89.6% reported by the state of the art prompting method QualityFlow(**?**).

Table 1. Pass@1 Results on HumanEvalPlus

	pass@1
GPT-4o	83.43
Ours (w/o Assert)	85.18
Ours (w/ Assert)	88.27

Comparing to the original `gpt-4o` model [1], our framework achieves a 4.84% improvement in pass@1 rate. I also conducted an ablation study to evaluate the impact of assertion-based debugging on the performance of our framework. The results show that the pass@1 rate of our framework without assertion-based debugging is 85.18%, which is 3.09% loIr than the version with assertion-based debugging. This indicates that the assertion-based debugging method is effective in improving the correctness of LLM-generated code.

Our implementation and testing results are available at Github[2]. Due to the limited time and resources, I only test our framework on the HumanEvalPlus benchmark. I plan to extend our framework to other benchmarks such as MBPPPlus(**?**) and MHPP(**?**) in the future.

5. Contribution

In this project, I have:

- Proposed a novel assertion-based debugging framework for LLM-generated code, which enhances the reliability and correctness of synthesized code.
- Designed a multi-agent architecture that incorporates assertion-guided strategies, enabling efficient bug detection and localization during runtime.
- Implemented a multi-agent system that includes a divide-and-conquer agent for problem decomposition, programmer agents for code generation with assertions, and a debugger agent for iterative analysis of assertion failures.
- Conducted experiments on the HumanEvalPlus benchmark to evaluate the effectiveness of our framework.

6. Conclusion and Future Directions

In this project, I proposed an assertion-based debugging framework that enhances the reliability and correctness of LLM-generated code. By leveraging a multi-agent architecture, I demonstrated how assertion-guided strategies can improve bug detection and localization during runtime. Our experimental results on the HumanEvalPlus benchmark showed a significant performance gain, highlighting the effectiveness of our approach.

However, our project is not complete because of the limited time and resources. The experimental results are only conducted on the HumanEvalPlus benchmark, and even though it has been shown to have a lot randomness, I don't have the resourcaes to test it repeatedly. In the future, I plan to extend our framework to other benchmarks such as MBPPPlus(**?**) and MHPP(**?**). I also aim to explore the integration of assertion-based debugging with other self-debugging and

[1] Instructed the same as our programmer agent.

[2] https://github.com/YuantianDing/AssertDBG.git

multi-agent collaboration techniques to further enhance the robustness and efficiency of program synthesis.

7. LLM Usage Acknowledgements

I have used ChatGPT and Github Copilot to complete this term paper. Here are the links to my conversations. I also put my usage of Github Copilot in the same place:

- https://chatgpt.com/share/67fc7d4f-f974-8006-9237-2e2d4fe3eadb
- https://chatgpt.com/share/67fc7cd5-d450-8006-af1c-df824848c97f
- https://chatgpt.com/share/67fc7d5d-ea08-8006-a7fc-adf3aa387d13
- https://chatgpt.com/share/67cb68ef-9168-8006-8ad2-7fe15217ad08
- https://chatgpt.com/share/67cb6920-14c8-8006-937e-e3a9c38144a7

However, I failed to track some of the text generated by Github Copilot, which I borrowed from the Checkpoint 2 writeup. It is mainly in Section 1 & 2.

A. You *can* have an appendix here.

You can have as much text here as you want. The main body must be at most 8 pages long. For the final version, one more page can be added. If you want, you can use an appendix like this one.

The `\onecolumn` command above can be kept in place if you prefer a one-column appendix, or can be removed if you prefer a two-column appendix. Apart from this possible change, the style (font size, spacing, margins, page numbering, etc.) should be kept the same as the main bod.y