Exploring Behavior-Driven Development for Code Generation

Anonymous ACL submission

Abstract

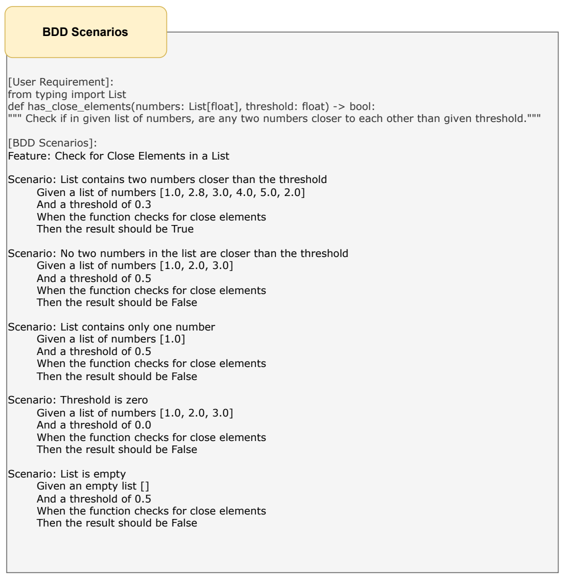
Code generation has long been a challenging task in natural language processing, with existing models often struggling to produce correct and functional code solutions. This paper explores integrating Behavior-Driven Development (BDD)—a user-centric agile methodology—into the code generation process. We propose BDDCoder, a novel multi-agent framework comprising four roles: Programmer, Tester, Requirements Analyst, and User, designed to simulate real-world BDD workflows. BDDCoder consists of two variants: BDD-NL, which uses natural language scenarios for code generation and LLM-based self-validation and BDD-Test, which converts scenarios into executable test cases for code validation. Through empirical evaluation on benchmark datasets (HumanEval, MBPP, and their EvalPlus variants), we demonstrate that BDD-NL with LLM self-validation could hinder code generation performance, while BDD-Test significantly outperforms BDD-NL, achieving up to a 15.1% improvement in *pass@1* scores. Our findings highlight the potential of BDD to enhance requirement clarity and code alignment with user needs, offering a robust framework for future research on integrating software engineering methodologies into automated code generation.

1 Introduction

Large Language Models (LLMs) have revolutionized code generation task by enabling the automatic translating natural language descriptions into executable code (Jiang et al., 2024; Gu et al., 2024; Tong and Zhang, 2024; Lyu et al., 2024). LLMs like Codex (Chen et al., 2021a), Code Llama (Roziere et al., 2023), and GPT-4 (Achiam et al., 2023) have demonstrated remarkable capabilities in understanding and generating code across various programming languages and tasks. By leveraging vast amounts of code-related data from public repositories like GitHub, these models have become indispensable tools for developers, significantly reducing the time and effort required for coding tasks (Jin et al., 2024a; Zhang et al., 2024). However, despite their advancements, challenges remain in ensuring the functional correctness and alignment with user requirements of generated code. Studies have shown that LLMs often produce syntactically valid but logically flawed code, especially when handling edge cases or complex dependencies (Pan et al., 2025; Liu et al., 2024). For instance, while LLMs achieve high pass rates on benchmarks like HumanEval, their performance drops significantly on rigorous evaluations such as HumanEval+.

To address these limitations, recent research has explored multi-agent frameworks that incorporating software engineering methodologies like TDD, Agile development, and the Waterfall model into the code generation process (Hong et al., 2023; Mathews and Nagappan, 2024; Lin et al., 2024; Jin et al., 2024b). These frameworks decompose the code generation task into collaborative subtasks (e.g., requirement analysis, coding, testing), and distribute tasks among specialized agents, such as programmers, testers, and requirement analysts, to simulate a collaborative software development environment.

Test-Driven Development (TDD) (Beck, 2022), where test cases are provided alongside problem statements to ensure that the generated code must be functionally correct and can pass all provided test cases, , has been adapted to guide the code generation process (Mathews and Nagappan, 2024). This approach has been shown to improve LLMs's code generation performance, as providing test cases can improve understanding of the requirements in the problem description and ensure the logical correctness of the generated code. While TDD focuses on technical correctness, Behavior-Driven Development (BDD) (Stevens, 2014; Fox, 2015; North, 2006) extends TDD by emphasizing user-centric requirements, translating natural language scenarios (e.g., "Given-When-Then") into executable specifications. This approach fosters collaboration among developers, testers and non-technical stakeholders, ensuring code aligns with business goals. It extends TDD by focusing on user behavior and writing test cases in natural language, making it accessible to non-programmers. By centering the development process around user needs, BDD ensures that the generated code not only functions correctly but also aligns with business requirements. Despite its widespread adoption in real-world software development, the potential of BDD in automated code generation remains largely unexplored. An example of BDD scenario generated by LLM on a code generation task is shown in fig 1.


BDD Scenarios

```
[User Requirement]:
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
""" Check if in given list of numbers, are any two numbers closer to each other than given threshold."""

[BDD Scenarios]:
Feature: Check for Close Elements in a List

Scenario: List contains two numbers closer than the threshold
        Given a list of numbers [1.0, 2.8, 3.0, 4.0, 5.0, 2.0]
        And a threshold of 0.3
        When the function checks for close elements
        Then the result should be True

Scenario: No two numbers in the list are closer than the threshold
        Given a list of numbers [1.0, 2.0, 3.0]
        And a threshold of 0.5
        When the function checks for close elements
        Then the result should be False

Scenario: List contains only one number
        Given a list of numbers [1.0]
        And a threshold of 0.5
        When the function checks for close elements
        Then the result should be False

Scenario: Threshold is zero
        Given a list of numbers [1.0, 2.0, 3.0]
        And a threshold of 0.0
        When the function checks for close elements
        Then the result should be False

Scenario: List is empty
        Given an empty list []
        And a threshold of 0.5
        When the function checks for close elements
        Then the result should be False
```

Figure 1: An Example of BDD Scenario Generated by LLM.

In this paper, we hope to explore the integration of Behavior-Driven Development (BDD) into the code generation process. We simulate the real-world Behavior-Driven Development (BDD) process in a simple multi-agent system BDDCoder, consisting of four roles: the Programmer, responsible for generating code; the Tester, who validates the code against BDD scenarios; the Requirements Analyst, who formalizes user requirements into structured BDD scenarios; and the User, who provides requirements and feedback. This framework ensures that the generated code is not only functionally correct but also aligned with business goals and user expectations. For fully evaluating the effectiveness of BDD in code generation, we propose two variants of BDDCoder: - **BDD-NL**: Directly using natural language scenarios for code generation and LLM-based validation, simulating a "pure" BDD process. - **BDD-Test**: Converting scenarios into executable test cases for execution feedback, simulating the real-world BDD process.

We conduct experiments to answer the following research questions.

- **RQ1: Can natural language BDD scenarios (BDD-NL) effectively guide LLMs in code generation compared to direct code generation?** While Test-Driven Development (TDD) has proven effective for LLM code generation by aligning code with predefined test cases, its user-centric counterpart—Behavior-Driven Development (BDD)—remains underexplored. BDD's emphasis on natural language scenarios (e.g., "Given-When-Then" templates) could theoretically enhance requirement clarity, but LLMs' reliance on structural patterns (e.g., unit tests in pre-training data) poses a potential mismatch. We first ask: Can raw BDD scenarios (BDD-NL) effectively guide LLMs, or do they introduce noise due to semantic misalignment?

- **RQ2: Does converting BDD scenarios into executable test cases (BDD-Test) improve code generation performance over BDD-NL?** If BDD-NL's natural language scenarios would hinder performance, an alternative approach may bridge the gap. Inspired by real-world BDD process, we propose BDD-Test, which automatically converts BDD scenarios into executable test cases (e.g., Python assert statements). This raises a critical question: Does BDD-Test resolve the limitations of BDD-NL, and if so, to what extent?

- **RQ3: Can LLM act as reliable validators in BDD-NL (natural language self-checking)?** BDD-NL assumes LLMs can self-validate code against natural language scenarios—a capability crucial for simulating human-centric workflows. However, LLMs' inherent limitations in understanding natural language nuances may hinder this process. We thus investigate: Can LLMs reliably act as testers in BDD-NL, or does self-validation introduce false positives/negatives?

- **RQ4: How is the BDD Scenario Quality and Test Cases Correctness?** As we use BDD scenarios(BDD-NL) and corrsponding test cases(BDD-Test) to guide the code generation process, the quality of the BDD scenarios and test cases is crucial.

- **RQ5: How does BDD compare to TDD in guiding LLM code generation?** Finally, we compare the performance of BDD(BDD-Test) with TDD in guiding LLM code generation, as BDD is a user-centric extension of TDD. We aim to investigate whether BDD can provide additional benefits over TDD in guiding LLM code generation.

2 Methodology

2.1 Multi-Agent System Design (BDDCoder)

As shown in fig 2, we simulate the BDD process by integrating four key roles: the Requirements Analyst, User, Programmer, and Tester.

2.1.1 Requirements Analyst

In our framework the Requirements Analyst acts as the central communication bridge between the user and the development team. The Analyst is responsible for generating BDD scenarios based on user requirements and iteratively refining these scenarios through feedback from the user. The BDD scenarios, written in natural language, capture the desired behavior of the system and serve as a high-level specification for the development process. The iterative refinement process ensures that the scenarios accurately reflect user needs and provide clear guidance for the subsequent coding phase.

2.1.2 User

The User is the ultimate stakeholder in our framework, providing requirements and feedback, used to simulate a human user in the BDD process. During the scenario refinement phase, the User reviews the BDD scenarios drafted by the development team, ensuring they accurately meet user needs. The active involvement of the User ensures that the final solution is user-centric and meets business goals.

2.1.3 Programmer

The Programmer takes the refined BDD scenarios as input and generates the corresponding code. The scenarios guide the programmer in implementing the functionality required to meet user expectations. The generated code is then passed to the Tester for validation. In BDD-Test mode, the Programmer takes the converted test cases and user requirements as input for generating code.

2.1.4 Tester

The Tester evaluates the generated code against the BDD scenarios to determine whether the code satisfies all specified requirements. This involves: Validation: Evaluating whether the code can pass all scenarios. Test Report Generation: Documenting any discrepancies or issues and providing detailed feedback to the Programmer for further refinement. The iterative feedback loop between the Tester and the Programmer continues until the code successfully passes all scenarios, ensuring that the final output meets user requirements.

Scenario-to-Test Conversion In the BDD-Test mode, the Tester automatically converts natural language scenarios into Python `assert` statements using prompt-based parsing (prompt template in Appendix A). For example, a scenario *"Given input X, when processed, then output Y"* is mapped to `assert func(X) == Y`. These executable tests are validated via `pytest` to provide deterministic feedback.

2.2 Framework Overview

The BDDCoder framework operates in two modes to evaluate different BDD integration strategies:

BDD-NL Mode (Pure BDD Simulation) In this mode, all artifacts—scenarios, code, and test reports—are expressed in natural language. The Tester Agent validates code by prompting the LLM to check scenario compliance, mimicking human-centric BDD workflows.

BDD-Test Mode Here, natural language scenarios are automatically converted to executable test cases (Python `assert` statements) after user validation. The Tester agent runs these tests via `pytest` against the generated code, , providing a test report as feedback.

2.3 Experiment Setup

2.3.1 Benchmark Datasets

In this work, we utilize the following widely-used code generation benchmarks for experimental investigation: HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021) and their EvalPlus

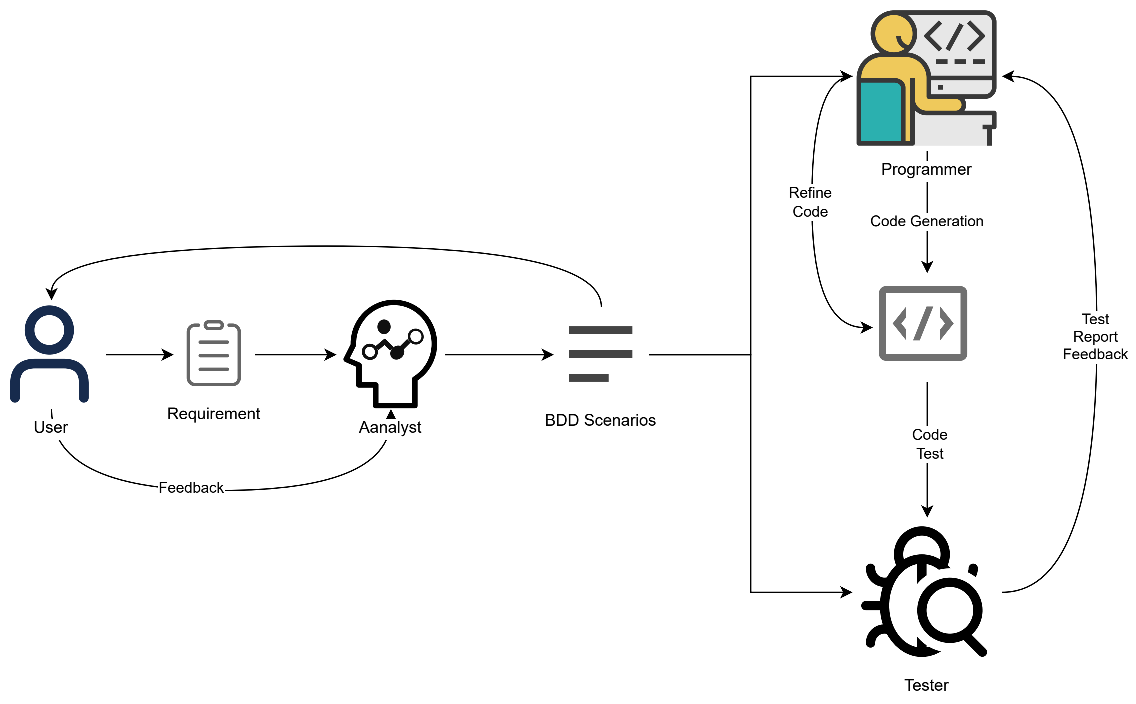


Figure 2: The BDDCoder framework.

(Liu et al., 2024) variants(i.e. HumanEval+ and MBPP+). HumanEval consists of 164 human-written programming problems, each with a function signature, natural language description, canonical solution, and test cases. MBPP(EvalPlus version) contains 378 manually verified programming problems, each with a natural language description, a code solution, and 3 test cases. To enhance evaluation rigor, we employed EvalPlus, which extends HumanEval and MBPP with 80x and 35x more test cases, respectively. These extended datasets, HumanEval+ and MBPP+, provide broader coverage of edge cases and complex scenarios, enabling more accurate detection of errors in generated code.

2.3.2 Evaluation Metrics

We use *pass@1* as the evaluation metrics for the code generation task, a widdly used metric which evaluates whether a single generated code solution can pass all the tests (Chen et al., 2021b; Austin et al., 2021; Dong et al., 2024; Huang et al., 2023).

2.3.3 BDD Variant Configuration

For each dataset, we evaluate two configurations: - **BDD-NL**: Use raw scenarios for code generation and LLM self-validation. - **BDD-Test**: Convert scenarios to tests for code generation and pytest execution. The iterations limit rounds are set to 5 for both scenarios refining and code fixing.

3 Results and Analysis

3.1 RQ1: Can natural language BDD scenarios (BDD-NL) effectively guide LLMs in code generation compared to direct code generation?

We evaluate the performance of BDD-NL with fllowing LLMs: GPT-3.5-turbo(0125 version) and GPT-4o-mini to answer this research question. The results is reported in Table 1, the "Direct" method refers to directly prompt LLM to generate code, and this serves as the baseline, representing the basic code generation perfomance of LLMs. We process HumanEval and MBPP by extracting necessary imports, function signature and the natural language description of the problem to construct the prompt input in same format, illustrated with the examples in the appendix. And the "BDD-NL" method refers to using BDD scenarios to guide the code generation process. From the results we can find that directly guiding LLMs with natural language BDD scenarios (BDD-NL) resulted in performance degradation across all LLMs, with *pass@1* scores dropping by up to 15.1% on MBPP+ and at least 0.6% on HumanEval for GPT-4o-mini.

It suggests that the BDD scenarios, while providing a structured and user-centric approach to code generation, may introduce additional complexity or constraints that hinder the LLMs' ability to generate correct and functional code. One possible explanation is that the BDD scenarios written in natural language may not align perfectly with the LLMs' internal representations of the problem, leading to a suboptimal code generation.

3.2 RQ2: Can converting BDD scenarios into executable test cases (BDD-Test) mitigate the limitations of BDD-NL?

As reported in Table 1, the results show that converting BDD scenarios into executable test cases (BDD-Test) significantly improves code generation performance compared to BDD-NL. For instance, on the HumanEval dataset, the *pass@1* score for GPT-3.5-turbo improved from 76.8% (BDD-NL) to 87.7% (BDD-Test), and for GPT-4o-mini, it increased from 74.4% to 85.4%. We attribute this phenomenon to the model has learned a large amount of code data containing test cases during pre-training and instruction fine-tuning, while much more limited code data containing BDD scenarios. This indicates that the structural alignment with LLMs' pre-training patterns (via BDD-Test) resolves the limitations of BDD-NL, leading to better code generation performance. The findings suggest that while natural language scenarios may introduce noise, converting them into executable test cases can bridge the gap between BDD's user-centric scenarios and LLMs' reliance on structural inputs.

3.3 RQ3: Can LLM act as reliable validators in BDD-NL (natural language self-checking)?

In BDD-NL, we directly use LLM itself as a code tester to evaluate the code correctness by prompting LLM to act as a tester to detemine whether the generated code can pass all the BDD scenarios to similate the real-world real testing process of validating input-output matches. However, we overlooked whether this approach is truly effective, i.e. whether LLM can genuinely serve as a BDD test validator. To address this issue, we conducted ablation experiment to evaluate the effectiveness of LLM as a code tester in BDD-NL by removing the Tester Agent from the BDDCoder framework and directly using the code generated by the Programmer Agent as the final output. As shown in 2, we find that the LLM model is not a reliable code tester. Specifically, using LLMs to validate whether generated code passes all scenarios did not enhance correctness and even led to a slight performance degradation. Across all models and datasets, the *pass@1* scores decreased when LLMs were used as validators in BDD-NL. For example, on the HumanEval dataset, the *pass@1* score for GPT-3.5-turbo dropped from 77.4% (BDD-NL without LLM self-validation) to 76.8% (BDD-NL with LLM self-validation). The findings highlight the limitations of LLMs in self-validation via natural language scenarios and suggest that alternative validation methods, such as automated testing frameworks, may be more effective.

3.4 RQ4: How is the BDD Scenario Quality and Test Cases Correctness?

We evaluate the quality of the generated BDD scenarios and convertd test cases by excuting them against ground truth code and measured the pass rate and accuracy of the generated test cases. For further investigating, we also prompt the LLMs to directly generate the same number of Python assert statements. Tables 3 and 4 summarize the results obtained on the HumanEval and MBPP datasets for two models: GPT-3.5-turbo and GPT-4o-mini. Here, "Total Items" refers to the total number of problems in the dataset, "Total Cases" is the number of test cases generated, "Passed Cases" indicates the number of test cases that passed when executed on the real code solution, and "Correct Items" denotes the number of problems for which the test cases were entirely correct.

The results indicate that for both models and datasets, direct generation of test cases generally yields higher accuracy metrics compared to the BDD-based conversion approach. For instance, in the case of GPT-3.5-turbo on the HumanEval dataset, the direct generation method achieved an Item-Accuracy of 0.5427 and a Cases-Accuracy of 0.7129, which represent slight improvements over the corresponding BDD-based results (0.5366 and 0.7092, respectively). On the MBPP dataset, the improvement is more pronounced; the direct method achieved an Item-Accuracy of 0.5319 (an increase of nearly 13 percentage points) and a Cases-Accuracy of 0.6706 (approximately 5 percentage points higher) compared to the BDD-based approach. Similarly, for GPT-4o-mini, the HumanEval results under the direct generation condition show an improvement in Item-Accuracy (0.4085 versus 0.3902) and a more substantial gain in Cases-Accuracy (0.6942 versus 0.6451) relative to the BDD-based conversion. On MBPP, the di-

LLM	Method	HumanEval	HumanEval+	MBPP	MBPP+
gpt-3.5-turbo	Direct	78.7	75.0	77.0	66.4
	BDD-NL	76.8(↓ 1.9)	69.5(↓ 5.5)	75.7(↓ 1.7)	64.6(↓ 1.8)
	BDD-Test	87.8(↑ 10.1)	84.1(↑ 9.1)	85.6(↑ 8.6)	78.3(↑ 11.9)
gpt-4o-mini	Direct	75.0	71.3	74.9	64.3
	BDD-NL	74.4(↓ 0.6)	69.5(↓ 1.8)	66.1(↓ 8.8)	49.2(↓ 15.1)
	BDD-Test	85.4(↑ 10.4)	82.9(↑ 11.3)	87.3(↑ 13.4)	79.1(↑ 14.8)

Table 1: Code generation performance with BDDCoder.

LLM	Method	HumanEval	HumanEval+	MBPP	MBPP+
gpt-3.5-turbo	BDD-NL	76.8	69.5	75.7	64.6
	BDD-NL$_{w/o\,test}$	77.4(↑ 0.6)	72.6(↑ 3.1)	74.9(↓ 0.8)	65.1(↑ 0.5)
gpt-4o-mini	BDD-NL	74.4	69.5	70.6	51.1
	BDD-NL$_{w/o\,test}$	76.2(↑ 1.8)	72.0(↑ 2.5)	71.4(↑ 0.8)	54.8(↑ 3.7)

Table 2: BDD-NL code generation performance with and w/o LLM self-verification.

rect generation method also outperforms the BDD-based method in terms of Item-Accuracy (0.5132 compared to 0.3942), while the improvement in Cases-Accuracy is modest (0.6461 vs. 0.6040).

These findings suggest that although converting BDD scenarios into executable test cases provides a structured and user-centric framework, the scenarios themselves and conversion process may introduce noise or result in information loss, thereby slightly degrading the overall test case quality. Conversely, when the language model is directly prompted to generate test cases, it appears to leverage its pre-training on code and testing patterns more effectively, yielding higher accuracy. Future work should focus on refining the conversion process from natural language scenarios to executable tests, aiming to combine the strengths of BDD (i.e., clear specification of user requirements) with the robust test generation capabilities of LLMs.

3.5 RQ5: How does BDD compare to TDD in guiding LLM code generation?

We compare the performance of BDD(BDD-Test) with TDD (TGen) in guiding LLM code generation on the HumanEval and MBPP datasets, and report the results in Table 5. For the GPT-3.5-turbo model, we observe that BDD consistently outperforms TDD. Specifically, on the HumanEval dataset, BDD-Test achieves a *pass@1* of 87.7%, compared to 76.2% for TGen. A similar trend is observed across the other datasets: HumanEval+ (84.1% for BDD-Test vs. 73.2% for TGen), MBPP (85.6% vs. 76.2%), and MBPP+ (78.3% vs. 69.0%). For the GPT-4o-mini model, BDD also shows generally superior performance over TDD accrross all datasets.

In summary, these results indicate that BDD generally provides superior guidance for LLM-based code generation when compared to TDD. This performance superiority can be attributed to the fact that BDD emphasizes user-centric scenarios, which can provide LLMs with clearer guidance on how to align generated code with user needs. These findings highlight the potential of BDD to improve code quality and accuracy, making it a preferable approach for tasks where user behavior and requirements play a central role in guiding the development process.

4 Related Work

4.1 Large Language Models for Code Generation

Large Language Models (LLMs) have significantly advanced the field of automatic code generation. Models like Codex, StarCoder, and Code Llama leverage extensive training on large-scale code repositories to generate code across multiple programming languages, demonstrating capabilities ranging from function-level completion to competitive programming. Despite their success, the generated code often suffer form syntactic correctness but semantic flaws, the code may pass basic tests but fail to align with implicit user requirements or handle edge cases. Recent studies highlight that LLMs struggle with dynamic requirements and contextual nuances, particularly in real-world scenarios where specifications evolve iteratively. These

Dataset	Condition	Total Cases	Passed Cases	Correct Items	Item-Accuracy	Cases-Accuracy
HumanEval	BDD	533	378	88	0.5366	0.7092
	Direct	533	380	89	0.5427	0.7129
MBPP	BDD	1172	724	151	0.4016	0.6177
	Direct	1172	786	200	0.5319	0.6706

Table 3: Results for GPT-3.5-turbo on HumanEval and MBPP

Dataset	Condition	Total Cases	Passed Cases	Correct Items	Item-Accuracy	Cases-Accuracy
HumanEval	BDD	896	578	64	0.3902	0.6451
	Direct	896	622	67	0.4085	0.6942
MBPP	BDD	1735	1048	149	0.3942	0.6040
	Direct	1735	1121	194	0.5132	0.6461

Table 4: Results for GPT-4o-mini on HumanEval and MBPP

limitations underscore the need for methodologies that bridge high-level user intent and low-level code implementation.

4.2 Agent-based Methods

To enhance performance and robustness, recent studies have explored the integration of multi-agent systems in code generation that simulate collaborative software development workflows (Jin et al., 2024b). These frameworks decompose code generation into specialized roles (e.g., analyst, programmer, tester) and leverage iterative feedback: MetaGPT integrates standardized operating procedures (SOPs) to coordinate agents, reducing error propagation through role-specific prompts and validation. CodeAgent extends this paradigm by incorporating DevOps tools (e.g., CI/CD pipelines) for repository-level code synthesis, outperforming commercial tools like GitHub Copilot in complex tasks. TGen adopts a Test-Driven Development (TDD) approach, where agents iteratively refine code based on test feedback, demonstrating higher pass rates than direct generation. By combining multi-agent collaboration with software engineering methodologies, these frameworks address key challenges such as error propagation, context understanding, and real-world applicability. They not only improve the functional correctness and robustness of generated code but also enhance the adaptability and scalability of AI-driven software development, offering a comprehensive solution to the challenges of modern software engineering.

5 Conclusion and Future Work

In this work, we perform the first empirical study to explore the integration of Behavior-Driven Development (BDD) into the code generation process. Through experimental exploration, we believe that directly using natural language-described BDD scenarios to guide code generation and verification is not very effective. However, this does not negate the feasibility of introducing BDD into code generation. On the contrary, we consider this a promising research direction, though currently limited by the fundamental capabilities of LLMs, requiring models trained on more BDD scenario data. We hope that in the future, BDD scenarios can be directly used as guidance and standards for the entire development process, enabling end-to-end complex software development. Moreover, since BDD introduces user participation, further exploration of human-in-the-loop applications is possible. In practical applications, users can collaborate and communicate with AI systems acting as software teams to jointly develop scenarios that clarify requirements and guide the development process.

Code generation has been a challenging task in the field of natural language processing, and existing models often fail to generate correct code solutions that pass all tests. Behavior-Driven Development (BDD) is an agile software development methodology that enhances team collaboration and software quality by focusing on user behavior. In this work, we propose BDDCoder, a novel multi-agent framework that incorporates Behavior-Driven Development (BDD) into the code generation process to improve the performance of code solutions. BDDCoder includes four roles: the Programmer, who generates code; the Tester, who generates and executes test cases; the Requirements Analyst, who analyzes user requirements; and the User, who writes behaviors.

LLM	Method	HumanEval	HumanEval+	MBPP	MBPP+
gpt-3.5-turbo	BDD-Test	87.7	84.1	85.6	78.3
	TGen	76.2	73.2	76.2	69.0
gpt-4o-mini	BDD-Test	89.6	87.2	85.7	77.8
	TGen	88.4	85.4	80.4	72.8

Table 5: Code generation preformance with BDD and TDD.

6 Limitations

In this study, we evaluated the effectiveness of Behavior-Driven Development (BDD) in guiding Large Language Model (LLM) code generation. While our findings indicate that BDD can improve code generation performance, several limitations should be considered. First, due to the resource constraints, our experiments focused on two LLMs—GPT-3.5-turbo and GPT-4o-mini—and four datasets: HumanEval, HumanEval+, MBPP, and MBPP+. The limited scope of models and datasets may not fully represent the broader applicability of BDD. Future studies should incorporate a wider range of models and datasets to assess the generalizability of these findings. Besides, we only evaluated the performance of BDD in guiding code generation tasks and did not explore the potential of BDD in real-world software development, future work should extend the BDDCoder framework to more complex software development tasks and evaluate its effectiveness in real-world scenarios. Finally, our study focused on exploring the effectiveness of BDD in guiding LLM code generation and did not consider other code generation approaches or methodologies.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Kent Beck. 2022. *Test driven development: By example*. Addison-Wesley Professional.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38.

S. Fox. 2015. All you need to know about behaviour-driven software. https://web.archive.org/web/20150901151029/http://behaviourdriven.org/.

Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2024. On the effectiveness of large language models in domain-specific code generation. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024a. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024b. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Feng Lin, Dong Jae Kim, and TH Chen. 2024. Soen-101: Code generation by emulating software process models using large language model agents. *arXiv preprint arXiv:2403.15852*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. 2024. Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*.

Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-driven development for code generation. *arXiv preprint arXiv:2402.13521*.

D. North. 2006. Introducing bdd. https://dannorth.net/introducing-bdd/.

Ruwei Pan, Hongyu Zhang, and Chao Liu. 2025. Codecor: An llm-based self-reflective multi-agent framework for code generation. *arXiv preprint arXiv:2501.07811*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

P. Stevens. 2014. Understanding the differences between bdd tdd. https://cucumber.io/blog/bdd/bdd-vs-tdd/.

Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051, Miami, Florida, USA. Association for Computational Linguistics.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

A Appendix

A.1 Data Format and Basic Prompt

We process HumanEval and MBPP by extracting necessary imports, function signature and the natural language description of the problem to construct the prompt input in same format, the data format and the direc code generation prompt(basic prompt) are illustrated as follows:

```
import math
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than given threshold."""
```

Figure 3: Data Format

```
[system prompt]: You are a Python programmer.
[user prompt]:
Complete the following code
import math
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than given threshold."""
**Important**:
1. Only generate a single complete Python code snippet, without any additional information or strings before or after
the code.
2. The code should start with ```python and end with ```
```

Figure 4: Basic Prompt

A.2 BDDCoder Prompt Templates

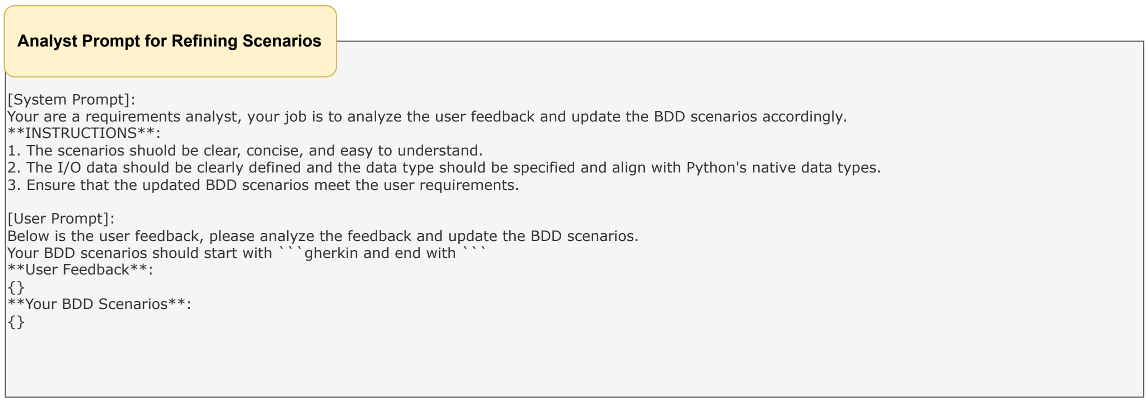
Below are the prompt templates used in BDDCoder for the Programmer, Tester, Requirements Analyst, and User roles.

Analyst Prompt for Gnerating Scenarios

```
[System Prompt]:
Your are a requirements analyst, your job is to design BDD scenarios according to the user requirements.
**INSTRUCTIONS**:
1. The scenarios shuold be clear, concise, and easy to understand.
2. The I/O data should be clearly defined and fllows the function signature and the data type should be specified and align with Python's native data types.
3. Write at least 3 scenarios to cover the user requirements.
4. Do not include any exception handling in the scenarios.

[User Prompt]:
Below is the user requirement, your BDD scenarios should start with ```gherkin and end with ```
**User Requirement**:
```

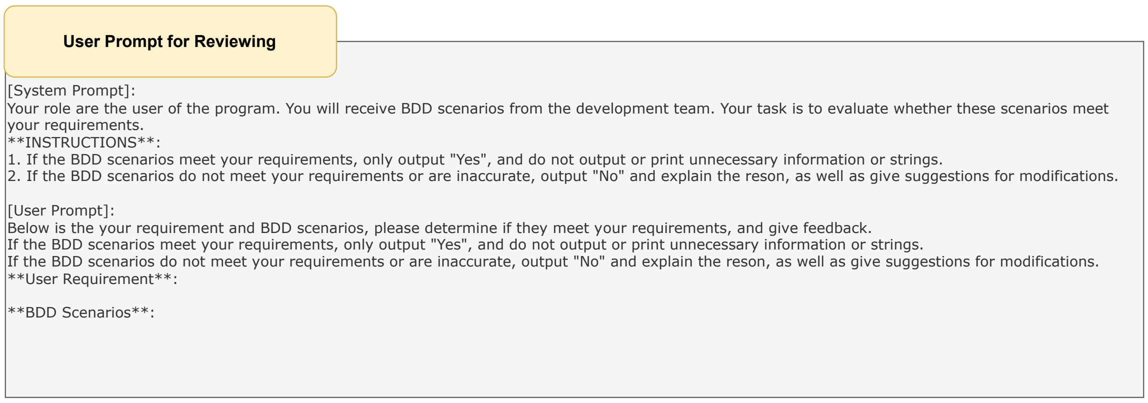
Figure 5: Analyzer Prompt for Gnenrating BDD Scenarios


Analyst Prompt for Refining Scenarios

[System Prompt]:
Your are a requirements analyst, your job is to analyze the user feedback and update the BDD scenarios accordingly.
INSTRUCTIONS:
1. The scenarios shuold be clear, concise, and easy to understand.
2. The I/O data should be clearly defined and the data type should be specified and align with Python's native data types.
3. Ensure that the updated BDD scenarios meet the user requirements.

[User Prompt]:
Below is the user feedback, please analyze the feedback and update the BDD scenarios.
Your BDD scenarios should start with ```gherkin and end with ```
User Feedback:
{}
Your BDD Scenarios:
{}

Figure 6: Analyzer Prompt for Refining BDD Scenarios

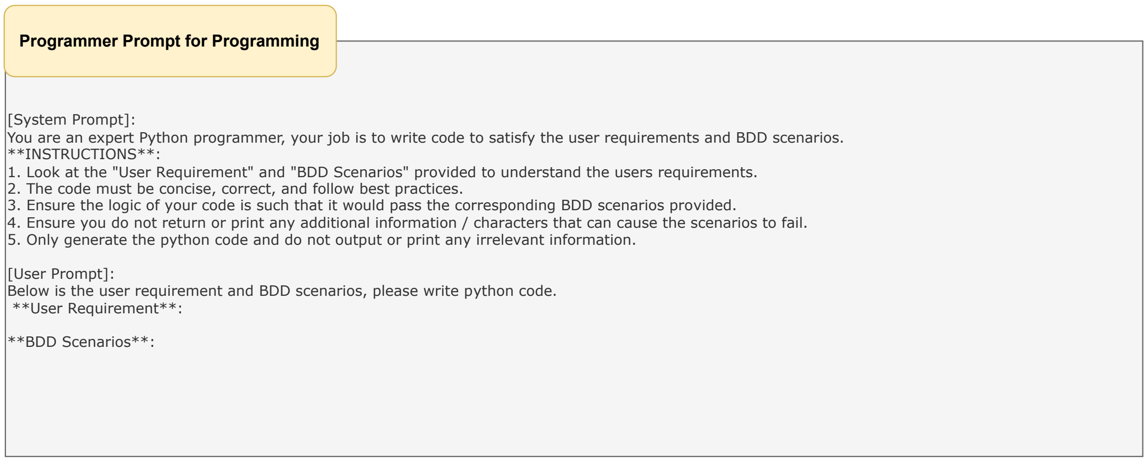

User Prompt for Reviewing

[System Prompt]:
Your role are the user of the program. You will receive BDD scenarios from the development team. Your task is to evaluate whether these scenarios meet your requirements.
INSTRUCTIONS:
1. If the BDD scenarios meet your requirements, only output "Yes", and do not output or print unnecessary information or strings.
2. If the BDD scenarios do not meet your requirements or are inaccurate, output "No" and explain the reson, as well as give suggestions for modifications.

[User Prompt]:
Below is the your requirement and BDD scenarios, please determine if they meet your requirements, and give feedback.
If the BDD scenarios meet your requirements, only output "Yes", and do not output or print unnecessary information or strings.
If the BDD scenarios do not meet your requirements or are inaccurate, output "No" and explain the reson, as well as give suggestions for modifications.
User Requirement:

BDD Scenarios:

Figure 7: User Prompt for Reviewing BDD Scenarios

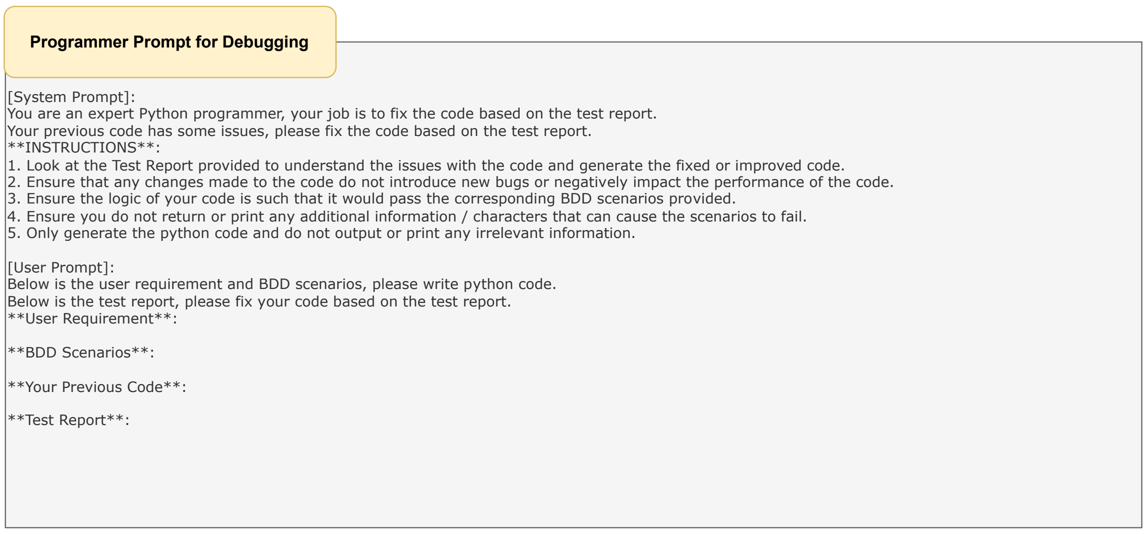

Programmer Prompt for Programming

[System Prompt]:
You are an expert Python programmer, your job is to write code to satisfy the user requirements and BDD scenarios.
INSTRUCTIONS:
1. Look at the "User Requirement" and "BDD Scenarios" provided to understand the users requirements.
2. The code must be concise, correct, and follow best practices.
3. Ensure the logic of your code is such that it would pass the corresponding BDD scenarios provided.
4. Ensure you do not return or print any additional information / characters that can cause the scenarios to fail.
5. Only generate the python code and do not output or print any irrelevant information.

[User Prompt]:
Below is the user requirement and BDD scenarios, please write python code.
User Requirement:

BDD Scenarios:

Figure 8: Programmer Prompt for Gnenrating Code



Programmer Prompt for Debugging

[System Prompt]:
You are an expert Python programmer, your job is to fix the code based on the test report.
Your previous code has some issues, please fix the code based on the test report.
INSTRUCTIONS:
1. Look at the Test Report provided to understand the issues with the code and generate the fixed or improved code.
2. Ensure that any changes made to the code do not introduce new bugs or negatively impact the performance of the code.
3. Ensure the logic of your code is such that it would pass the corresponding BDD scenarios provided.
4. Ensure you do not return or print any additional information / characters that can cause the scenarios to fail.
5. Only generate the python code and do not output or print any irrelevant information.

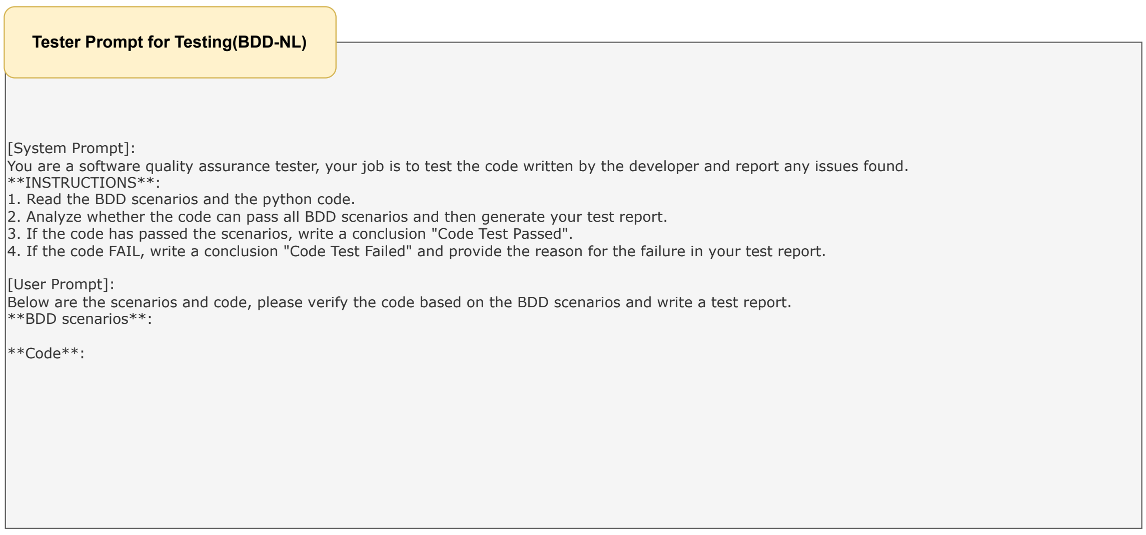
[User Prompt]:
Below is the user requirement and BDD scenarios, please write python code.
Below is the test report, please fix your code based on the test report.
User Requirement:

BDD Scenarios:

Your Previous Code:

Test Report:

Figure 9: Programmer Prompt for Refining Code



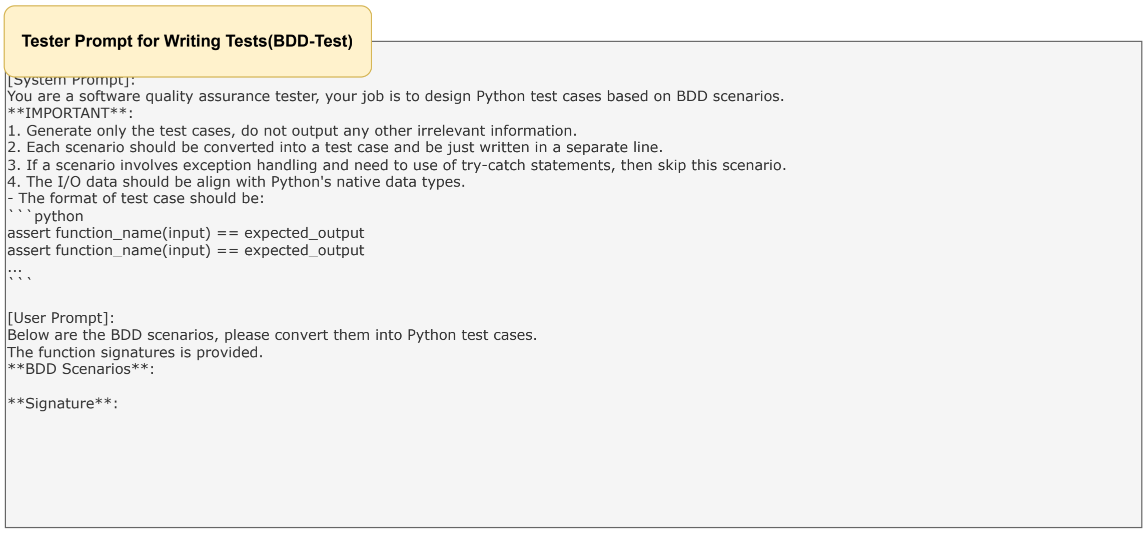
Tester Prompt for Testing(BDD-NL)

[System Prompt]:
You are a software quality assurance tester, your job is to test the code written by the developer and report any issues found.
INSTRUCTIONS:
1. Read the BDD scenarios and the python code.
2. Analyze whether the code can pass all BDD scenarios and then generate your test report.
3. If the code has passed the scenarios, write a conclusion "Code Test Passed".
4. If the code FAIL, write a conclusion "Code Test Failed" and provide the reason for the failure in your test report.

[User Prompt]:
Below are the scenarios and code, please verify the code based on the BDD scenarios and write a test report.
BDD scenarios:

Code:

Figure 10: Tester Prompt for Testing



Tester Prompt for Writing Tests(BDD-Test)

~~~
[System Prompt]:
You are a software quality assurance tester, your job is to design Python test cases based on BDD scenarios.
**IMPORTANT**:
1. Generate only the test cases, do not output any other irrelevant information.
2. Each scenario should be converted into a test case and be just written in a separate line.
3. If a scenario involves exception handling and need to use of try-catch statements, then skip this scenario.
4. The I/O data should be align with Python's native data types.
- The format of test case should be:
```python
assert function_name(input) == expected_output
assert function_name(input) == expected_output
```

[User Prompt]:
Below are the BDD scenarios, please convert them into Python test cases.
The function signatures is provided.
**BDD Scenarios**:

**Signature**:
~~~

Figure 11: Tester Prompt for Generating Test Cases