Large Language Models as Test Case Generators: Performance Evaluation and Enhancement

Anonymous ACL submission

Abstract

Code generation with Large Language Models (LLMs) has been extensively studied and achieved remarkable progress. As a complementary aspect to code generation, test case generation is of crucial importance in ensuring the quality and reliability of code. However, using LLMs as test case generators has been much less explored. Current research along this line primarily focuses on enhancing code generation with assistance from test cases generated by LLMs, while the performance of LLMs in test case generation alone has not been comprehensively examined. To bridge this gap, we conduct extensive experiments to study how well LLMs can generate high-quality test cases. We find that as the problem difficulty increases, state-of-the-art LLMs struggle to generate correct test cases, largely due to their inherent limitations in computation and reasoning. To mitigate this issue, we further propose a multi-agent framework called *TestChain* that decouples the generation of test inputs and test outputs. Notably, TestChain uses a ReAct format conversation chain for LLMs to interact with a Python interpreter in order to provide more accurate test outputs. Our results indicate that TestChain outperforms the baseline by a large margin. Particularly, in terms of the accuracy of test cases, TestChain using GPT-4 as the backbone achieves a 13.84% improvement over the baseline on the LeetCode-hard dataset.

1 Introduction

Large Language Models (LLMs) have shown significant capabilities in code generation, paving the way for transformative changes in software development. Among these LLMs, there are those specifically designed for coding, such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2022) and CodeLlama (Roziere et al., 2023), as well as general-purpose models like GPT-3.5 and GPT-4 (Achiam et al., 2023).

Test case generation usually refers to the automatic process of creating test cases. It is essential for ensuring the quality and reliability of code, serving as a complementary aspect to code generation. Note that the concept of test cases spans various levels (unit testing, system testing, etc.) and scopes (functional testing, security testing, etc.) within the software testing process. Following the recent related studies (Chen et al., 2022; Huang et al., 2023a,b; Shinn et al., 2023), we consider function-level unit test cases in this paper, where *a test case* refers to a pair of input and expected output for the function defined in the given context. Figure 1 describes the formulation of test case generation.

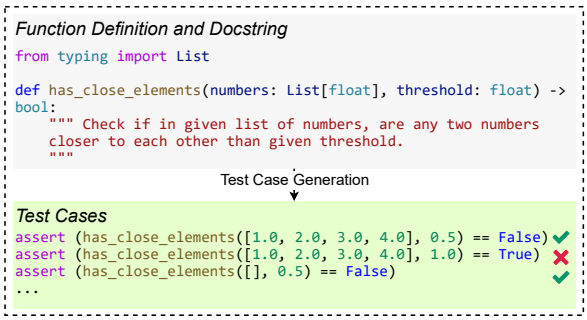


Figure 1: Formulation of test case generation in this paper, where an assertion encapsulates a test case.

Because of breakthroughs in code generation achieved by LLMs, it is intuitive to apply these models to test case generation as well. This type of research has only recently emerged as a field of interest. CodeT (Chen et al., 2022) uses the test cases generated by LLMs to select a single solution from multiple code samples. In CodeCoT (Huang et al., 2023a), Reflexion (Shinn et al., 2023) and AgentCoder (Huang et al., 2023b), the execution of programs on test cases generated by LLMs provides feedback information for self-correction.

However, most of current research in this direction primarily focuses on utilizing test cases

to benefit code generation, with little attention to the quality of test cases generated by LLMs. If the test cases generated by LLMs contain many errors, the effectiveness of these approaches will be significantly diminished. A great challenge in test case generation is how to accurately map each test input to its corresponding output, often requiring precise mathematical calculations and complex logical reasoning. Nonetheless, existing studies (Frieder et al., 2023; Liu et al., 2023) have shown that even the most capable LLMs are not proficient in this regard. Thus, it naturally leads us to pose the following question: *Can LLMs really generate high-quality test cases?*

In this paper, we first conduct extensive experiments to explore this question, where the quality of test cases is evaluated by comprehensive metrics including accuracy, line coverage and our proposed new metric called *Code-with-Bugs* concerning the strength of tests. We examine four notable LLMs comprising both open and closed-source frameworks. Specifically, we include two open-source models: StarChat (Li et al., 2023) and CodeLlama (Roziere et al., 2023), alongside two close-source models: GPT-3.5 and GPT-4. We choose a relatively easy dataset HumanEval (Chen et al., 2021) and a more challenging dataset LeetCode-hard (Shinn et al., 2023) for evaluation. Experiments show that StarChat and CodeLlama struggle to generate correct test cases, while GPT-3.5 and GPT-4 perform well on HumanEval but tend to produce many incorrect test cases on LeetCode-hard.

Based on the insights from the empirical evaluation, we further propose a multi-agent framework called *TestChain*, in order to enhance the performance of LLMs in test case generation. In TestChain, one agent called *Designer agent* is responsible for generating diverse test inputs, while another agent called *Calculator agent* is responsible for correctly mapping the inputs to outputs. Notably, Calculator agent uses a ReAct (Yao et al., 2022) format conversation chain interaction with a Python interpreter for executing code, which can significantly reduce the complexity and inaccuracy in the input-output mapping. Experiments demonstrate that our approach significantly outperforms the baseline in terms of all metrics. In summary, the main contributions of this paper are twofold:

- We conduct a comprehensive evaluation of LLMs in test case generation. We find that the performance of LLMs drops sharply in handling harder problems, where the generated test cases may contain many errors.
- We propose *TestChain*, which significantly enhances the performance of LLMs in generating test cases. TestChain achieves an accuracy of **71.79%** with GPT-4 on the LeetCode-hard dataset, while the few-shot baseline method only gets an accuracy of **57.95%**.

2 Can LLMs Really Generate High-Quality Test Cases?

To thoroughly investigate the capabilities of LLMs in generating test cases, we select four LLMs and two datasets for experimentation. We employ three metrics to evaluate the quality of test cases from different aspects, including correctness and strength.

2.1 Experimental Setup

Models. The four typical LLMs include two open-source models: *StarChat-beta* and *CodeLlama-13b-Instruct-hf*, and two closed-source models: *GPT-3.5-turbo-1106* and *GPT-4-turbo-preview*. In the following, we refer to the four models by *StarChat*, *CodeLlama*, *GPT-3.5* and *GPT-4* in short.

Datasets. We choose a popular Python programming dataset HumanEval that contains a canonical solution for each question. Moreover, we choose a more challenging dataset LeetCode-hard that includes 39 difficult Python programming tasks. For some questions in the original LeetCode-hard, the canonical solution is either missing or faulty. So we manually collect correct solutions for them and verify these solutions via the LeetCode website.

We have also noticed that LLMs may plagiarize the test case examples included in the question prompt, so we remove all the examples from the prompts of the two datasets. Examples of prompts with and without examples can be found in Appendix A. In the following, we refer to the two datasets without examples by *HumanEval-no-exp* and *LeetCode-no-exp*.

Generation Setup. For using LLMs as test case generators, we design a *system prompt* that sets the LLM as a Python tester and instructs it to generate *basic*, *edge*, and *large scale* test cases (Huang et al., 2023b) for a given context. We consider two paradigms for generation: *zero-shot* and *few-shot*. For few-shot generation, examples illustrating the test case generation are prefixed to the user prompt. Please refer to Figures 9–10 in Appendix B for the details of the prompts used in our experiments.

Model	Method	HumanEval-no-exp			LeetCode-no-exp	
		Accuracy (%)	**Line Cov** (%)	**CwB** (%)	**Accuracy** (%)	**Line Cov** (%)
StarChat	Zero-shot	37.44	51.86	45.61	9.74	31.10
	Few-shot	41.10	52.27	45.46	12.31	23.61
CodeLlama	Zero-shot	51.83	63.88	54.85	23.08	58.61
	Few-shot	67.07	71.71	63.23	28.72	55.95
GPT-3.5	Zero-shot	65.98	71.60	69.60	26.67	63.68
	Few-shot	74.02	74.69	74.15	38.97	73.66
GPT-4	Zero-shot	84.02	77.54	82.53	58.97	88.83
	Few-shot	84.63	77.04	83.11	57.95	88.47

Table 1: Evaluation results of zero-shot method and few-shot method.

For each question, we remove duplicate and syntactically erroneous test cases. If more than five remain, we retain the first five for evaluation; otherwise, we keep all. The HumanEval-no-exp dataset comprises 164 questions, with a maximum of 820 test cases. The LeetCode-no-exp dataset includes 39 questions, with a maximum of 195 test cases.

Metrics. A test case is considered to be correct if it can be passed by the canonical solution. The accuracy measures the percentage of correct test cases among all those generated. To comprehensively assess the quality of test cases for a question, line coverage (Line Cov) is used as another metric, measuring the proportion of lines accessed when the test cases are executed on the canonical solution. For comparison, we calculate the average line coverage on the full dataset.

Line coverage alone is recognized as insufficient for evaluating the strength of test cases (Andrews et al., 2006). We also introduce a new metric called *Code-with-Bugs* pass rate (CwB in short). To compute CwB for a set of test cases, we first randomly generate a number of faulty programs (20 in our experiments) for the question. Then, CwB is the proportion of programs that fail this test set. We use the average CwB on the full dataset for comparison. Note that to conduct a fair comparison, we utilize another model, CodeGen-Mono 6B (Nijkamp et al., 2022), to generate faulty programs. Additionally, CwB is only applied to HumanEval in our experiments, as the canonical test cases for verifying correctness are not available for LeetCode-hard.

To further investigate why LLMs generate incorrect test cases, we analyze error types, categorizing them into *Assertion Error*, *Runtime Error* and *Timeout Error*:

- **Assertion Error**: Indicates a mismatch between the execution output of the target function and the expected output.
- **Runtime Error**: Indicates an internal error during function execution, often due to non-compliant input.
- **Timeout Error**: Indicates that the execution of the target function exceeded the allotted time limit (1 second in our experiments).

We will provide statistics on the occurrence of each error type.

Parameters. For all the models, we use a temperature of $T = 0.2$ and top_p is set to 0.95. The max_new_tokens for HuggingFace models and the max_tokens for OpenAI models are set to 1024. For GPT-3.5 and GPT-4, we call them via the official OpenAI APIs [1]. For StarChat and CodeLlama, we use HuggingFace transformers (Wolf et al., 2019) for inference. We conduct experiments on two V100 GPUs with 32GB of memory each.

2.2 Evaluation Results

Figure 2 shows the number of test cases each model and method successfully generated. It reveals that, for the relatively easy HumanEval-no-exp dataset, models are capable of producing an ample number of test cases. Conversely, for the harder LeetCode-no-exp dataset, less advanced models like StarChat face difficulties in generating a sufficient quantity of test cases. This also implies that the most powerful models like GPT-3.5 and GPT-4 can capture the syntax of the test cases very well.

Table 1 presents the evaluation results across all models and datasets. The results reveal that the few-shot method generally outperforms the zero-shot method in terms of accuracy. This is particularly clear for CodeLlama on HumanEval-no-exp (a

[1] https://platform.openai.com

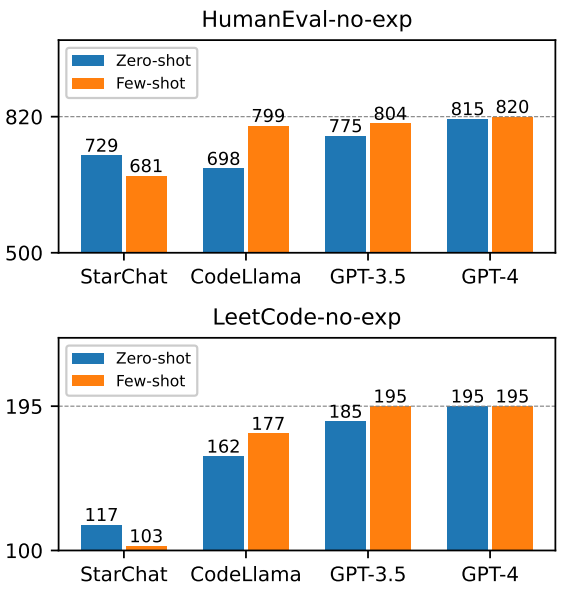


Figure 2: Statistics on the number of test cases successfully generated. The gray line indicates the maximum number of test cases allowed by the dataset.

15.24% improvement) and GPT-3.5 on LeetCode-no-exp (a 12.3% improvement). While in terms of test strength, the situation varies significantly between different models. For example, few-shot is of great help to GPT-3.5 in achieving an improved test strength, while it is unhelpful or even harmful to StarChat in this respect. It is worth noting that, for GPT-4, there is little difference between zero-shot and few-shot in all metrics. This suggests that for the particularly powerful models like GPT-4, the illustrating examples may not be necessary.

In Table 1, we can also observe a sharp decline in accuracy for all the generation methods from HumanEval-no-exp to LeetCode-no-exp. Even the accuracy of GPT-4 decreases by about 25%, reaching only about 58% on LeetCode-hard. This underscores the need for further advancements in test case generation, particularly for hard questions. In terms of line coverage, there has been a notable decrease observed in StarChat and CodeLlama, whereas GPT-3.5 and GPT-4 have not experienced a similar decline. This indicates that the overall quality of test cases generated by the most advanced models like GPT-4 is bottlenecked by accuracy rather than coverage.

Figure 3 shows the number of incorrect test cases for each type of error. It can be seen that Assertion Error is the most prevalent type of error, while the other types of errors constitute only a small fraction. This suggests that the trickiest issue for generating correct test cases lies in accurately computing the corresponding test outputs given the test inputs.

In summary, LLMs are capable of generating a considerable number of correct test cases for relatively easy questions, such as those in HumanEval. However, for harder questions such as those in LeetCode-hard, all of them experience a sharp decline in accuracy and struggle to generate correct test cases. It becomes evident that the pivotal factor in elevating the quality of generated test cases is the enhancement of their correctness, as the overall quality is primarily constrained by accuracy for the advanced LLMs like GPT-4. The correctness of test cases essentially hinges on the accurate mapping of test inputs to their outputs, a process that may involve complex calculations and reasoning, areas where LLMs do not excel. This motivates us to decompose this mapping problem into a number of steps to reduce the complexity, and to utilize external tools to enhance the precision of computation and reasoning.

3 Methodology

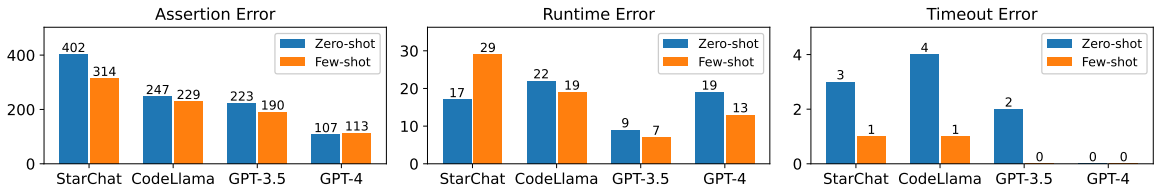
3.1 TestChain Framework

Based on the above insights, we propose the *TestChain* framework that allows LLMs to generate test cases with the assistance of a Python interpreter. TestChain uses the divide-and-conquer idea to decompose the test case generation into two sequential sub-tasks: test input generation and test output generation. The two subtasks are handled by two agents called *Designer agent* and *Calculator agent*, respectively. Because the test input-output mapping may involve complex computation and reasoning, TestChain divides the mapping process into a number of small steps using a conservation chain similar to ReAct (Yao et al., 2022). In each step, the Python code snippet will be written by the LLM and then be executed by the Python interpreter, in order to achieve the desired goal. Figure 4 illustrates our TestChain framework.

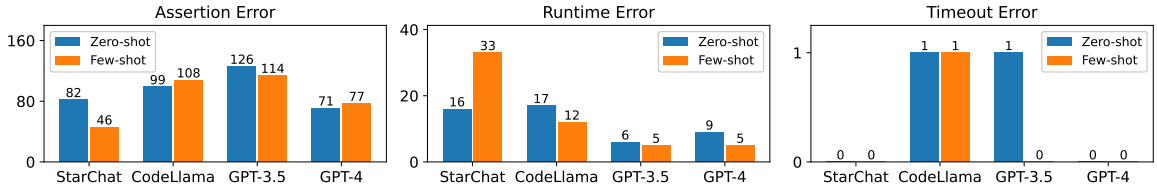
It should be noted that in the generation process, Designer agent will be called only once to generate diverse test inputs, but Calculator agent will be called for each test input.

3.2 Designer Agent

Designer agent employs the LLM for test input generation. In this agent, the system prompt describes that it will generate basic, edge, and large scale test



(a) Statistics on the HumanEval-no-exp dataset.



(b) Statistics on the LeetCode-no-exp dataset.

Figure 3: Statistics on the number of incorect test cases for each type of error.

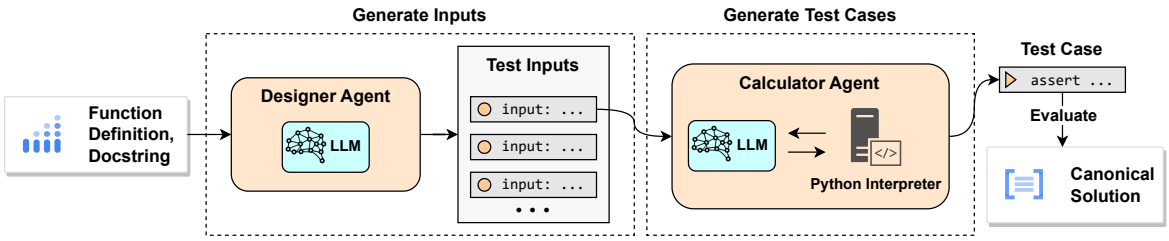


Figure 4: Illustration of the TestChain framework.

inputs. The few-shot prompt provides an example of the three types of test inputs in addition to the function definition and docstring. Please refer to Figures 11–12 in Appendix B for the details of the prompts used by Designer agent. After generation, the duplicated test inputs will be omitted and the retained ones will be the input to Calculator agent.

Note that Designer agent does not need to concern itself with test outputs; it only needs to generate test inputs rather than complete test cases. With this approach, the LLM can concentrate on producing diverse test inputs without the necessity of computing the corresponding test outputs, thereby minimizing the likelihood of errors.

3.3 Calculator Agent

Calculator agent is also powered by LLMs. Based on each test input generated by Designer agent, Calculator agent is instructed to determine the corresponding test output and write the entire assert statement. In Calculator agent, the system prompt defines the conversation format in addition to role assignment, and the few-shot prompt contains a full example of a conversation trajectory. Please refer to Figures 13–14 in Appendix B for the details of the prompts used by Calculator agent.

Interaction with a Python Interpreter. We create a ReAct format conversation chain for LLMs to interact with the Python interpreter. The conversation starts with a user message that contains the function definition, docstring and a test input. Note that if the few-shot setting is used, the few-shot prompt should be prefixed to this user message. In each round, the LLM will think about what to do and write a Python code snippet or write the test case directly. The executed result of the Python code snippet will be returned to the LLM subsequently. The conversation ends when the LLM writes the test case successfully.

During the conversation process, if the LLM only generates thinking text without code and test case, a *go_on_prompt* like "Observation: go on." will be sent to the LLM. The max round of iterations is set to 5, if the LLM fails to write the test

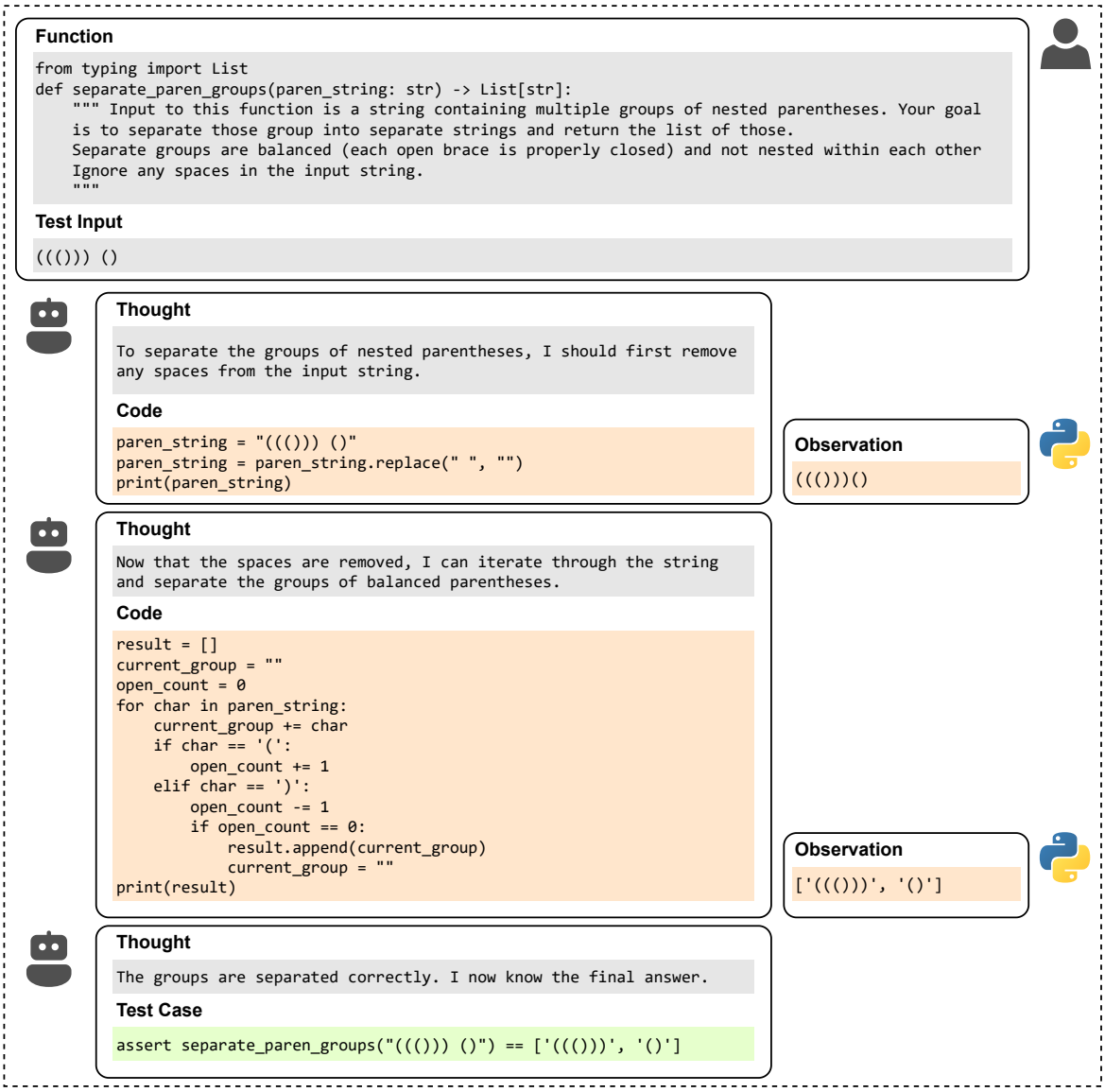


Figure 5: Example of the conversation process produced by Calculator agent.

case after 5 rounds, a *final_prompt* like "Thought: I now know the final answer.\nTest Case:" will be sent to the LLM to force it to write the test case.

Once started, Calculator agent will run a Python shell environment. During the conversation, all the code snippets will be executed in the same context, which means the subsequent code snippets can access variables from preceding code snippets.

Figure 5 shows an example of the conversation process produced by Calculator agent. In this example, Calculator agent is tasked with parsing the input string into a list of nested parentheses while disregarding any spaces within the input string. To handle this question, the LLM divides the task into two steps: (1) Remove any spaces from the input string; (2) Iterate through the string and separate it into parentheses. For each step, the LLM writes Python code snippet and obtains the executed results from the Python interpreter. Finally, the LLM receives the separated parentheses and writes the complete test case. Please refer to Figures 17–34 in Appendix B for more examples.

4 Evaluation of TestChain

4.1 Experimental Setup

We follow the same setups in Section 2.1 for datasets and metrics. We consider two advanced models GPT-3.5 and GPT-4 in our TestChain framework. Because TestChain uses the few-shot setting by default, we choose the few-shot method depicted

Model	Method	HumanEval-no-exp			LeetCode-no-exp	
		Accuracy (%)	Line Cov (%)	CwB (%)	Accuracy (%)	Line Cov (%)
GPT-3.5	Few-shot	74.02	74.69	74.15	38.97	73.66
	TestChain	**80.85**	**77.53**	**80.80**	**48.72**	**80.23**
GPT-4	Few-shot	84.63	77.04	83.11	57.95	88.47
	TestChain	**90.24**	**80.00**	**88.66**	**71.79**	**90.60**

Table 2: Evaluation results of the few-shot method and the TestChain method.

in Section 2.1 as the baseline for comparison.

4.2 Evaluation Results

We analyze the statistics regarding the number of successfully generated test cases, as illustrated in Figure 6. Our results indicate that the advanced models such as GPT-3.5 and GPT-4 exhibit the strong capability to produce a sufficient number of valid test cases using both the few-shot method and the TestChain method. Hence, the accuracy comparison among generation methods primarily relies on the number of correct test cases that a method can generate.

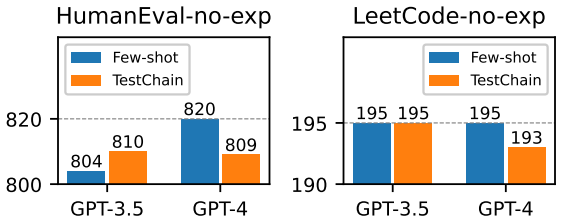


Figure 6: Statistics on the number of test cases successfully generated.

Table 2 presents a comparison of all metrics between the TestChain method and the few-shot method. It is evident that TestChain achieves significant improvements over the few-shot baseline across all metrics and datasets, regardless of whether GPT-3.5 or GPT-4 is utilized as the backbone model. Notably, on the challenging LeetCode-no-exp dataset, TestChain with GPT-4 surpasses the corresponding baseline by 13.84% in terms of accuracy. As for Line Cov and CwB metrics, our TestChain method also always achieves better results than the few-shot method. These findings underscore the ability of our proposed TestChain to generate high-quality test cases that are both accurate and reliable.

Figure 7 displays the statistical distribution of incorrect test cases across different error categories. It can be seen that Assertion Error is still the most common type of error for all the methods. But the number of assertion errors by TestChain is greatly reduced compared to the few-shot method, contributing most to TestChain's higher accuracy. This demonstrates that our TestChain, leveraging Python codes for step-by-step computation and reasoning, effectively decreases the incidence of errors in test outputs, consequently improving the accuracy of the test cases generated.

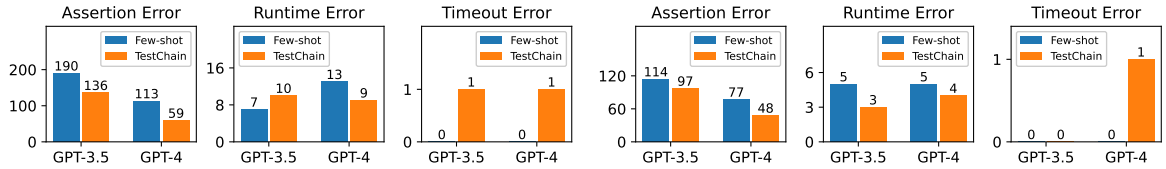
4.3 Effectiveness of the Ingredients in TestChain

In this subsection, we want to examine the effectiveness of two ingredients in the TestChain framework: the decoupling of test input and output generation, and the interaction with the Python interpreter. To this end, we design a modified TestChain method, where Calculator agent no longer uses the Python interpreter and instead uses a few-shot prompt to generate the test case directly. Please refer to Figures 15–16 in Appendix B for the details of the prompts used by this modified TestChain method. We use the same setups with Section 4.2 and compare it with the original TestChain method along with the few-shot baseline method.

Method	Accuracy (%)	Line Cov (%)
Few-shot	57.95	88.47
TestChain (no-py)	61.54	83.65
TestChain	**71.79**	**90.60**

Table 3: Results on the LeetCode-no-exp dataset with GPT-4. TestChain (no-py) refers to the TestChain method without the Python interpreter interaction.

Table 3 shows the comparison results. The main difference between the modified TestChain method and the few-shot method is that the modified TestChain separates the generation of test inputs and test outputs, while the few-shot method generates them together. The modified TestChain outperforms the few-shot method by 3.59% in accuracy, demonstrating the benefit of the decoupling through two agents. The original TestChain



(a) On the HumanEval-no-exp dataset.

(b) On the LeetCode-no-exp dataset.

Figure 7: Statistics on the number of incorrect test cases for each type of error.

achieves substantial improvements over the modified one in terms of both accuracy and line coverage. This suggests that in TestChain, the interaction with the Python interpreter is vital in enhancing the generation performance.

5 Related Work

LLMs for Test Case Generation. Recently, there has been a growing trend of utilizing LLMs for test case generation. Tufano et al. (2020) and Li et al. (2022) utilize pre-trained language models fine-tuned on labeled data to generate test cases. CodeT (Chen et al., 2022) directly generates test cases with the powerful model Codex using zero-shot prompts. CodeCoT (Huang et al., 2023a) employs a Chain of Thought prompt to generate both solutions and test cases for programming tasks. Reflexion and AgentCoder employ LLMs to generate test cases and execute them on programs to produce feedback information for self-correction.

Automatically generating test cases with LLMs can also be used in real-world software development. Zhang et al. (2023) utilizes GPT-4 to generate security tests to find vulnerabilities. Schäfer et al. (2023) employs LLMs to assist in automatic software testing by generating unit tests. Kang et al. (2023) utilizes LLMs to generate test cases to reproduce general bugs in software.

Multi-Agent Structure with LLMs. Utilizing multiple agents driven by LLMs to collaborate in solving tasks has emerged as a popular approach. Self-Collaboration (Dong et al., 2023) establishes a team comprising analyst, coder, and tester roles for software development. INTERVENOR (Wang et al., 2023b) employs a learner agent for generating programs and a teacher agent for suggesting repairs. AgentCoder (Huang et al., 2023b) introduces a programmer agent, a test designer agent, and a test executor agent for code generation and refinement, with the programmer agent and the test designer agent powered by LLMs. AutoGen (Wu et al., 2023a) provides a straightforward method for creating multi-agent applications with LLMs.

LLMs Interaction with Tools. By defining specific conversation formats, the generations of LLMs can be parsed into tool invocations, enabling interaction between LLMs and external tools. ReAct (Yao et al., 2022) employs a chain in "Thought/Action/Observation" format, allowing LLMs to provide tool types and return the execution results of these tools to LLMs. Recently, an open-source project LangChain[2] allows the creation of LLM applications with ReAct or custom-defined conversation chains. FunSearch (Romera-Paredes et al., 2023) utilizes Google's PaLM 2 (Anil et al., 2023) and customized evaluators to discover new solutions to the cap set problem. HuggingGPT (Shen et al., 2023) employs GPT-4 and numerous expert models to solve AI tasks in a collaborative way. WebGPT (Nakano et al., 2021) allows LLMs to use a browser to assist in answering questions. AutoGen can create a Python interpreter environment locally or via docker and interact with LLM agents. Research based on AutoGen can create conversations to conduct hyperparameter Optimization (Wang et al., 2023a) and solve math problems (Wu et al., 2023b).

6 Conclusion

In this paper, we have conducted a thorough performance evaluation of LLMs in test case generation. We find that, for relatively hard questions such as those in LeetCode-hard, even the state-of-the-art LLMs like GPT-4 have great difficulty in generating correct test cases. To address this challenge, we propose TestChain, characterized by decoupling the generation of test inputs and test outputs, and interacting with the Python interpreter through the ReAct chain. Our results clearly demonstrate the superiority of TestChain over the baselines.

[2]https://python.langchain.com

Limitations

The proposed TestChain framework demands high model capabilities, thus it is tailored for more robust models such as GPT-3.5 and GPT-4. Future work can explore how to enhance the performance of weaker models in test case generation based on the few-shot paradigm.

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*.

James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*.

Simon Frieder, Luca Pinchetti, Alexis Chevalier, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, and Julius Berner. 2023. Mathematical capabilities of chatgpt. *arXiv preprint arXiv:2301.13867*.

Dong Huang, Qingwen Bu, and Heming Cui. 2023a. Codecot and beyond: Learning to program and test like a developer. *arXiv preprint arXiv:2308.08784*.

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

Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. 2023. Evaluating the logical reasoning ability of chatgpt and gpt-4. *arXiv preprint arXiv:2304.03439*.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2023. Mathematical discoveries from program search with large language models. *Nature*.

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

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*.

Chi Wang, Susan Xueqing Liu, and Ahmed H. Awadallah. 2023a. Cost-effective hyperparameter optimization for large language model generation inference. In *AutoML'23*.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2023b. Intervenor: Prompt the coding ability of large language models with the interactive chain of repairing. *arXiv preprint arXiv:2311.09868*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023a. Autogen: Enabling next-gen llm applications via multi-agent conversation framework.

Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2023b. An empirical study on challenging math problem solving with gpt-4. In *ArXiv preprint arXiv:2306.01337*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Ying Zhang, Wenjia Song, Zhengjie Ji, Na Meng, et al. 2023. How well does llm generate security tests? *arXiv preprint arXiv:2310.00710*.

A Dataset Examples

Figure 8 shows the prompt of HumanEval and LeetCode datasets with and without examples.

B Prompts for Test Case Generation

Figures 9–16 provide prompts used by different generation methods considered in our paper.

C Examples of Trajectories by Calculator Agent

Figures 17–34 provide four examples of trajectories produced by Calculator agent, where the correct test outputs are obtained given the test inputs.

Prompt with Examples

```python
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.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

Prompt without Examples

```python
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 8: Illustration of the modifications for the prompt of the HumanEval and LeetCode-hard datasets.

System Prompt

```
As a Python tester, your task is to create comprehensive test cases for the function given the
definition and docstring. These test cases should encompass Basic, Edge, and Large Scale scenarios to
ensure the code's robustness, reliability, and scalability. Write each test case with a single line of
assert statement.
```

Figure 9: System prompt for the zero-shot and the few-shot method.

Few-shot Prompt

```
EXAMPLES:

Function:
```python
from typing import List
def find_the_median(arr: List[int]) -> float:
    """
    Given an unsorted array of integers `arr`, find the median of the array.
    The median is the middle value in an ordered list of numbers.
    If the length of the array is even, then the median is the average of the two middle numbers.
    """
```
Test Cases:
```python
# basic test cases
assert find_the_median([3, 1, 2]) == 2
assert find_the_median([1, 3, 2, 5]) == 2.5
# edge test cases
assert find_the_median([1]) == 1
assert find_the_median([-1, -2, -3, 4, 5]) == -1
assert find_the_median([4, 4, 4]) == 4
# large scale test cases
assert find_the_median([3] * 20) == 3
```

END OF EXAMPLES.
```

Figure 10: Prompt for the few-shot method.

Designer Agent System Prompt

```
As a Python tester, your task is to create comprehensive test inputs for the function given the
deinition and docstring. These test inputs should encompass Basic, Edge, and Large Scale scenarios to
ensure the code's robustness, reliability, and scalability. Write each test input in a single line and
start with a `input:`
```

Figure 11: System prompt for the Designer agent.

Designer Agent Few-shot Prompt

```
EXAMPLES:

Function:
```python
from typing import *
def find_the_median(arr: List[int]) -> float:
    """
    Given an unsorted array of integers `arr`, find the median of the array.
    The median is the middle value in an ordered list of numbers.
    If the length of the array is even, then the median is the average of the two middle numbers.
    """
```
Test Inputs:
```text
# basic test inputs
input: [3, 1, 2]
input: [1, 3, 2, 5]
# edge test inputs
input: [1]
input: [-1, -2, -3, 4, 5]
input: [4, 4, 4]
# large scale test inputs
input: [3] * 20
```

END OF EXAMPLES.
```

Figure 12: Few-shot prompt for the Designer agent.

Calculator Agent System Prompt

```
As a Python programmer, your task is to calculate the test output and write the test case statement
corresponding to the test input for the function given the definition and docstring. Write each test
case with a single line of assert statement.
You have access to a Python interpreter, allowing you to execute any Python code snippets that assist
in calculating the test cases.

Use the following format:
Function: The function definition and docstring
Test Input: One test input to the function
Thought: You should always think about what to do
Code: The Python code you want to execute
Observation: The execute result of the Python code
... (this Thought/Code/Observation can repeat N times)
Thought: I now know the final answer.
Test Case: The test case statement

Begin!
```

Figure 13: System prompt for the Calculator agent.

Calculator Agent Few-shot Prompt

```
EXAMPLES:

Function:
```python
from typing import List
def find_the_median(arr: List[int]) -> float:
    """
    Given an unsorted array of integers `arr`, find the median of the array.
    The median is the middle value in an ordered list of numbers.
    If the length of the array is even, then the median is the average of the two middle numbers.
    """
```
Test Input:
```text
input: [1, 3, 2, 5]
```
Thought: To find the median of the given array, I should first sort the array.
Code:
```python
arr = [1, 3, 2, 5]
arr = sorted(arr)
print(arr)
```
Observation:
[1, 2, 3, 5]
Thought: I should then verify whether the length of the array is even or odd.
Code:
```python
len(arr) % 2 == 0
```
Observation:
Out[1]: True
Thought: The length of the array is even, the median is the average of the two middle numbers.
Code:
```python
median = float(arr[int(len(arr)/2)] + arr[int(len(arr)/2 - 1)]) / 2
print(median)
```
Observation:
2.5
Thought: I now know the final answer.
Test Case:
```python
assert find_the_median([1, 3, 2, 5]) == 2.5
```

END OF EXAMPLES.
```

Figure 14: Few-shot prompt for the Calculator agent.

Calculator Agent (no-py) System Prompt

```
As a Python programmer, your task is to calculate the test output and write the test case statement
corresponding to the test input for the function given the definition and docstring. Write each test
case with a single line of assert statement.
```

Figure 15: System prompt for the Calculator agent without the Python interpreter.

Calculator Agent (no-py) Few-shot Prompt

```
EXAMPLES:

Function:
```python
from typing import List
def find_the_median(arr: List[int]) -> float:
    """
    Given an unsorted array of integers `arr`, find the median of the array.
    The median is the middle value in an ordered list of numbers.
    If the length of the array is even, then the median is the average of the two middle numbers.
    """
```
Test Input:
```text
input: [1, 3, 2, 5]
```
Test Case:
```python
assert find_the_median([1, 3, 2, 5]) == 2.5
```

END OF EXAMPLES.
```

Figure 16: Few-shot prompt for the Calculator agent without the Python interpreter.

Example 1.

user message

```
<few-shot prompt>

Function:
```python
from typing import List


def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups of nested
parentheses. Your goal is to
    separate those group into separate strings and return the list of those.
    Separate groups are balanced (each open brace is properly closed) and not nested
within each other
    Ignore any spaces in the input string.
    """

```
Test Input:
```text
input: "()"
```
```

Figure 17: Message 1 of Example 1.

assistant message

```
Thought: The input string contains only one group of balanced parentheses, so the function
should return a list containing only the input string.
Test Case:
```python
assert separate_paren_groups("()") == ["()"]
```
```

Figure 18: Message 2 of example 1.

Example 2.

user message

```
<few-shot prompt>

Function:
```python
from typing import List


def separate_paren_groups(paren_string: str) -> List[str]:
    """ Input to this function is a string containing multiple groups of nested
parentheses. Your goal is to
    separate those group into separate strings and return the list of those.
    Separate groups are balanced (each open brace is properly closed) and not nested
within each other
    Ignore any spaces in the input string.
    """

```
Test Input:
```text
input: ""
```
```

Figure 19: Message 1 of Example 2.

assistant message

```
Thought: The input string is empty, so there are no separate groups of nested parentheses
to separate.
Test Case:
```python
assert separate_paren_groups("") == []
```
```

Figure 20: Message 2 of Example 2.

Example 3.

user message

```
<few-shot prompt>

Function:
```python
def move_one_ball(arr):
    """We have an array 'arr' of N integers arr[1], arr[2], ..., arr[N].The
    numbers in the array will be randomly ordered. Your task is to determine if
    it is possible to get an array sorted in non-decreasing order by performing
    the following operation on the given array:
        You are allowed to perform right shift operation any number of times.

    One right shift operation means shifting all elements of the array by one
    position in the right direction. The last element of the array will be moved to
    the starting position in the array i.e. 0th index.

    If it is possible to obtain the sorted array by performing the above operation
    then return True else return False.
    If the given array is empty then return True.

    Note: The given list is guaranteed to have unique elements.

    """
```
Test Input:
```text
input: [3, 1, 2]
```
```

Figure 21: Message 1 of Example 3.

assistant message

```
Thought: To solve this problem, I should first check if the array is already sorted in
non-decreasing order.
Code:
```python
arr = [3, 1, 2]
sorted_arr = sorted(arr)
sorted_arr == arr
```
```

Figure 22: Message 2 of Example 3.

tool message

```
Observation:
Out[1]: False
```

Figure 23: Message 3 of Example 3.

assistant message

```
Thought: The array is not already sorted, so I need to check if it's possible to obtain
the sorted array by performing right shift operations.
Code:
```python
n = len(arr)
sorted_arr = sorted(arr)
for i in range(n):
    arr = [arr[-1]] + arr[:-1]  # Perform right shift operation
    if arr == sorted_arr:
        print(True)
        break
else:
    print(False)
```
```

Figure 24: Message 4 of Example 3.

tool message

```
Observation:
True
```

Figure 25: Message 5 of Example 3.

assistant message

```
Thought: After performing the right shift operation, it is possible to obtain the sorted
array.
Test Case:
```python
assert move_one_ball([3, 1, 2]) == True
```
```

Figure 26: Message 6 of Example 3.

Example 4.

user message

```
<few-shot prompt>

Function:
```python
def simplify(x, n):
    """Your task is to implement a function that will simplify the expression
    x * n. The function returns True if x * n evaluates to a whole number and False
    otherwise. Both x and n, are string representation of a fraction, and have the
following format,
    <numerator>/<denominator> where both numerator and denominator are positive whole
numbers.

    You can assume that x, and n are valid fractions, and do not have zero as denominator.

    """
```
Test Input:
```text
input: ("2/5", "5/2")
```
```

Figure 27: Message 1 of Example 4.

assistant message

```
Thought: To simplify the expression x * n, I should first convert the fractions into
integers.
Code:
```python
from fractions import Fraction
x = "2/5"
n = "5/2"
x = Fraction(x)
n = Fraction(n)
print(x, n)
```
```

Figure 28: Message 2 of Example 4.

tool message

```
Observation:
2/5 5/2
```

Figure 29: Message 3 of Example 4.

assistant message

```
Thought: Now that I have the fractions as integers, I can multiply them and check if the
result is a whole number.
Code:
```python
result = x * n
result.is_integer()
```
```

Figure 30: Message 4 of Example 4.

tool message

```
Observation:
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
File <ipython-input-1-662cc8044a62>:2
      1 result = x * n
----> 2 result.is_integer()

AttributeError: 'Fraction' object has no attribute 'is_integer'
```

Figure 31: Message 5 of Example 4.

assistant message

```
Thought: It seems that the Fraction object does not have an is_integer method. I can
convert the result to a float and check if it is equal to its integer value.
Code:
```python
float(result) == int(result)
```
```

Figure 32: Message 6 of Example 4.

tool message

```
Observation:
Out[1]: True
```

Figure 33: Message 7 of Example 4.

assistant message

```
Thought: The result is True, which means the expression x * n evaluates to a whole number.
Test Case:
```python
assert simplify("2/5", "5/2") == True
```
```

Figure 34: Message 8 of Example 4.