

Fixing Large Language Models' Specification Misunderstanding for Better Code Generation

Zhao Tian
College of Intelligence and
Computing, Tianjin University
China
tianzhao@tju.edu.cn

Junjie Chen*
College of Intelligence and
Computing, Tianjin University
China
junjiechen@tju.edu.cn

Xiangyu Zhang
Department of Computer Science,
Purdue University
USA
xyzhang@cs.purdue.edu

Abstract—Code generation is to automatically generate source code conforming to a given programming specification, which has received extensive attention especially with the development of large language models (LLMs). Due to the inherent difficulty of code generation, the code generated by LLMs may not be aligned with the specification. Although thought-eliciting prompting techniques have been proposed to enhance the code generation performance of LLMs, producing correct understanding for complicated programming problems remains challenging, resulting in unsatisfactory performance. Also, some feedback-based prompting techniques have been proposed to fix incorrect code using error messages produced by test execution. However, when the generated code deviates significantly from the ground truth, they encounter difficulties in improving performance based on such coarse-grained information.

In this work, we propose a novel prompting technique, called μ FiX, to improve the code generation performance of LLMs by devising both sophisticated thought-eliciting prompting and feedback-based prompting and making the first exploration on their synergy. It first exploits test case analysis to obtain specification understanding and enables a self-improvement process to identify and refine the misunderstanding in the thought-eliciting prompting phase. μ FiX further fixes the specification understanding towards the direction reducing the gap between the provided understanding (from the first phase) and the actual understanding implicitly utilized by LLMs for code generation in the feedback-based prompting phase. By improving the understanding with μ FiX, the code generation performance of LLMs can be largely improved. Our evaluation on two advanced LLMs (ChatGPT and DeepSeek-Coder) with six widely-used benchmarks by comparing with 15 baselines, demonstrates the effectiveness of μ FiX. For example, μ FiX outperforms the most effective baseline with an average improvement of 35.62% in terms of Pass@1 across all subjects.

Index Terms—Code Generation, Large Language Models, Prompting Engineering

I. INTRODUCTION

Code generation aims to automatically generate source code conforming to a given programming specification (which is usually a natural language description). It can help reduce repetitive programming efforts and improve software development productivity. In recent years, code generation has received extensive attention from both academia and industry. In particular, with LLMs being rapidly developed, significant progress has been made in code generation, such

as ChatGPT [1] and DeepSeek-Coder [2]. The LLMs take the programming specification (i.e., prompt) as input and output the corresponding code solution, demonstrating notable advancements in code generation.

Despite their popularity, LLMs still suffer from some performance issues. That is, the generated code may be not aligned with the human-provided specification, especially when the programming logic is complicated [3]. For example, one state-of-the-art LLM, i.e., ChatGPT, generates code passing all test cases for only 16.33% of programming problems in a real-world benchmark [4]. The performance issues can negatively affect the practical use of LLMs, even slow down software development process and harm software quality. Hence, it is important to enhance the ability of LLMs in code generation.

Although fine-tuning strategies have been widely adopted to improve the code generation performance of LLMs, they are time-consuming and require a large amount of computing resources [5]–[8]. In recent years, prompting techniques have been proposed to achieve this goal in a plug-and-play manner [9]–[13]. Among them, thought-eliciting prompting is the most popular category. It aims to elicit LLMs to produce intermediate reasoning steps as the specification understanding for more accurate code generation. The typical thought-eliciting prompting techniques include CoT [9] (that elicits LLMs to produce intermediate natural language reasoning steps), Self-planning [13] (that guides LLMs to decompose the specification into a set of easy-to-solve sub-problems and produce code plans to facilitate code generation), SCoT [14] (that enhances CoT by utilizing program structures to build intermediate reasoning steps, thereby eliciting LLMs for code generation), and so on. Feedback-based prompting is another category of prompting techniques (such as Self-Debugging [15], Self-Edit [16], and Self-repair [17]), which leverages error messages produced by test execution to enable LLMs to fix incorrectly generated code.

Although these techniques have been studied extensively, their performance still needs to be improved. For example, the existing thought-eliciting prompting techniques have difficulties in producing correct understanding according to the (concise) specification, when facing with complicated programming problems [13], [18]. Moreover, they cannot identify or fix incorrect specification understanding that occurs before

*Junjie Chen is the corresponding author.

code generation, leading to inaccurate results. For feedback-based prompting, the existing techniques just utilize error messages produced by test execution to understand why the generated code is incorrect, which is too coarse-grained to identify root causes and thus leads to suboptimal performance. If the generated code deviates largely from the ground truth, it is quite hard for them to improve the performance based on error messages from such low-quality generated code [17]. In particular, the two categories play roles at different stages (the former working before test execution whereas the latter after) but there is no existing work exploring their synergy.

To improve the code generation performance of LLMs, we propose a novel prompting technique, called μ FiX (Misunderstanding FiXing), to overcome the aforementioned limitations. In particular, μ FiX is the first to explore the synergy of the above two categories, by devising both thought-eliciting prompting and feedback-based prompting. The key insight is that, by improving specification understanding in the thought-eliciting prompting phase via test case analysis, the effectiveness of the subsequent feedback-based prompting (and code generation) can be enhanced. Namely, even though LLMs may not generate correct code based on the specification understanding, it can make the generated code as close to the ground truth as possible, which can provide a greater chance for the subsequent feedback-based prompting to further improve the code generation performance.

In the thought-eliciting prompting phase, the key lies in improving LLMs’ understanding of the specification. We propose to achieve this by leveraging test cases inherently included as part of a specification. Note that such test cases are prevalent in practice, including those widely-studied datasets in code generation (e.g., HumanEval [19] and APPS [4]). Due to the intrinsic reasoning capabilities of LLM, if its understanding is correct, it shall be able to resolve a natural language request derived from the (embedded) test cases and produce expected outputs. In contrast, when the understanding is incorrect, by prompting LLM to fix the test outputs, we could improve the understanding, thereby enhancing the later code generation performance. This is analogous to how software developers use test case examples to understand complicated programming logic in practice [20], [21].

Although the understanding can be refined to infer the correct test outputs, it does not mean that the corresponding generated code can really pass the test cases in actual execution due to the gap between specification understanding and code generation. Specifically, specification understanding and code generation emphasize different aspects of abilities in LLMs, where the former relies on the reasoning ability of LLMs while the latter emphasizes the ability of translating the natural language description into the source code [22], [23]. Hence, when invoking LLMs to generate code, the actual understanding implicitly utilized by LLMs may be inconsistent with the provided understanding. μ FiX further designs feedback-based prompting to improve code generation performance (if the test execution fails). Instead of prompting LLMs to directly fix the generated code with coarse-grained error messages, μ FiX

prompts LLMs to understand the root cause (i.e., the above-mentioned gap) by comparatively analyzing the provided understanding and the actual understanding (obtained via code summarization [24]), and then adjust the natural language description for the specification understanding to enhance its accessibility to the code generation ability of LLMs. It is also a kind of fixing on specification understanding, which aims to make the understanding be better utilized by the code generation ability of LLMs.

We conducted extensive experiments to evaluate μ FiX on two state-of-the-art LLMs (i.e., ChatGPT [1] and DeepSeek-Coder [2]) based on six widely-used benchmarks. Our results show that μ FiX significantly outperforms all the 15 compared prompting techniques on both LLMs across all six benchmarks, demonstrating our idea for enhancing code generation performance by improving LLMs’ specification understanding. For example, the average improvement of μ FiX over all 15 compared techniques is 35.62%~80.11% in terms of Pass@1 (measuring the ratio of programming problems for which the generated code passes all evaluation test cases) across all subjects. Moreover, we constructed four variants of μ FiX for the ablation study. The results confirm the contribution of our thought-eliciting prompting and feedback-based prompting strategies in μ FiX.

We summarize our contributions in this work as follows:

- We propose a novel prompting technique, called μ FiX, to enhance code generation performance of LLMs by improving their specification understanding with both sophisticated thought-eliciting prompting and feedback-based prompting.
- We design a novel thought-eliciting prompting strategy, which exploits test case analysis to produce more accurate specification understanding and enables a self-improvement process to identify and refine the misunderstanding before generating code.
- We design a novel feedback-based prompting strategy that adjusts the understanding towards the direction reducing the gap between the provided understanding and the actual understanding implicitly utilized by LLMs for code generation.
- We conducted extensive experiments on two LLMs (i.e., ChatGPT and DeepSeek-Coder) across six widely-used benchmarks by comparing with 15 baselines, demonstrating the effectiveness of μ FiX for improving the code generation performance of LLMs.

II. AN ILLUSTRATING EXAMPLE

We use an example to illustrate why the state-of-the-art prompting techniques do not work well, motivating our idea. Figure 1 shows the (simplified) example sampled from the real-world HumanEval benchmark [19]. In this figure, the programming specification provides a natural language description (including a test case).

Recent work has shown that prompting techniques can obtain the understanding of the specification by exploiting the logical reasoning ability of LLMs, thereby improving their performance in code generation [13], [18]. Here, we employed two state-of-the-art thought-eliciting prompting techniques




<Programming Specification>		
# <Description>: def solve(N): """ Given a positive integer N, return the total sum of its digits in binary. Variables: @N integer. Constraints: 0 ≤ N ≤ 10000. Output: a string of binary number. """ # <Test Cases>: assert solve(1000) == "1"		
 CoT Incorrect Understanding # Chain-of-Thought: Let's think it step by step: 1. Convert the decimal number N to binary. 2. Calculate the sum of the binary digits. 3. Convert the sum of binary digits to its binary representation. 4. Return the result as a string representing the binary number. <hr/> # Code: 1 def solve(N): 2 binary_representation = bin(N) 3 binary_digits = binary_representation[2:] 4 binary_sum = 0 5 for digit in binary_digits: 6 binary_sum += int(digit) 7 binary_sum = bin(binary_sum) 8 return binary_sum[2:] <hr/> Self-repair # Repair: * 7 binary_sum = bin(binary_sum % 2)	 SCoT Incorrect Understanding # Structured Chain-of-Thought: Input: N: a positive integer Output: binary_sum: a string representing the total sum of N's digits in binary 1: Convert N to a string to iterate through its digits. 2: Initialize binary_sum as an empty string. 3: For each digit in the string representation of N: 4: Convert digit to its binary representation and append it to binary_sum. 5: Return binary_sum. <hr/> # Code: 1 def solve(N): 2 N_str = str(N) 3 binary_sum = "" 4 for digit in N_str: 5 binary_sum += bin(int(digit))[2:] 6 return binary_sum <hr/> Self-repair # Repair: * 5 binary_sum += format(int(digit), '04b')	 μFiX Incorrect Understanding # Initial Specification Understanding: The input is 1000. The output is "1". Analysis: 1000 is represented as 1111101000 in binary, and the sum of its digits in binary is 6. Therefore, the expected output is "1". <hr/> Refined Understanding # Refined Specification Understanding: The input is 1000. The output is "1". Analysis: The sum of its digits is 1. In binary, 1 is represented as "1". Therefore, the expected output is "1". <hr/> Adjusted Understanding # Adjusted Specification Understanding: The input is 1000. The output is "1". Analysis: The sum of its decimal digits is 1 (1 + 0 + 0 + 0). When converted to binary, the decimal sum 1 is represented as "1" in binary. Therefore, the expected output is "1". <hr/> Correct Code # Code: 1 def solve(N): 2 digits = str(N) 3 digit_sum = sum(int(digit) for digit in digits) 4 binary_representation = bin(digit_sum)[2:] 5 return binary_representation

Fig. 1. An example from HumanEval #84 with ChatGPT

(i.e., CoT [9] and SCoT [14]) to prompt the state-of-the-art LLM (i.e., ChatGPT [1]) for generating code with regard to this specification. As shown in Figure 1, both CoT and SCoT obtained the understanding of the specification by dividing it into several intermediate reasoning steps (i.e., chain of thought and structured chain of thought). Then, they integrated these detailed reasoning steps with the original specification as the prompt, facilitating ChatGPT for code generation.

However, we found that both obtained understandings are incorrect due to the complicated programming logic implied in the specification, leading to generating incorrect code. Moreover, both CoT and SCoT are incapable of checking the correctness of the specification understanding before code generation, further limiting their performance. This motivates the importance of improving specification understanding before code generation for a thought-eliciting prompting technique.

Subsequently, we employed the state-of-the-art feedback-based prompting technique, Self-repair [17], to fix the incorrect code generated by CoT and SCoT. It leverages error messages produced by test execution on the incorrect code to understand why it is incorrect and then prompts ChatGPT to fix the code accordingly. Unfortunately, both fixed code remained incorrect, due to the substantial deviation of the initially generated code from the ground truth. It is quite hard to improve the code generation performance solely based on error messages from such low-quality code. This motivates that a feedback-based prompting technique requires not only possessing the ability to improve incorrectly generated code but also starting with high-quality code.

The two categories of techniques operate at different stages (the former working before test execution whereas the latter after). The output of thought-eliciting prompting (i.e., code generated by LLMs enhanced by the specification understand-

ing) serves as the input of feedback-based prompting. That is, the effectiveness of the former can affect that of the latter, thereby affecting the code generation performance, but there is no existing work exploring this synergy. This motivates us to explore the synergy of these two categories for better code generation. Specifically, we can first improve the specification understanding for generating the code as close to the ground truth as possible in the thought-eliciting prompting phase, and then design a more effective feedback-based prompting strategy for generating more accurate code on the basis of high-quality code from the first phase. With this insight, we propose a novel prompting technique, called μ FiX, which leverages the test cases inherent in the specification to achieve this goal. With μ FiX, the improved specification understanding is indeed produced, and also the correct code is generated for this example in Figure 1.

A. Overview

We propose a novel prompting technique, called μ FiX, to improve the code generation performance of LLMs. μ FiX devises both sophisticated thought-eliciting prompting and feedback-based prompting, as the first exploration on the synergy of both categories of prompting techniques. The key insight is to utilize test cases (inherently included in specifications) to improve specification understanding, inspired by the practice where software developers often use test cases to understand complex programming logic. It is helpful to generate high-quality code (close to the ground truth) as the starting point for the feedback-based prompting phase to further improve LLM performance. In practice, test cases are commonly included in programming specifications [4], [19], and we also evaluated the influence of the number of test cases on the effectiveness of μ FiX in Section VI-B. If a specification

III. APPROACH

lacks test cases, existing work has demonstrated the effectiveness of LLMs in generating them [25], [26], thereby ensuring the feasibility of μFiX , as detailed in Section VI-B.

Figure 2 shows the overview of μFiX . It consists of two phases: (1) the thought-eliciting prompting phase (Section III-B), which emphasizes the analysis on test cases for producing specification understanding and enables a self-improvement process to refine the misunderstanding with the aid of test cases, and (2) the feedback-based prompting phase (Section III-C), which comparatively analyzes the provided understanding and the actual understanding implicitly utilized by LLMs for code generation and minimizes the gap between the two for better code generation. The latter is activated only if the generated code does not pass the test cases (used for obtaining the specification understanding) in actual execution.

For ease of understanding on our μFiX , we reuse the example (i.e., HumanEval #84) introduced in Section II to illustrate the details of μFiX .

B. Thought-Eliciting Prompting Phase

In the thought-eliciting prompting phase, μFiX takes the programming specification as input, and aims to output more accurate specification understanding by leveraging the test cases inherently included in the specification. On one hand, emphasizing test case analysis helps LLMs improve specification understanding because test cases contain more specific details that facilitate the understanding of complicated programming logic. On the other hand, test cases enable the self-improvement process to refine LLMs’ misunderstanding before generating code. In the self-improvement process, μFiX first checks the correctness of the specification understanding and then improves it (if it is a misunderstanding). Note that existing thought-eliciting prompting techniques cannot identify and refine misunderstanding before generating code due to overlooking the importance of such test cases. Overall, the thought-eliciting prompting phase in μFiX consists of three steps: initial understanding generation, correctness checking of the understanding, and misunderstanding fixing. These steps will be detailed in the following.

Initial understanding generation: As shown in Figure 1, existing thought-eliciting prompting techniques also take the programming specification (including test cases) as input, but they do not pay special attention to these test cases, thereby limiting their effectiveness. Instead, μFiX emphasizes test case analysis by providing a structured instruction to LLMs, which can help them utilize specific test cases to understand complicated programming logic. The <Initial Prompt> in Figure 3 shows the structured instruction activating test case analysis, which facilitates explaining the programming logic that processes the inputs provided in the test cases to get the expected output specified in the test cases. Then, μFiX enables LLMs to complete <?> in the instruction, producing initial specification understanding. For example, Figure 3 (<Initial Understanding>) shows the initial understanding produced by ChatGPT with test case analysis regarding the programming specification in Figure 1.

Correctness checking of the understanding: From Figure 3, we find that the initial understanding is incorrect even though the expected output shown in the understanding is correct. Actually, the expected output is correctly specified due to its leakage in the test cases as shown in the programming specification of Figure 1. With this specification misunderstanding, LLMs hardly generate correct code. Hence, with the aid of test cases, μFiX includes a mechanism to check the correctness of the understanding in advance, in order to have a chance for improving the understanding before generating code. Specifically, μFiX masks the expected output in the understanding and enables LLMs to infer it according to the analyzed logic in the understanding. If the inferred output differs from the expected output, it indicates an incorrect specification understanding. As shown in Figure 4, this mechanism is effective to identify the misunderstanding (produced in Figure 3).

Misunderstanding fixing: Once μFiX identifies a specification misunderstanding, it prompts LLMs to refine this misunderstanding by providing the corresponding instruction (e.g., “The above understanding is incorrect. Please provide the correct analysis...”). This process terminates until the refined understanding passes the checking mechanism or the number of refinements reaches the pre-defined threshold (denoted as N). Note that passing the checking mechanism cannot guarantee that LLMs completely understand the programming specification as the number of test cases equipped in the specification tend to be limited. However, at least, the refined understanding (as shown in Refined Specification Understanding of Figure 1) passing the checking mechanism is more accurate than that does not pass it. With such high-quality refined understanding, LLMs may directly generate correct code. If not, μFiX still provides a high-quality starting point for its subsequent feedback-based prompting phase for further improving code generation performance.

C. Feedback-based Prompting Phase

By integrating the more accurate specification understanding into the programming specification along with an additional instruction “Please implement the function in a markdown-style code block (pay attention to the specification understanding)”, μFiX prompts LLMs to generate code.

Although the understanding produced in the thought-eliciting prompting phase is more accurate regarding the test cases, it does not guarantee that the corresponding generated code will pass the test cases in actual execution. This is due to the gap between specification understanding and code generation, which emphasize different aspects of abilities in LLMs. That is, the actual understanding implicitly utilized by LLMs for code generation may be inconsistent with the provided understanding. For example, when prompting LLMs to generate code, they may miss some important contents in the provided understanding due to the natural-language-description style unfamiliar to this ability of LLMs.

Existing feedback-based prompting techniques leverage error messages produced by test execution to prompt LLMs for enhancing code generation [17]. This kind of information

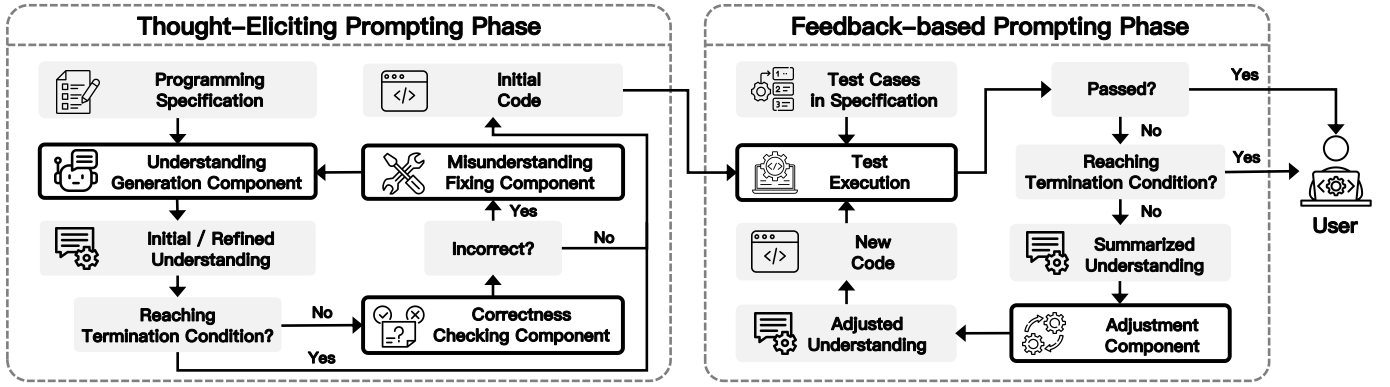


Fig. 2. Overview of μ FiX

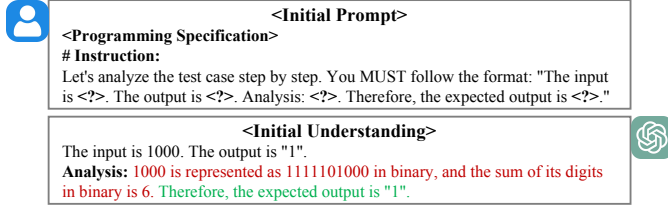


Fig. 3. An example of μ FiX in understanding generation

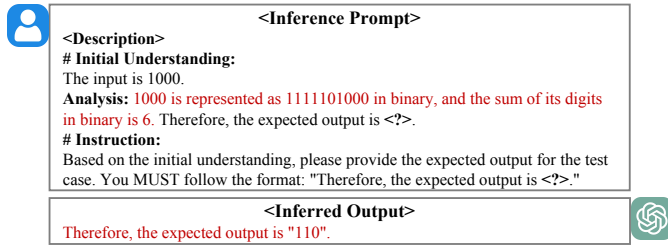


Fig. 4. An example of μ FiX in correctness checking of the understanding

is too coarse-grained to identify the root cause of incorrect code (especially when the generated code significantly deviates from the ground truth), leading to unsatisfactory performance. Different from them, μ FiX prompts LLMs to understand the root cause (i.e., the aforementioned gap) by comparatively analyzing the provided refined understanding and the actual understanding implicitly utilized by LLMs for code generation. Here, based on the symmetry between code generation (from natural language descriptions to code) and code summarization [27], [28] (from code to natural language descriptions), μ FiX estimates the actual understanding implicitly utilized by LLMs for code generation through code summarization (also called summarized understanding for ease of presentation). As shown in **<Summarization Prompt>** of Figure 5, μ FiX prompts LLMs to produce the summarized understanding (**<Summarized Understanding>** of Figure 5).

μ FiX then adjusts the natural language description of the refined specification understanding towards the direction reducing the gap, by exploiting the logical reasoning ability of LLMs. For example, some important contents missed by the summarized understanding can be highlighted or the unfamiliar natural-language-description style to LLMs can be improved accordingly through comparative analysis. Here, we design an adjustment prompt (**<Adjustment Prompt>**

of Figure 5) to enable LLMs to achieve the above goal, which consists of the generated code, test execution results, the refined understanding produced from the thought-eliciting prompting phase, summarized understanding, and an additional instruction guiding LLMs for adjusting understanding.

Taking Refined Specification Understanding of Figure 1 (the provided refined understanding) and **<Summarized Understanding>** of Figure 5 (the summarized understanding) as an example for illustration, the comparative analysis between them implies that LLMs just capture some keywords (e.g., *digits*, *binary*, and *sum*) but fail to understand the logic among keywords in the summarized understanding, which provides a direction to adjust the understanding. Indeed, the adjusted understanding emphasizes the correct computation logic with $(1+0+0+0)$, as shown in Adjusted Specification Understanding of Figure 1. With the adjusted understanding by μ FiX, ChatGPT successfully generates correct code for the programming specification, as shown in Figure 1. This phase is an iterative process and terminates until the generated code passes the corresponding test cases or the number of adjustments reaches the threshold (denoted as M).

In fact, the adjustment process can be also regarded as a kind of fixing on specification understanding, which aims to ensure that the understanding is being correctly utilized during the code generation. That is, the two phases in μ FiX involve complementary aspects of specification understanding fixing, and synergetically enhance code generation performance. Note that both fixing components are not simple regeneration. Similar to human-centric iterative improvement processes (e.g., debugging), they essentially provide counter-factual evidence (e.g., historical incorrect cases and their explanations) as feedback to LLMs. Analogous to human processes, they may not succeed potentially due to the limited ability of LLMs or insufficient feedback to LLMs. In the future, we will explore more effective LLMs in this aspect and more informative feedback so as to further enhance both fixing components.

IV. EVALUATION DESIGN

Our study addresses the following research questions (RQs).

- **RQ1:** How does μ FiX perform in improving the code generation performance of LLMs compared to the state-of-the-art prompting techniques?

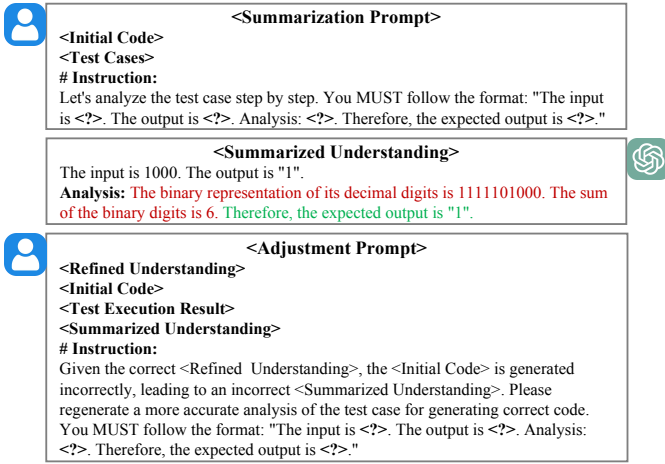


Fig. 5. An example of μ FiX in adjusting the understanding. For saving space, we show the refined understanding and adjusted understanding in Refined Specification Understanding and Adjusted Specification Understanding of Figure 1, respectively.

- **RQ2:** Does each main component in μ FiX contribute to the overall effectiveness?
- **RQ3:** How does μ FiX perform under different hyper-parameters' configurations?

A. Studied LLMs

Following the OpenAI's news [29], Codex's access was discontinued and GPT-3.5 has been recommended instead, and thus we selected GPT-3.5 (ChatGPT [1]) as the representative commercial LLM in our study following the existing studies [13], [17], [30]. Here, we did not choose GPT-4 due to its high cost. Following the existing studies [31]–[33], we selected DeepSeek-Coder [23] as the representative open-source LLM, since it has exhibited state-of-the-art effectiveness among open-source LLMs across multiple programming languages and various benchmarks in terms of coding capabilities [34]–[36]. Based on the two state-of-the-art LLMs in code generation, the generality of μ FiX can be investigated to some extent. Specifically, we used ChatGPT (version gpt-3.5-turbo-0613) via OpenAI's APIs, and DeepSeek-Coder (version DeepSeek-Coder-6.7B-Instruct) from Huggingface [37]. Note that LLM-generated code often includes natural-language text snippets, leading to compilation failures. Therefore, following the existing work [26], we employed a code sanitizer tool [38] to clean LLM-generated code.

B. Benchmarks

To sufficiently evaluate μ FiX, we adopted six widely-used datasets in our study, i.e., HumanEval [19], HumanEval+ [26], APPS [4], HumanEval-ET [39], APPS-ET [39], and MBPP-ET [39]. These datasets have been widely used in many existing studies to measure the performance of prompting techniques in code generation [13], [14], [40].

HumanEval has 164 human-written programming problems proposed by OpenAI. The specification for each problem consists of a function signature, a natural language problem description, and several test cases. Each problem has another set of test cases for evaluating the correctness of generated

code, which is called *evaluation test cases* here to distinguish with the test cases in the specification.

HumanEval+ extends HumanEval by automatically generating more evaluation test cases for each problem via the EvalPlus framework [26]. More evaluation test cases could determine the correctness of generated code more accurately.

APPS contains 10,000 programming problems collected from public programming competition platforms (e.g., LeetCode [41]), including 5,000 training data and 5,000 test data. The specification for each problem contains a natural language problem description and several test cases. Following the existing work [17], we randomly sampled 300 problems from test data according to the difficulty distribution, so as to balance evaluation cost and conclusion generality.

The existing work [39] extended the HumanEval and APPS benchmarks as *HumanEval-ET* and *APPS-ET* by constructing over 100 additional evaluation test cases per problem, respectively. These additional evaluation test cases cover more corner cases to enhance evaluation sufficiency on generated code. Regarding APPS-ET, we used the same 300 problems as APPS. Besides, we used *MBPP-ET*, an extended version of MBPP (that does not have additional evaluation test cases), by complementing evaluation test cases for each programming problem in a similar way. This dataset contains 974 programming problems, each with a specification comprising a natural language problem description and three test cases.

Note that μ FiX just utilizes the test cases in the specification for each problem. For these datasets, 81.57% of problems have at least three test cases in their corresponding specifications. The evaluation test cases are just used for assessing generated code following the practice in code generation [39].

C. Metrics

Following existing work [13], [39], we executed evaluation test cases to check correctness of generated code for each programming problem, and calculated Pass@ k and AvgPassRatio to measure the performance of LLMs in code generation.

Pass@ k measures the functional correctness of generated code on evaluation test cases. Given a programming problem, the LLM generates k code instances. The problem is considered solved if any instance passes all evaluation test cases. Pass@ k is the percentage of solved problems out of the total number of problems. As demonstrated by the existing work [10], developers tend to consider and evaluate one code instance produced by the used LLM, and thus we set $k = 1$ following the existing studies [10], [13], [18], [39], [42]. Note that Pass@1 is a more strict setting and thus improving it is challenging. Larger Pass@ k values mean better code generation performance.

AvgPassRatio measures the degree of correctness of generated code on evaluation test cases, differing from Pass@ k that considers whether the generated code is completely correct on evaluation test cases [4]. Both metrics are complementary to a large extent. AvgPassRatio calculates the ratio of passing evaluation test cases to all evaluation test cases for each problem, and then measures the average ratio across all problems. Larger AvgPassRatio values mean better code generation

performance. For ease of presentation in tables, we abbreviated *AvgPassRatio* as *APR*. More metrics (i.e., Pass@2, Pass@3, and CodeBLEU [43]) will be discussed in Section VII.

D. Compared Techniques

To evaluate μFiX sufficiently, we considered nine typical or state-of-the-art prompting techniques for comparison:

- **Zero-shot prompting** [19]: directly utilizes the original specification to prompt LLMs for code generation.
- **Few-shot prompting** [19]: enables LLMs to learn the relationship between specifications and code based on randomly selected <specification, code> examples. It concatenates these examples with the original specification to form a prompt, which is then fed to LLMs for code generation.
- **CoT prompting** [9]: elicits LLMs to produce intermediate reasoning steps as specification understanding. It incorporates the specification understanding into the original specification to form a prompt, which is then fed to LLMs for code generation. Following the existing work [14], [40], we applied CoT prompting in both zero-shot [44] and few-shot [9] settings. For ease of presentation, we call them **Zero-shot-CoT prompting** and **Few-shot-CoT prompting**.
- **Self-planning prompting** [13]: guides LLMs to decompose the specification into a set of easy-to-solve sub-problems and produce code plan by providing few-shot intent-to-plan examples. It incorporates the code plan into the original specification to form a prompt, which is then fed to LLMs for code generation.
- **SCoT prompting** [14]: enhances CoT prompting by utilizing program structures (i.e., sequence, branch, and loop structures) to produce intermediate reasoning steps.
- **Self-Debugging prompting** [15]: utilizes the runtime errors and test execution results to guide LLMs for fixing incorrectly generated code.
- **Self-Edit prompting** [16]: wraps error messages (produced by test execution) as supplementary comments, including test inputs, expected outputs, actual outputs, and runtime errors. Then, the supplementary comments serve as feedback to guide LLMs to fix incorrectly generated code.
- **Self-repair prompting** [17]: leverages error messages produced by test execution to enable LLMs to produce a short explanation of why the code failed. Then, the explanation guides LLMs to fix the incorrectly generated code.

In summary, Zero-shot and Few-shot prompting are typical prompting techniques widely used as baselines in code generation studies [14], [45]. Zero-shot-CoT and Few-shot-CoT are typical thought-eliciting prompting techniques, while Self-planning and SCoT are state-of-the-art techniques in this category. Self-Debugging, Self-Edit, and Self-repair are state-of-the-art feedback-based prompting techniques.

Note that μFiX is the first to integrate both thought-eliciting and feedback-based prompting techniques. For thorough evaluation, we also combined each of the two state-of-the-art thought-eliciting prompting techniques (Self-planning and SCoT) with each of the three state-of-the-art feedback-based prompting techniques (Self-Debugging, Self-Edit, and

Self-repair). For example, SCoT combined with Self-repair (referred to as **SCoT + Self-repair**) involves applying Self-repair to the code generated by SCoT. In total, we implemented 15 compared techniques as baselines. Given the input length limit of LLMs, we used the 4-shot setting for all few-shot baselines following the existing work [13], [46], [47].

V. RESULTS AND ANALYSIS

A. RQ1: Overall Effectiveness of μFiX

1) *Process*: To answer RQ1, we applied μFiX and 15 compared techniques to ChatGPT and DeepSeek-Coder. We then measured the effectiveness of each technique on 6 widely-used benchmarks in terms of Pass@1 and AvgPassRatio.

2) *Results*: Tables I and II show the effectiveness comparison results on ChatGPT and DeepSeek-Coder, respectively. We found that SCoT outperforms all thought-eliciting prompting baselines, while Self-repair excels compared to all feedback-based prompting baselines, confirming their effectiveness as state-of-the-art baselines in their corresponding categories. Also, each combination of thought-eliciting and feedback-based prompting techniques outperforms the corresponding individual techniques, which confirms the synergy between both categories, motivating our μFiX technique.

In particular, μFiX achieves the best effectiveness among all studied techniques, demonstrating its stable superiority in both metrics across all subjects (two LLMs with six benchmarks). Based on ChatGPT, μFiX significantly improves all compared techniques by 16.02%~45.30% and 11.84%~40.03% in terms of Pass@1 and AvgPassRatio on average across all six benchmarks, respectively. Based on DeepSeek-Coder, μFiX significantly improves them by 55.23%~114.92% and 16.83%~102.37% in terms of Pass@1 and AvgPassRatio, respectively. Furthermore, the *Wilcoxon Signed-Rank Test* [48] at the significance level of 0.05 confirms that all p-values are smaller than 5.14e-6, demonstrating the statistically significant superiority of μFiX over all compared techniques.

B. RQ2: Contribution of Each Main Component in μFiX

1) *Variants*: μFiX consists of two phases: thought-eliciting and feedback-based prompting phases. To investigate the contribution of each phase (including some key components in each phase), we created four variants of μFiX :

- $\mu\text{FiX}_{\text{woF}}$: we removed the feedback-based prompting phase from μFiX , i.e., it just uses the specification understanding produced in the thought-eliciting prompting phase to generate code. It can measure the effectiveness of the individual thought-eliciting prompting phase in μFiX and also reflect the contribution of the feedback-based prompting phase.
- $\mu\text{FiX}_{\text{SR}}$: we replaced the feedback-based prompting strategy in μFiX with the state-of-the-art Self-repair prompting strategy. It can investigate the effectiveness of our designed feedback-based prompting strategy.
- $\mu\text{FiX}_{\text{SCoT}}$: we replaced the thought-eliciting prompting strategy in μFiX with the state-of-the-art SCoT strategy. It can further investigate the effectiveness of our designed thought-eliciting strategy by complementing $\mu\text{FiX}_{\text{woF}}$.

TABLE I
EFFECTIVENESS COMPARISON ON CHATGPT IN TERMS OF PASS@1 (\uparrow) AND AVGPASSRATIO (\uparrow). APR IS SHORT FOR AVGPASSRATIO.

ChatGPT	HumanEval		HumanEval+		HumanEval-ET		MBPP-ET		APPS		APPS-ET	
Prompting Technique	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
Zero-shot	73.78%	79.14%	66.46%	68.66%	67.07%	82.69%	58.21%	71.94%	16.33%	27.98%	6.00%	23.93%
Few-shot	75.61%	79.77%	69.51%	70.81%	68.29%	83.04%	59.34%	73.34%	17.33%	24.45%	6.33%	20.37%
Zero-shot-CoT	74.39%	79.25%	67.68%	69.69%	67.07%	83.21%	59.14%	72.91%	18.00%	30.02%	6.00%	24.97%
Few-shot-CoT	76.83%	81.62%	70.73%	72.03%	68.90%	84.41%	59.34%	73.22%	20.33%	32.65%	7.00%	27.60%
Self-planning	78.66%	81.95%	70.73%	72.36%	70.73%	83.67%	62.01%	75.22%	21.33%	33.48%	8.33%	28.97%
SCoT	79.27%	83.97%	71.95%	73.66%	71.34%	85.54%	62.22%	77.36%	22.00%	34.07%	7.67%	29.56%
Self-Debugging	77.44%	81.52%	71.34%	72.94%	70.12%	84.18%	60.37%	75.69%	18.67%	31.83%	6.33%	27.36%
Self-Edit	76.83%	81.15%	69.51%	70.82%	68.29%	84.34%	61.19%	76.35%	20.67%	33.28%	7.00%	27.49%
Self-repair	80.49%	82.61%	74.39%	75.65%	72.56%	81.25%	64.07%	75.25%	23.00%	30.21%	8.00%	25.19%
Self-planning+Self-Debugging	80.49%	83.33%	72.56%	74.20%	72.56%	85.66%	62.01%	75.40%	21.33%	33.67%	8.33%	29.23%
Self-planning+Self-Edit	79.88%	83.25%	71.34%	72.92%	71.34%	84.10%	62.01%	75.31%	21.33%	33.81%	8.33%	29.36%
Self-planning+Self-repair	82.32%	84.83%	73.78%	75.70%	73.78%	86.56%	64.17%	75.55%	23.33%	37.19%	8.67%	31.65%
SCoT+Self-Debugging	81.10%	85.50%	73.17%	74.90%	71.95%	86.83%	63.14%	78.45%	22.00%	34.16%	7.67%	29.77%
SCoT+Self-Edit	79.88%	84.27%	72.56%	74.31%	71.95%	86.12%	63.96%	78.58%	22.33%	34.43%	7.67%	29.78%
SCoT+Self-repair	81.71%	84.48%	75.00%	76.28%	73.17%	84.66%	64.37%	79.21%	25.00%	38.38%	8.67%	33.05%
μ FiX	90.24%	91.59%	80.49%	82.16%	79.88%	91.66%	69.10%	83.07%	35.67%	47.72%	10.33%	38.81%

TABLE II
EFFECTIVENESS COMPARISON ON DEEPSEEK-CODER IN TERMS OF PASS@1 (\uparrow) AND AVGPASSRATIO (\uparrow). APR IS SHORT FOR AVGPASSRATIO.

DeepSeek-Coder	HumanEval		HumanEval+		HumanEval-ET		MBPP-ET		APPS		APPS-ET	
Prompting Technique	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
Zero-shot	76.22%	80.92%	72.56%	73.79%	69.51%	82.97%	55.75%	68.75%	4.00%	11.38%	1.00%	10.95%
Few-shot	78.05%	82.41%	73.17%	74.91%	70.73%	84.28%	56.78%	69.86%	4.33%	6.28%	1.33%	4.81%
Zero-shot-CoT	76.83%	80.86%	72.56%	73.80%	69.51%	82.68%	55.85%	68.78%	4.67%	11.77%	2.00%	10.83%
Few-shot-CoT	78.66%	82.48%	74.39%	75.65%	71.34%	83.49%	57.08%	70.55%	4.67%	10.53%	1.67%	10.29%
Self-planning	78.65%	82.11%	73.17%	74.40%	70.73%	83.45%	57.08%	70.05%	4.00%	8.92%	1.00%	7.64%
SCoT	78.05%	82.03%	73.78%	75.04%	70.73%	83.48%	58.01%	71.16%	4.33%	10.21%	1.33%	9.29%
Self-Debugging	77.44%	82.45%	73.17%	74.78%	70.73%	83.83%	56.88%	69.73%	4.67%	12.11%	1.33%	11.60%
Self-Edit	76.83%	81.97%	73.17%	74.40%	70.12%	84.32%	57.08%	71.52%	4.67%	12.22%	1.67%	11.70%
Self-repair	79.27%	83.08%	72.56%	74.55%	70.12%	85.34%	58.73%	73.52%	5.67%	12.68%	1.33%	10.41%
Self-planning+Self-Debugging	79.27%	82.72%	73.78%	75.01%	71.34%	84.55%	59.34%	72.50%	5.33%	11.05%	1.67%	9.76%
Self-planning+Self-Edit	79.88%	83.33%	74.39%	75.62%	71.95%	84.67%	58.83%	71.96%	5.67%	13.14%	1.67%	12.46%
Self-planning+Self-repair	79.88%	83.10%	74.39%	75.62%	71.95%	84.19%	59.34%	73.84%	6.33%	13.91%	1.33%	11.35%
SCoT+Self-Debugging	79.27%	83.25%	74.39%	76.26%	71.95%	84.70%	58.21%	71.81%	5.67%	11.87%	1.67%	10.88%
SCoT+Self-Edit	78.66%	82.64%	74.39%	75.65%	70.73%	83.91%	58.21%	71.74%	4.67%	11.45%	1.67%	10.56%
SCoT+Self-repair	80.49%	84.15%	73.78%	75.77%	71.34%	86.10%	59.45%	72.94%	6.67%	15.47%	1.67%	15.00%
μ FiX	83.54%	86.17%	78.66%	80.22%	75.00%	87.12%	63.35%	78.03%	14.00%	23.59%	5.00%	19.81%

- μ FiX_{woS}: we removed the self-improvement component in the thought-eliciting prompting phase from μ FiX. It can investigate the contribution of identifying and refining misunderstanding before code generation in μ FiX.

2) *Process*: Due to the limited computational resource and evaluation time cost, we selected ChatGPT as the representative LLM for the ablation study, as it achieves the best effectiveness with μ FiX (in Tables I and II). Similarly, we used ChatGPT for the experiments in Section VI. Specifically, we applied μ FiX and its four variants to ChatGPT respectively, and measured the effectiveness of each technique on 6 widely-used benchmarks in terms of Pass@1 and AvgPassRatio.

3) *Results*: Table III shows the comparison results among μ FiX and its four variants in terms of Pass@1 and AvgPassRatio. First, μ FiX_{woF} achieves superior effectiveness compared to all 15 baselines in terms of both metrics (except SCoT+Self-repair on APPS-ET in terms of AvgPassRatio). It demonstrates the effectiveness of our thought-eliciting prompting phase. μ FiX outperforms μ FiX_{SCoT} with average improvements of 12.03% and 10.74% in terms of Pass@1 and AvgPassRatio,

further confirming the effectiveness of μ FiX’s thought-eliciting prompting strategy, when combining with the same feedback-based prompting strategy (designed in μ FiX).

Second, μ FiX outperforms μ FiX_{woF} with average improvements of 10.25% in Pass@1 and 9.65% in AvgPassRatio, demonstrating the contribution of our feedback-based prompting phase. Moreover, μ FiX demonstrates superior effectiveness over μ FiX_{SR} with average improvements of 5.26% in Pass@1 and 5.40% in AvgPassRatio, further confirming the superiority of our feedback-based prompting strategy over the state-of-the-art Self-repair strategy, when combining with the same thought-eliciting prompting strategy (designed in μ FiX).

Third, μ FiX demonstrates superior effectiveness compared to μ FiX_{woS} with average improvements of 17.15% and 10.72% in terms of Pass@1 and AvgPassRatio. In addition to the fixing process in the feedback-based phase, μ FiX also performs another misunderstanding fixing through the self-improvement process in the thought-eliciting phase. The results confirm the necessity of the self-improvement process in μ FiX.

Furthermore, the *Wilcoxon Signed-Rank Test* [48] at the

TABLE III
COMPARISON BETWEEN μFiX AND ITS VARIANTS IN TERMS OF PASS@1 (\uparrow) AND AVGPASSRATIO (\uparrow)

Variant	HumanEval		HumanEval+		HumanEval-ET		MBPP-ET		APPS		APPS-ET	
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
$\mu\text{FiX}_{\text{woF}}$	85.37%	87.33%	76.83%	79.11%	77.44%	88.60%	66.32%	80.12%	27.67%	38.63%	9.00%	32.75%
$\mu\text{FiX}_{\text{SR}}$	85.98%	87.94%	78.05%	79.67%	78.05%	89.54%	68.07%	82.38%	30.67%	42.80%	10.00%	35.14%
$\mu\text{FiX}_{\text{SCoT}}$	83.54%	87.16%	76.22%	77.87%	74.39%	87.83%	65.81%	80.20%	26.33%	37.72%	9.33%	32.50%
$\mu\text{FiX}_{\text{woS}}$	81.10%	84.30%	73.78%	75.09%	72.56%	85.31%	64.78%	79.36%	24.33%	40.24%	8.67%	33.58%
μFiX	90.24%	91.59%	80.49%	82.16%	79.88%	91.66%	69.10%	83.07%	35.67%	47.72%	10.33%	38.81%

significance level of 0.05 confirms that all p-values are smaller than $1.39\text{e-}4$, demonstrating the statistically significant superiority of μFiX over all variants. Overall, each main component does make contributions to the overall effectiveness of μFiX .

C. RQ3: Influence of Hyper-parameters

1) *Setup*: μFiX involves three main hyper-parameters: N (the number of refinements), M (the number of adjustments), and the decoding temperature of LLMs (denoted as T). By default, we set $N=1$ and $M=1$ to balance the effectiveness and efficiency. Following the existing work [49], [50], we set $T=0.7$ for both LLMs. In this RQ, we investigated the performance of μFiX under different settings. Due to the evaluation cost, we considered $N=\{1, 2\}$, $M=\{1, 2\}$, and $T=\{0.6, 0.7, 0.8, 0.9, 1.0\}$.

2) *Results*: Table IV shows that increasing N and M enhances the effectiveness of μFiX . Specifically, $\mu\text{FiX}_{N=2, M=2}$ outperforms $\mu\text{FiX}_{N=1, M=1}$ with the average improvements of 4.46% and 4.03% in terms of Pass@1 and AvgPassRatio, respectively, across all the six benchmarks and both LLMs. However, larger N and M also incur more time and token costs. $\mu\text{FiX}_{N=2, M=2}$ takes 44.54% more time and 40.94% more tokens than $\mu\text{FiX}_{N=1, M=1}$. We will discuss the efficiency of our studied techniques in detail in Section VI-A. Hence, we used $N=1$ and $M=1$ as the default setting, as μFiX under this setting consistently outperforms all baselines across all benchmarks and both LLMs and has lower time and token costs than other settings with larger N and M , demonstrating the cost-effectiveness of μFiX under this setting.

We then investigated the influence of decoding temperature. Due to the space limit, we put the detailed results on our homepage [51]. For example, μFiX under all the studied settings for the decoding temperature, consistently outperforms the state-of-the-art SCoT + Self-repair, by achieving 16.02%~22.60% higher Pass@1 and 11.84%~24.36% higher AvgPassRatio averaging across all six benchmarks on ChatGPT. This demonstrates the stable superiority of μFiX under different decoding-temperature settings.

VI. DISCUSSION

A. Efficiency

We measured the time and token overhead on code generation. On average, the most efficient baseline (Zero-shot prompting) takes 5.78s for a programming task, while the most effective baseline (SCoT + Self-repair) and μFiX take 20.61s and 22.75s, respectively. The average token overheads for Zero-shot, SCoT + Self-repair, and μFiX are 0.29K, 3.31K,

and 3.44K, respectively. The time and token overhead of μFiX are slightly higher than those of SCoT + Self-repair. This is because μFiX involves a bit more LLM invocations compared to existing prompting techniques. For example, the state-of-the-art thought-eliciting technique (SCoT) involves two LLM invocations for a programming task, whereas our thought-eliciting strategy involves four invocations. Considering the significant effectiveness of μFiX , some extra cost is acceptable, illustrating its excellent balance of cost and effectiveness.

Recent studies [14], [25], [52] suggest that increasing LLM invocations can enhance code generation performance. Hence, we further conducted a comparison experiment under the same number of LLM invocations to demonstrate μFiX 's effectiveness more clearly. CodeT [25] is the state-of-the-art method to improve code generation by increasing LLM invocations. It generates more candidate code instances and ranks them by test execution. Such ranking strategies can be used to enhance thought-eliciting prompting due to their orthogonal effect [14], [45]. Hence, we combined CodeT with the state-of-the-art thought-eliciting technique (SCoT) to compare with $\mu\text{FiX}_{\text{woF}}$ under the same number of LLM invocations. Both techniques do not involve feedback-based prompting for a fair comparison. We used ChatGPT as the representative LLM.

Due to the space limit, we put detailed results on our homepage [51] and summarized the conclusions here. $\mu\text{FiX}_{\text{woF}}$ achieves 11.45% higher Pass@1 and 6.93% higher AvgPassRatio than SCoT averaging across all six benchmarks. SCoT + CodeT can also outperform SCoT, but just achieve 3.65% higher Pass@1 and 3.55% higher AvgPassRatio. The results clearly demonstrate the effectiveness of our μFiX .

B. Influence of Test Cases

First, we conducted an experiment to investigate the influence of the number of test cases used in μFiX . We set the number of test cases to $\{1, 2, 3+\}$, with the notation 3+ indicating the utilization of all available test cases in a programming specification (the default setting in μFiX). Note that we randomly selected the corresponding numbers of test cases from the whole set of test cases in each programming specification. If fewer test cases in the specification were available than desired, we used all available ones. We used ChatGPT as the representative LLM on three benchmarks (HumanEval, HumanEval+, and HumanEval-ET). Table V shows the effectiveness of μFiX with different numbers of test cases in terms of Pass@1 and AvgPassRatio. μFiX always exhibits better performance than the most effective baseline (SCoT + Self-repair), even with only one test case. As the number

TABLE IV
INFLUENCE OF HYPER-PARAMETERS N AND M IN TERMS OF PASS@1 (\uparrow) AND AVGPASSRATIO (\uparrow)

LLM	Configuration	HumanEval		HumanEval+		HumanEval-ET		MBPP-ET		APPS		APPS-ET	
		Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
ChatGPT	$\mu\text{FiX}_{N=1, M=1}$	90.24%	91.59%	80.49%	82.16%	79.88%	91.66%	69.10%	83.07%	35.67%	47.72%	10.33%	38.81%
	$\mu\text{FiX}_{N=2, M=2}$	90.85%	92.88%	81.10%	82.74%	80.49%	92.04%	69.61%	83.32%	39.00%	49.81%	11.33%	40.43%
DeepSeek-Coder	$\mu\text{FiX}_{N=1, M=1}$	83.54%	86.17%	78.66%	80.22%	75.00%	87.12%	63.35%	78.03%	14.00%	23.59%	5.00%	19.81%
	$\mu\text{FiX}_{N=2, M=2}$	84.76%	86.98%	79.88%	81.44%	76.22%	88.35%	65.71%	79.59%	16.33%	27.31%	5.33%	22.84%

TABLE V
INFLUENCE OF TEST CASES USED IN μFiX

Number	HumanEval		HumanEval+		HumanEval-ET	
	Pass@1	APR	Pass@1	APR	Pass@1	APR
SCoT+Self-repair	81.71%	84.48%	75.00%	76.28%	73.17%	84.66%
μFiX_{woT}	86.00%	88.66%	78.05%	80.17%	78.66%	90.47%
1	82.32%	85.98%	75.61%	77.19%	73.17%	87.32%
2	85.37%	87.04%	77.44%	79.12%	76.22%	87.52%
3+	90.24%	91.59%	80.49%	82.16%	79.88%	91.66%

of test cases increased, μFiX 's code generation performance improved, demonstrating the importance of test-case-driven specification misunderstanding fixing for code generation.

Second, although it is quite common that a few test cases are included as part of a specification, we need to consider the scenario when test cases are absent, further enhancing the generality of μFiX . To prove the concept, we utilized LLMs to automatically generate test cases for each programming specification (without test cases) following the existing work [25], [26]. We call this variant μFiX_{woT} . We investigated the effectiveness of μFiX_{woT} by taking ChatGPT as the representative model on the same three benchmarks as above. We set the number of generated test cases to 3 as most of the programming specifications include three test cases as mentioned in Section IV-B. From Table V, μFiX_{woT} significantly outperforms the most effective baseline (SCoT + Self-repair) by 5.61% and 5.64% in terms of Pass@1 and AvgPassRatio, respectively, which demonstrates the practical effectiveness of μFiX in such a scenario. Also, test-case quality could affect the effectiveness of μFiX . We manually analyzed some cases where μFiX_{woT} underperformed μFiX and found that LLM-generated test cases covered only one branch mentioned in the specification or even contained errors, leading to slightly worse effectiveness. In the future, we will design better test generation methods, such as incorporating test coverage to guide LLM-based test generation, to further enhance the effectiveness of μFiX . Besides, the resources may be limited in practice. In such scenarios, employing test prioritization to select higher-quality test cases may help ensure the effectiveness of μFiX . We regard this promising exploration as our future work.

C. Comparison with Agent-based Techniques

Recently, some agent-based techniques leveraging interactions among agents have been proposed to enhance LLMs' effectiveness in software development. They are actually orthogonal to μFiX to a large extent, and we could utilize μFiX to enhance individual agents for better code generation.

Nevertheless, we still investigated 4 state-of-the-art agent-based techniques (MetaGPT [53], Self-collaboration [10], ChatDev [54], and AgentCoder [55]) for comparison. They share the same high-level insight, which simulates the software development process by assigning roles to several LLM agents for collaboration in code generation. The difference among them mainly lies in that they assign different roles to LLM agents and design different strategies to solve the corresponding tasks. Due to the limited space, more details about these techniques can be found in their corresponding papers.

We used ChatGPT as the representative in this experiment due to the evaluation cost. Note that we used the default numbers of iterations for MetaGPT, Self-collaboration, ChatDev, and AgentCoder in this experiment, which are 3, 4, 10, and 5, respectively, and are all larger than the single iteration of μFiX . While this comparison may be unfavorable to μFiX , μFiX still outperforms the four agent-based techniques by 19.23%~37.20% and 13.48%~60.33% in terms of Pass@1 and AvgPassRatio, respectively, averaging across all the six benchmarks. Due to space limit, we put the detailed results on our project homepage [51]. The *Wilcoxon Signed-Rank Test* [48] at the significance level of 0.05 confirms that all p-values are smaller than $1.38\text{e-}3$, demonstrating the statistically significant superiority of μFiX over all the studied agent-based techniques. Particularly, as demonstrated in Section V-C, μFiX 's effectiveness can be further enhanced with additional iterations (i.e., increasing N and M). Among the four agent-based techniques, AgentCoder is the most effective and efficient but still takes an average of 211.75 seconds and 63.77K tokens per programming task, consuming 830.77% more time and 1753.78% more tokens than μFiX . Overall, μFiX exhibits significant superiority over these agent-based techniques.

D. Soundness of Our Checking Mechanism

Achieving optimality in semantic analysis is generally undecidable. Many existing techniques, e.g., search-based solutions in software engineering, are known to suffer from local optimum [56]. The uninterpretable nature of LLMs exacerbates this difficulty. Hence, there does not exist a decision procedure to check if an LLM correctly understands a specification, rendering sound checking mechanisms for LLMs' understanding intractable. Such similar limitations are general for all deep-learning-based methods. To relieve this challenge, in μFiX , we employed a checking mechanism *on the downstream code generation task*. Specifically, we checked the correctness of the generated code during the feedback-based prompting phase through the actual execution of test cases inherently included

in specifications. That is, the improvement of understanding is demonstrated by the better effectiveness in downstream code generation. Our study indeed confirms that this mechanism helps improve the effectiveness of code generation, empirically demonstrating that leveraging existing test cases in specifications can alleviate LLM hallucinations.

E. Exploration of LLM Combinations in μFiX

We explored the effectiveness of using different LLMs for understanding refinement and code generation via a preliminary experiment. Due to the inherent characteristics, DeepSeek-Coder’s reasoning capability is weaker than ChatGPT’s. When we used ChatGPT for understanding refinement and DeepSeek-Coder for code generation, the improvements are 1.55% and 1.01% in terms of Pass@1 and AvgPassRatio, respectively, averaging across HumanEval, HumanEval+, and HumanEval-ET, compared to using DeepSeek-Coder for both. This indicates the potential of μFiX with more appropriate LLMs in corresponding aspects, which can be regarded as our future work.

VII. THREATS TO VALIDITY

The first threat lies in the generalizability of experimental results. To mitigate this threat, we comprehensively selected benchmarks, metrics, baselines, and LLMs. Following previous studies [13], [14], [39], [40], we selected six widely-used benchmarks in code generation and employed two metrics for code correctness assessment. Besides, we used Pass@2, Pass@3, and CodeBLEU [43] for measuring code generation performance (although CodeBLEU suffers from some limitations [14], [39], [57]), and put the results on our homepage [51] due to space limitation. The conclusions are consistent. We also selected 9 typical or state-of-the-art prompting techniques (as well as 6 combined techniques) for comparison and conducted a comprehensive evaluation on two state-of-the-art LLMs (i.e., ChatGPT and DeepSeek-Coder). In the future, we will evaluate μFiX to improve LLMs’ code generation performance on more comprehensive benchmarks.

The second threat lies in the randomness involved in LLMs. On one hand, our large-scale study and consistent conclusions across all subjects can help reduce this threat. On the other hand, we repeated the experiment comparing μFiX with the most effective baseline (SCoT + Self-repair) on three benchmarks (HumanEval, HumanEval+, and HumanEval-ET) as the representative for three times due to the huge evaluation cost. The standard derivations for SCoT + Self-repair and μFiX are only 0.021 and 0.009 for Pass@1, demonstrating the robustness of our conclusions to a large extent. This further reduces the threat. Due to the limited space, we place the detailed results on our project homepage [51].

The third threat lies in the design of prompts in μFiX . We did not specially tune the natural-language-description format of prompts, and thus cannot ensure their optimality. We designed some similar prompts via paraphrasing (e.g., synonym replacement and active-passive sentence transformation), which did not affect μFiX ’s effectiveness much.

We will systematically investigate μFiX ’s robustness in the future. Furthermore, the prompting structure in μFiX mainly emphasizes the inputs and outputs of test cases, which are essential test-case elements regardless of specifications, also demonstrating generalizability.

VIII. RELATED WORK

Prompting techniques have been demonstrated effective to improve the code generation performance of LLMs in a plug-and-play manner [9]–[13]. In general, they can be divided into two main categories: thought-eliciting prompting techniques and feedback-based prompting techniques. The former aims to elicit LLMs to produce intermediate reasoning steps for more accurate code generation. We have introduced and compared CoT [9], Self-planning [13], and SCoT [14] techniques in Section IV. Besides, KPC [58] is a knowledge-driven prompting technique, which decomposes code generation into intermediate reasoning steps and utilizes fine-grained knowledge extracted from API documentation for code generation (especially in exception-handling code). The latter category of prompting techniques use error messages produced by test execution to enable LLMs to fix incorrectly generated code, such as SED [59], CodeRL [60], Self-Debugging [15], Self-Edit [16], and Self-repair [17]. Due to the evaluation cost, we did not select all these prompting techniques in our study but just selected some typical or state-of-the-art ones as baselines.

Different from the existing techniques, μFiX is the first to explore the synergy of both categories by devising both sophisticated thought-eliciting and feedback-based prompting. Its core of improving LLMs’ code generation performance is to fix LLMs’ specification misunderstanding in each phase.

IX. CONCLUSION

In this work, we propose a novel prompting technique μFiX to improve the code generation performance of LLMs. Different from the existing prompting techniques, μFiX devises both sophisticated thought-eliciting prompting and feedback-based prompting, and explores their synergy. Our thought-eliciting prompting strategy in μFiX exploits test case analysis and the misunderstanding fixing process to obtain more accurate specification understanding. Then, our feedback-based prompting strategy further fixes the understanding to reduce the gap between the provided refined understanding (from the first phase) and the actual understanding implicitly utilized by LLMs for code generation. We conducted an extensive study on two advanced LLMs with six widely-used benchmarks, demonstrating the superiority of μFiX over the state-of-the-art prompting techniques.

ACKNOWLEDGEMENTS

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFB4506300), and the National Natural Science Foundation of China (Grant Nos. 62322208, 12411530122).

REFERENCES

- [1] OpenAI, “Chatgpt: Optimizing language models for dialogue.” <https://openai.com/blog/chatgpt>, 2022.
- [2] D. AI, “Deepseek coder: Let the code write itself.” <https://github.com/deepseek-ai/DeepSeek-Coder>, 2024.
- [3] Y. Ma, Y. Yu, S. Li, Y. Jiang, Y. Guo, Y. Zhang, Y. Xie, and X. Liao, “Bridging code semantic and llms: Semantic chain-of-thought prompting for code generation,” *arXiv preprint arXiv:2310.10698*, 2023.
- [4] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, “Measuring coding challenge competence with apps,” *arXiv preprint arXiv:2105.09938*, 2021.
- [5] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, “Challenges and applications of large language models,” *arXiv preprint arXiv:2307.10169*, 2023.
- [6] Z. Tian, J. Chen, and Z. Jin, “Code difference guided adversarial example generation for deep code models,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 850–862.
- [7] Z. Tian, J. Chen, and X. Zhang, “On-the-fly improving performance of deep code models via input denoising,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 560–572.
- [8] Z. Tian, H. Shu, D. Wang, X. Cao, Y. Kamei, and J. Chen, “Large language models for equivalent mutant detection: How far are we?” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1733–1745.
- [9] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.
- [10] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatgpt,” *arXiv preprint arXiv:2304.07590*, 2023.
- [11] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan, “Improving chatgpt prompt for code generation,” *arXiv preprint arXiv:2305.08360*, 2023.
- [12] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE’23)*, 2023.
- [13] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, “Self-planning code generation with large language model,” *arXiv preprint arXiv:2303.06689*, 2023.
- [14] J. Li, G. Li, Y. Li, and Z. Jin, “Structured chain-of-thought prompting for code generation,” *arXiv preprint arXiv:2305.06599*, 2023.
- [15] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” *arXiv preprint arXiv:2304.05128*, 2023.
- [16] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, “Self-edit: Fault-aware code editor for code generation,” *arXiv preprint arXiv:2305.04087*, 2023.
- [17] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, “Demystifying gpt self-repair for code generation,” *arXiv preprint arXiv:2306.09896*, 2023.
- [18] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, “Clarifygpt: Empowering llm-based code generation with intention clarification,” *arXiv preprint arXiv:2310.10996*, 2023.
- [19] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [20] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [21] S. Gulwani, “Programming by examples: Applications, algorithms, and ambiguity resolution,” in *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8*. Springer, 2016, pp. 9–14.
- [22] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [23] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, “Deepseek-coder: When the large language model meets programming—the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.
- [24] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, “Automatic semantic augmentation of language model prompts (for code summarization),” in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 1004–1004.
- [25] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, “Codet: Code generation with generated tests,” in *The Eleventh International Conference on Learning Representations*, 2022.
- [26] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *arXiv preprint arXiv:2305.01210*, 2023.
- [27] T. Ahmed and P. Devanbu, “Few-shot training llms for project-specific code-summarization,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.
- [28] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” *arXiv preprint arXiv:2102.04664*, 2021.
- [29] OpenAI, “Openai kills its codex code model, recommends gpt3.5 instead,” <https://the-decoder.com/openai-kills-code-model-codex/>, 2023.
- [30] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, “Is chatgpt the ultimate programming assistant—how far is it?” *arXiv preprint arXiv:2304.11938*, 2023.
- [31] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, “Opencodeinterpreter: Integrating code generation with execution and refinement,” *arXiv preprint arXiv:2402.14658*, 2024.
- [32] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, “Exploring and evaluating hallucinations in llm-powered code generation,” *arXiv preprint arXiv:2404.00971*, 2024.
- [33] S. Dou, Y. Liu, H. Jia, L. Xiong, E. Zhou, J. Shan, C. Huang, W. Shen, X. Fan, Z. Xi *et al.*, “Stepcoder: Improve code generation with reinforcement learning from compiler feedback,” *arXiv preprint arXiv:2402.01391*, 2024.
- [34] EvalPlus, “Evalplus leaderboard,” <https://evalplus.github.io/leaderboard.html>, 2024.
- [35] T. Team, “Coding llms leaderboard,” <https://leaderboard.tabbyml.com/>, 2024.
- [36] W. Wang, C. Yang, Z. Wang, Y. Huang, Z. Chu, D. Song, L. Zhang, A. R. Chen, and L. Ma, “Testeval: Benchmarking large language models for test case generation,” <https://llm4softwaretesting.github.io/>, 2024.
- [37] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, “Huggingface’s transformers: State-of-the-art natural language processing,” *arXiv preprint arXiv:1910.03771*, 2019.
- [38] EvalPlus, “Code sanitizer,” <https://github.com/evalplus/evalplus?tab=readme-ov-file#code-post-processing>, 2024.
- [39] Y. Dong, J. Ding, X. Jiang, Z. Li, G. Li, and Z. Jin, “Codescore: Evaluating code generation by learning code execution,” *arXiv preprint arXiv:2301.09043*, 2023.
- [40] X.-Y. Li, J.-T. Xue, Z. Xie, and M. Li, “Think outside the code: enhancing boosts large language models in code generation,” *arXiv preprint arXiv:2305.10679*, 2023.
- [41] LeetCode, <https://leetcode.com>, 2023.
- [42] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Y. Zhuo, and T. Chen, “Chain-of-thought in neural code generation: From and for lightweight language models,” *arXiv preprint arXiv:2312.05562*, 2023.
- [43] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.
- [44] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.
- [45] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, “Aecoder: Utilizing existing code to enhance code generation,” *arXiv preprint arXiv:2303.17780*, 2023.
- [46] M. Liu, N. Pinckney, B. Khailany, and H. Ren, “Verilogeval: Evaluating large language models for verilog code generation,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.
- [47] X. Jiang, Y. Dong, Z. Jin, and G. Li, “Seed: Customize large language models with sample-efficient adaptation for code generation,” *arXiv preprint arXiv:2403.00046*, 2024.
- [48] F. Wilcoxon, S. Katti, R. A. Wilcox *et al.*, “Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test,” *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.
- [49] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, “Self-consistency improves chain of thought reasoning in language models,” *arXiv preprint arXiv:2203.11171*, 2022.

- [50] X. Wan, R. Sun, H. Nakhost, H. Dai, J. M. Eisenschlos, S. O. Arik, and T. Pfister, "Universal self-adaptive prompting," *arXiv preprint arXiv:2305.14926*, 2023.
- [51] μ FiX, <https://github.com/tianzhaotju/muFiX>, 2024.
- [52] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang, "Learning to construct better mutation faults," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [53] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.
- [54] C. Qian, X. Cong, C. Yang, W. Chen, Y. Su, J. Xu, Z. Liu, and M. Sun, "Communicative agents for software development," *arXiv preprint arXiv:2307.07924*, 2023.
- [55] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.
- [56] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," *Advances in Neural Information Processing Systems*, vol. 32, 2019.
- [57] A. Eghbali and M. Pradel, "Crystalbleu: precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [58] X. Ren, X. Ye, D. Zhao, Z. Xing, and X. Yang, "From misuse to mastery: Enhancing code generation with knowledge-driven ai chaining," *arXiv preprint arXiv:2309.15606*, 2023.
- [59] K. Gupta, P. E. Christensen, X. Chen, and D. Song, "Synthesize, execute and debug: Learning to repair for neural program synthesis," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 685–17 695, 2020.
- [60] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coder1: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.