Revisit Self-Debugging with Self-Generated Tests for Code Generation

Xiancai Chen[1,2*], Zhengwei Tao[1,2], Kechi Zhang[1,2], Changzhi Zhou[3], Xinyu Zhang[3]
Wanli Gu[4], Yuanpeng He[1,2], Mengdi Zhang[4], Xunliang Cai[4], Haiyan Zhao[1,2†], Zhi Jin[1,2†]
[1]Key Laboratory of High Confidence Software Technology (PKU), MOE, China
[2]School of Computer Science, Peking University
[3]Beijing Institute of Technology, [4]Meituan
xiancaich@stu.pku.edu.cn, {zhhy.sei,zhijin}@pku.edu.cn

Abstract

Large language models (LLMs) have demonstrated significant advancements in code generation, yet they still face challenges when tackling tasks that extend beyond their basic capabilities. Recently, the concept of self-debugging has been proposed as a way to enhance code generation performance by leveraging execution feedback from tests. However, the availability of high-quality tests in real-world scenarios is often limited. In this context, self-debugging with self-generated tests emerges as a promising solution, though its limitations and practical potential have not been fully explored. To address this gap, we investigate the efficacy of self-debugging in code generation tasks. We propose and analyze two distinct paradigms for the self-debugging process: post-execution and in-execution self-debugging. Our findings reveal that post-execution self-debugging struggles with the test bias introduced by self-generated tests, which can lead to misleading feedback. In contrast, in-execution self-debugging enables LLMs to mitigate this bias and leverage intermediate states during program execution. By focusing on runtime information rather than relying solely on potentially flawed self-generated tests, this approach demonstrates significant promise for improving the robustness and accuracy of LLMs in code generation tasks.

1 Introduction

Large language models (LLMs) have made significant strides in the field of code generation, but still face challenges to perform complex programming tasks beyond their basic capabilities (Jain et al., 2024). Such tasks demand LLMs to comprehend requirements in natural language and produce programs that behave as expected. Recently, Self-Debugging has emerged as a promising method to boost the performance of LLMs in code generation (Chen et al., 2024b; Jiang et al., 2023; Zhong et al., 2024), which allows models to iteratively refine their outputs through execution with *pre-defined oracle tests*. A critical limitation arises in real-world software development scenarios: those oracle tests are not readily available for each programming task, posing a significant challenge for the practical application of this method.

To address this challenge, recent studies have introduced *self-generated tests* into the self-debugging process (Shinn et al., 2023; Huang et al., 2023; Ridnik et al., 2024). Beyond leveraging intrinsic capabilities (Huang et al., 2024; Madaan et al., 2023; Chen et al., 2024b), execution feedback from *self-generated tests* could also serve as additional signals to help LLMs identify defects in their code generation. This approach helps reduce reliance on external feedback from humans or stronger models. However, it remains unclear whether self-debugging using self-generated tests is a reliable and scalable solution to enhance code generation. Reflexion (Shinn et al., 2023) leverages feedback from generated tests to debug the code but triggers the repair using hidden oracle tests. AlphaCodium (Ridnik et al., 2024) allows LLMs to generate tests with supervision of public oracle tests. The improvements observed using oracle tests do not accurately demonstrate the true self-debugging capabilities of LLMs. This highlights the need for a more comprehensive and transparent evaluation to better understand the inherent debugging potential with execution feedback from tests.

In this work, we formally define two distinct paradigms for doing this: **post-execution and in-execution self-debugging**. The conventional post-execution method operates binary feedback from self-testing—pass or fail labels obtained through testing with self-generated tests, susceptible to test bias as models must infer defects solely from potentially flawed tests. We propose in-execution self-

[*]Work done during internship at Meituan.
[†]Corresponding Author.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 18003–18023

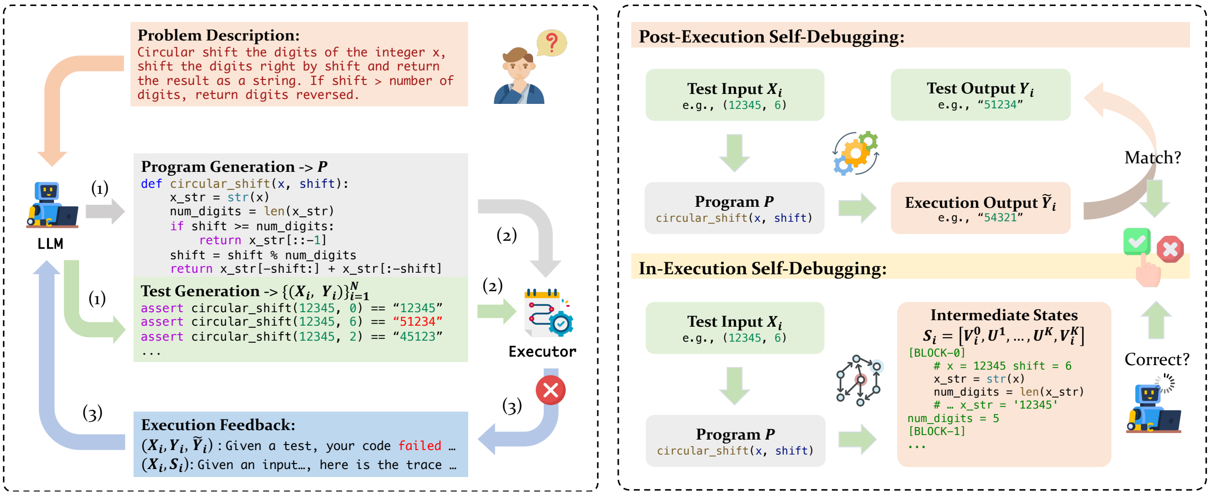

Figure 1: **Overview of self-debugging with execution feedback from self-generated tests.** (1) The model generates an initial program along with a suite of tests, based on the problem description. (2) The program is executed by an executor on the self-generated tests. (3) The feedback from execution is then utilized by the model to produce a revised version of the program.

debugging, a novel paradigm that allows models to analyze intermediate states during program execution without dependency on post-execution signals. This highlights a key gap: existing methods conflate in-execution traces with post-execution feedback (Zhong et al., 2024; Ni et al., 2024; Bouzenia et al., 2023), potentially failing to isolate the model from unreliable test verdicts.

We investigate the efficacy of both paradigms with self-generated tests applied to four frontier LLMs. Experimental results across both basic and competitive code generation tasks reveal key insights. We find that the post-execution method struggles with simpler tasks but shows potential for improvement on more challenging ones. The inconsistency across problem levels is attributed to the bias introduced by self-generated tests, which refers to the misalignment between self-testing labels and true labels of the programs, resulting in a critical bottleneck for the post-execution method. In-execution self-debugging transcends this limitation by solely focusing on the intermediate states during the program execution, reducing bias susceptibility. It consistently outperforms post-execution method, demonstrating its promise for advancing self-debugging capabilities in LLMs.

Our work is the first to systematically dissect the self-testing bias in self-debugging and propose a paradigm shift from post-execution label validation to in-execution state reasoning, thereby reducing dependency on unreliable test outputs. We summarize our contributions as follows:

- We formally define and evaluate two distinct paradigms: post-execution and in-execution self-debugging, providing a comprehensive analysis of their strengths and limitations.
- We identify the critical issue of test bias in self-generated tests, demonstrating how it undermines post-execution methods.
- We propose in-execution self-debugging as a robust alternative, leveraging intermediate runtime states to reduce bias and improve debugging accuracy.

Through our study, we aim to shed light on the practicality of self-debugging with self-generated tests, contributing valuable insights into the future development of LLMs in code generation tasks.

2 Self-Debugging with Self-Generated Tests

We focus on evaluating the self-debugging capabilities of LLMs through execution on their self-generated tests. Figure 1 provides a comprehensive overview of this process. Given a problem with a natural language specification, the language model (denoted as M) generates an initial program P along with a suite of tests $\{(X_i, Y_i)\}_{i=1}^N$, where X_i represents the input and Y_i represents the expected output for the i-th test. To enhance the model's debugging performance beyond its intrinsic reasoning capability, we utilize execution feedback as an additional signal to help the model identify defects in its generated program according to the problem description. Specifically, we employ an executor (denoted as E) to run the program on the test suite and collect execution information as feedback. This feedback is then used to guide the model in refining the program.

There are various implementations for utilizing execution feedback, which we categorize into two distinct paradigms: **post-execution and in-execution self-debugging**. These paradigms differ in the type of information employed during the self-debugging process. Post-execution information refers to content obtained after the program's execution, such as execution outputs or error messages. In contrast, in-execution information refers to content observed during the program's execution, providing finer-grained insights into its behavior. We now formally define these paradigms.

Post-Execution Self-Debugging. The paradigm leverages information obtained after the actual execution of the program. A widely adopted implementation involves comparing the execution output with the expected output (Olausson et al., 2024; Wang et al., 2024; Dong et al., 2023; Madaan et al., 2023; Zhang et al., 2023a; Chen et al., 2024b; Jiang et al., 2024b), as illustrated in Figure 1. Given an initial program P and a set of generated tests $\{(X_i, Y_i)\}_{i=1}^{N}$, an executor E processes each input X_i, yielding the corresponding execution output:

$$\tilde{Y}_i = \mathrm{E}(P, X_i), i \in [1, N] \quad (1)$$

The executor automatically assesses whether the execution output $\tilde{Y}_i$ aligns with the expected output Y_i to determine the success of the i-th test. If a discrepancy or error is detected, the test is marked as failed. The system then utilizes the failed test (X_i, Y_i), the execution output $\tilde{Y}_i$, and any associated error messages to repair the program. This process encourages the model M to generate a revised version of the program, denoted as:

$$\tilde{P} = \mathrm{M}(P, X_i, Y_i, \tilde{Y}_i) \quad (2)$$

In-Execution Self-Debugging. Post-execution self-debugging typically overlooks the runtime information of the program, which can provide valuable insights for program repair and enhancement. To address this limitation, in-execution self-debugging leverages feedback from the intermediate states during program execution (Nye et al., 2021; Ni et al., 2024; Bouzenia et al., 2023). Formally, a program's execution trace T_i can be divided into multiple basic units, denoted as $T_i = [U^1, U^2, ..., U^K]$, where U^k represents the k-th unit and K is the total number of units in the execution trace. Each unit is defined as a sequence of statements and expressions with a single entry and a single exit point. Given an input X_i, $i \in [1, N]$, the executor E initializes the input as the variable set $V_i^0 = X_i$ and passes it through the first unit U^1. The execution updates the variable set to $V_i^1 = \mathrm{E}(U^1, V_i^0)$, where V_i^1 denotes the set of variables after executing unit U^1. This process is repeated iteratively, with the executor traversing each subsequent unit U^k until the program execution is complete:

$$V_i^k = \mathrm{E}(U^k, V_i^{k-1}), k \in [1, K] \quad (3)$$

The sequence of intermediate states represented as $S_i = [V_i^0, U^1, V_i^1, ..., U^K, V_i^K]$, provides a detailed view of how the program behaves over time. By analyzing this runtime information, the model M is able to identify potential issues within specific units and refine the program accordingly. Notably, in-execution self-debugging excludes any post-execution signals (e.g., negative labels and corresponding test outputs), focusing solely on the dynamic behavior of the program during execution. This process results in the revised version of the program, denoted as:

$$\tilde{P} = \mathrm{M}(P, X_i, S_i) \quad (4)$$

3 Experiments

In this section, we comprehensively evaluate the self-debugging capabilities of a wide range of LLMs using self-generated tests on code generation benchmarks covering diverse levels of programming problems. We carry out experiments to answer the following research questions:

RQ1: *Are model-generated tests reliable enough? How does self-debugging perform leveraging post-execution information from these tests?*

RQ2: *Is the performance of post-execution self-debugging consistent on more complex tasks? If not, what factors contribute to the discrepancy?*

RQ3: *How does in-execution self-debugging perform compared with the post-execution method?*

3.1 Experimental Setup

Producing high-quality tests poses significant challenges for LLMs, as it necessitates a comprehensive understanding of natural language specifications and advanced code reasoning capabilities (Gu et al., 2024; Chen et al., 2024a). Therefore, we investigate the research questions with four advanced chat models: **LLaMA-3-70B-Instruct** (Dubey et al., 2024), **Qwen2.5-Coder-7B-Instruct** (Hui et al., 2024), the API-served **GPT-4o-2024-05-13** and **Claude-3.5-Sonnet**. We select three code generation benchmarks—**HumanEval+** (Chen et al., 2021; Liu et al., 2023), **MBPP+** (Austin et al., 2021; Liu et al., 2023), and **LiveCodeBench** (Jain et al., 2024), covering both basic and competitive* programming problems to comprehensively evaluate the efficacy of self-debugging.

We employ a greedy decoding strategy (with a temperature of zero) across all generation phases of self-debugging to ensure deterministic and consistent results. We design prompts for the initial program generation to ensure that no additional information is introduced by subsequent prompts for program repair. This premise is crucial for us to concentrate on investigating the true self-debugging capabilities of LLMs (Huang et al., 2024). To generate a test suite for each problem, we prompt the model to produce $n = 10$ diverse and extensive tests based on the corresponding natural language specification in a zero-shot manner. We discuss the effect of the varying n in Appendix C. For a detailed overview of the prompts used in our experiments, please refer to Appendix F.

3.2 RQ1: Post-Execution Self-Debugging Struggles with Potentially Flawed Tests

In this subsection, we evaluate the model-generated tests on basic tasks, and then examine the performance of self-debugging utilizing post-execution information from these tests, which is consistent with the conventional implementations. The correctness of a program is determined by comparing the actual output with the expected output for each test case. If the generated program manages to pass all tests, the iterative process terminates, and no further debugging is conducted.

LLMs Cannot Yet Produce Reliable Test Suites Even for Basic Tasks. To better understand the reliability of tests generated by the model itself, we employ program contracts and canonical solutions provided by HumanEval and MBPP to evaluate the validity of test inputs and outputs respectively. For detailed illustration for the validation, please refer to Appendix B. We also calculate the overall accuracy for the test suites. Figure 2 summarizes the results. GPT-4o and Claude-3.5-Sonnet demonstrate superior capability in producing high-quality tests compared to others, yet they remain prone to generating flawed tests. While models can often generate plausible inputs, they struggle to accurately predict the corresponding outputs, which are critical for validating program behavior. Notably, there is a significant gap between the accuracy of the test suites and the accuracy when we consider the tests individually. This suggests that post-execution information from self-generated tests has a high chance of providing unreliable feedback to the self-debugging process.

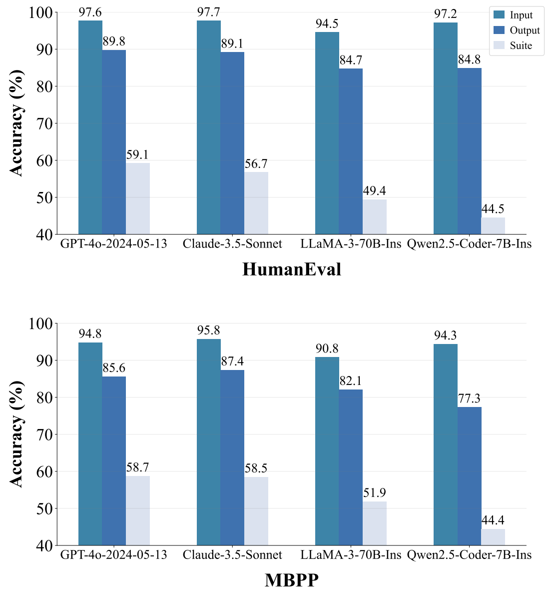

Figure 2: Accuracy of self-generated tests on HumanEval and MBPP. Test **input & output** are evaluated case-by-case; A test **suite** is deemed valid if all outputs within the suite are correct.

Post-Execution Self-Debugging Struggles on Basic Tasks. To assess the impact of the potentially flawed tests, we consider two different types of feedback derived from post-execution results. The first type is the binary correct *label*, which simply indicates the correctness of the previous program. If the program is incorrect, an instruction for repair is provided to the model. The second type is the *detail* of the failure, which includes the test input, expected output, and execution output. In cases where the program raises an exception during execution, the error message is incorporated into the detail in place of the execution output. We conduct experiments on problems from the Hu-

*In this work, we regard problems in HumanEval, MBPP as basic programming problems, and those in LiveCodeBench as competitive ones according to overall complexity and difficulty.

Table 1: Results of **post-execution self-debugging** (post-exec.) with *self-generated tests*. The values highlighted in red are declines while those in green are increases relative to the initial generation (w/o *self-debug*).

Model	Method	#Iter	HumanEval+	MBPP+	LiveCodeBench			
					easy	medium	hard	overall
GPT-4o-2024-05-13	w/o *self-debug*	0	87.8	76.5	89.3	33.1	6.0	46.0
	post-exec. w/ *label*	1	$87.2^{-0.6}$	$76.7^{+0.2}$	$89.9^{+0.6}$	$41.1^{+8.0}$	$6.0^{+0.0}$	$49.3^{+3.3}$
		2	$86.6^{-1.2}$	$77.5^{+1.0}$	$89.9^{+0.6}$	$40.0^{+6.9}$	$6.9^{+0.9}$	$49.1^{+3.1}$
	post-exec. w/ *detail*	1	$84.1^{-3.7}$	$76.2^{-0.3}$	$85.5^{-3.8}$	$36.0^{+2.9}$	$8.6^{+2.6}$	$46.4^{+0.4}$
		2	$85.4^{-2.4}$	$76.5^{+0.0}$	$87.4^{-1.9}$	$38.3^{+5.2}$	$8.6^{+2.6}$	$48.0^{+2.0}$
Claude-3.5-Sonnet	w/o *self-debug*	0	89.0	77.0	93.1	48.0	16.4	55.8
	post-exec. w/ *label*	1	$88.4^{-0.6}$	$77.8^{+0.8}$	$89.9^{-3.2}$	$49.1^{+1.1}$	$17.2^{+0.8}$	$55.3^{-0.5}$
		2	$86.6^{-2.4}$	$76.2^{-0.8}$	$91.2^{-1.9}$	$49.7^{+1.7}$	$16.4^{+0.0}$	$55.8^{+0.0}$
	post-exec. w/ *detail*	1	$81.1^{-7.9}$	$72.8^{-4.2}$	$89.9^{-3.2}$	$49.1^{+1.1}$	$13.8^{-2.6}$	$54.4^{-1.2}$
		2	$79.3^{-9.7}$	$75.4^{-1.6}$	$85.5^{-7.6}$	$43.3^{-4.7}$	$8.6^{-7.8}$	$49.3^{-6.5}$
LLaMA-3-70B-Ins	w/o *self-debug*	0	73.8	71.2	72.3	10.3	2.6	30.2
	post-exec. w/ *label*	1	$65.2^{-8.6}$	$68.3^{-2.9}$	$66.0^{-6.3}$	$9.1^{-1.2}$	$3.4^{+0.8}$	$27.8^{-2.4}$
		2	$69.5^{-4.3}$	$68.3^{-2.9}$	$64.8^{-7.5}$	$10.9^{+0.6}$	$2.6^{+0.0}$	$27.8^{-2.4}$
	post-exec. w/ *detail*	1	$66.5^{-7.3}$	$64.8^{-6.4}$	$56.6^{-15.7}$	$10.9^{+0.6}$	$4.3^{+1.7}$	$25.3^{-4.9}$
		2	$67.1^{-6.7}$	$63.8^{-7.4}$	$63.5^{-8.8}$	$12.0^{+1.7}$	$2.6^{+0.0}$	$27.8^{-2.4}$
Qwen2.5-Coder-7B-Ins	w/o *self-debug*	0	81.7	70.6	74.8	23.4	8.6	35.8
	post-exec. w/ *label*	1	$78.0^{-3.7}$	$69.8^{-0.8}$	$69.8^{-5.0}$	$24.0^{+0.6}$	$8.6^{+0.0}$	$34.2^{-1.6}$
		2	$79.3^{-2.4}$	$69.8^{-0.8}$	$71.7^{-3.1}$	$23.4^{+0.0}$	$8.6^{+0.0}$	$34.7^{-1.1}$
	post-exec. w/ *detail*	1	$76.2^{-5.5}$	$68.0^{-2.6}$	$69.2^{-5.6}$	$20.0^{-3.4}$	$8.6^{+0.0}$	$32.4^{-3.4}$
		2	$75.6^{-6.1}$	$69.0^{-1.6}$	$66.7^{-8.1}$	$21.1^{-2.3}$	$8.6^{+0.0}$	$32.0^{-3.8}$

manEval and MBPP benchmarks. Table 1 presents the pass rates achieved through post-execution self-debugging, showcasing notable declines in performance on both benchmarks for LLaMA-3-70B-Instruct and Qwen2.5-Coder-7B-Instruct. For GPT-4o and Claude-3.5-Sonnet, a consistent decrease is observed on HumanEval. While the performance on MBPP may show initial improvement, the use of more detailed feedback ultimately leads to worse outcomes compared to the initial generation. The results highlight the challenges and limitations of the post-execution method with self-generated tests, even for state-of-the-art models.

Failure Case Study In post-execution settings, erroneous test outputs introduce ambiguity into the self-debugging process. We illustrate this challenge with an example from HumanEval using GPT-4o in Figure 7 in Appendix E. *When a test fails, the model is expected to determine whether the failure stems from defects in the program or errors in the test itself.* This uncertainty complicates the self-debugging process and necessitates a further investigation into the effects of testing on self-generated tests. While post-execution information with self-generated tests is leveraged, self-debugging remains a bottleneck, limiting improvements beyond initial generation.

3.3 RQ2: Bias from Self-Testing Leads to Inconsistency Across Levels of Tasks

To investigate the performance of self-debugging on more challenging tasks, we further conduct post-execution self-debugging experiments using problems from LiveCodeBench. The problems are classified into three distinct difficulty levels: easy, medium, and hard. We report the pass rate achieved at each level of difficulty, as well as the overall performance across the benchmark.

Label Feedback Provides Some Relief on More Challenging Tasks. Table 1 also summarizes the results of post-execution self-debugging on LiveCodeBench. We find that self-debugging using label feedback generally leads to improvements on medium and hard-level problems. This is notably in contrast to their performance on basic and easy problems. All models except GPT-4o experience a decrease in overall performance, primarily due to significant declines in easy problems. Despite incorporating more post-execution results, the overall performance with detailed feedback remains inferior to that achieved with label feedback.

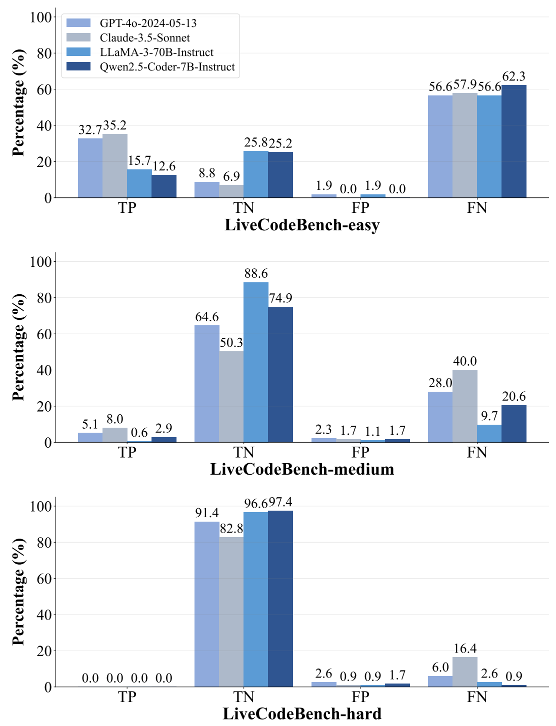

Figure 3: Test bias in labels when evaluating the programs with self-generated tests on different difficulty levels of problems in LiveCodeBench.

Self-Testing Bias Analysis To investigate the reasons behind the inconsistent results across different levels of problems, we delve into the impact on testing programs with self-generated tests. We acknowledge that even advanced LLMs are likely to generate inaccurate tests (Gu et al., 2024). As a result, a program that is actually correct might fail some of the generated tests, resulting in a false negative label (FN). Conversely, a flawed program might pass all the test cases, leading to a false positive label (FP), which could prevent necessary updates and prematurely present a buggy program as correct. *The misalignment between self-testing labels and true labels highlights the bias introduced by self-generated tests in program evaluation.*

We present an analysis of test bias in labels at the first iteration of self-debugging, as illustrated in Figure 3. Given the implementation of self-debugging, only programs identified with negative labels during the iteration would perform further repair. Therefore, our focus is primarily on the distribution of different negative labels. We observed that testing on self-generated tests is more likely to result in false negative labels than true negative ones on easy problems. However, a different pattern emerges on medium and hard ones, where false negatives outnumber true negatives[†]. *This discrepancy is primarily attributed to lower performance on more complex and challenging tasks, where negative labels from self-testing are more likely to align with the actual labels of the generated programs.* Relying solely on label feedback during self-debugging inadvertently reduces the bias from self-testing, thereby increasing the prevalence of true negative labels. However, when incorrect details are included in feedback, the performance declines compared to using only label feedback.

Moreover, when self-testing results in a false negative, it is crucial for the model to accurately identify the errors within the feedback and preserve the original answers. The efficacy of post-execution self-debugging, depends not only on the model's ability to identify the defects in its own programs when presented with true negative labels but also on its ability to recognize the faulty execution feedback in the case of false negatives.

3.4 RQ3: In-Execution Reasoning Helps Self-Debugging

Despite the conditional utility of post-execution self-debugging in challenging tasks, its dependency on potentially unreliable test labels remains a fundamental fragility. We therefore explore in this subsection whether the in-execution method—by debugging through intermediate states—can achieve consistent gains across different levels of tasks.

Drawing inspiration from the implementation presented in LDB (Zhong et al., 2024), we divide an execution trace into several units based on basic blocks in the program's control flow graph and collect the variables before and after these units during program execution to facilitate in-execution self-debugging. Please refer to Appendix D for a detailed illustration. Different from existing works that utilize runtime information for debugging (Zhong et al., 2024; Ni et al., 2024; Ding et al., 2024), we decouple post-execution signals from these methods, which means the labels and details from post-execution results are not accessible for the models. Therefore, the models must determine program correctness merely based on the test input and corresponding intermediate states.

In-Execution Self-Debugging Shows Significant Potential for Improvement. The results of in-

[†]We also present an analysis of test bias on HumanEval and MBPP in Figure 6.

Table 2: Results of **in-execution self-debugging** (in-exec.) with *self-generated tests*. The values in red are declines while those in green are increases compared to the initial generation (w/o self-debug).

Model	Method	#Iter	HumanEval+	MBPP+	LiveCodeBench			
					easy	medium	hard	overall
GPT-4o-2024-05-13	w/o *self-debug*	0	87.8	76.5	89.3	33.1	6.0	46.0
	in-exec. w/ *trace*	1	$89.0^{+1.2}$	$77.8^{+1.3}$	$91.2^{+1.9}$	$34.9^{+1.8}$	$6.0^{+0.0}$	$47.3^{+1.3}$
		2	$88.4^{+0.6}$	$79.1^{+2.6}$	$91.8^{+2.5}$	$34.9^{+1.8}$	$6.0^{+0.0}$	$47.6^{+1.6}$
Claude-3.5-Sonnet	w/o *self-debug*	0	89.0	77.0	93.1	48.0	16.4	55.8
	in-exec. w/ *trace*	1	$89.6^{+0.6}$	$77.2^{+0.2}$	$95.0^{+1.9}$	$49.1^{+1.1}$	$17.2^{+0.8}$	$57.1^{+1.3}$
		2	$87.2^{-1.8}$	$76.2^{-0.8}$	$93.7^{+0.6}$	$48.6^{+0.6}$	$17.2^{+0.8}$	$56.4^{+0.6}$
LLaMA-3-70B-Ins	w/o *self-debug*	0	73.8	71.2	72.3	10.3	2.6	30.2
	in-exec. w/ *trace*	1	$70.1^{-3.7}$	$69.6^{-1.6}$	$73.0^{+0.7}$	$11.4^{+1.1}$	$3.4^{+0.8}$	$31.1^{+0.9}$
		2	$74.4^{+0.6}$	$69.6^{-1.6}$	$71.1^{-1.2}$	$12.0^{+1.7}$	$3.4^{+0.8}$	$30.7^{+0.5}$
Qwen2.5-Coder-7B-Ins	w/o *self-debug*	0	81.7	70.6	74.8	23.4	8.6	35.8
	in-exec. w/ *trace*	1	$82.3^{+0.6}$	$71.4^{+0.8}$	$75.5^{+0.7}$	$24.0^{+0.6}$	$8.6^{+0.0}$	$36.2^{+0.4}$
		2	$82.3^{+0.6}$	$72.0^{+1.4}$	$76.1^{+1.3}$	$24.0^{+0.6}$	$8.6^{+0.0}$	$36.4^{+0.6}$

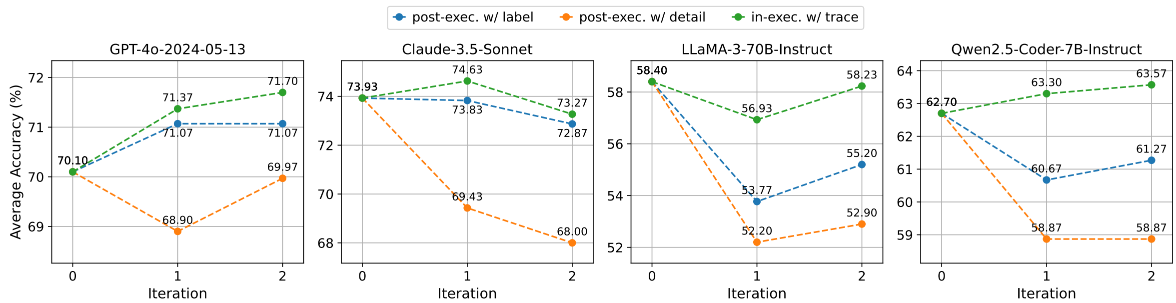

Figure 4: Average accuracy of both post-execution and in-execution self-debugging across three benchmarks over two iterations. In-execution self-debugging, which only leverages intermediate runtime information, consistently outperforms the post-execution approach.

execution self-debugging are detailed in Table 2. For basic problems from HumanEval and MBPP, we observe that self-debugging gains notable improvement for GPT-4o and Qwen2.5-Coder-7B-Instruct when utilizing in-execution information. Claude-3.5-Sonnet also shows performance improvements at the first iteration on both benchmarks. For LLaMA-3-70B-Instruct, the performance on HumanEval surpasses the baseline at the second iteration. However, there is a slight degradation in performance for certain tasks and iterations compared to the initial generation. For competitive problems from LiveCodeBench, the results demonstrate the efficacy of the in-execution self-debugging across all models. The results indicate that in-execution self-debugging is a potentially effective approach for improving performance on diverse programming problems.

How In-Execution Works Compared to Post-Execution Method We present the average accuracy of both post-execution and in-execution methods over two iterations in Figure 4. Since the debugging process is triggered using self-generated tests rather than ground truth oracle tests, performance does not necessarily improve with successive iterations. This differs from other existing debugging methods (Chen et al., 2024b; Shinn et al., 2023; Zhong et al., 2024), where the set of positive programs in the current iteration is typically a superset of those in the previous iteration. The results suggest that *in-execution self-debugging, which only leverages intermediate runtime information, consistently outperforms the post-execution approach. Moreover, in most cases, it shows improvements compared to the initial generation, indicating significant potential for future development.* To further illustrate how the in-execution method works, we provide a comparison with an example from HumanEval using GPT-4o in Appendix E.

Strengths and Weaknesses of In-Execution Self-Debugging To conclude, post-execution self-debugging utilizes final execution results to reflect upon and debug programs. However, the unreliability of the self-generated tests could bias the model away from the correct answer. *Although this can provide some relief on challenging tasks, it is not a long-term solution, especially when those hard tasks can also be solved well over time.* On the contrary, in-execution self-debugging allows the models to perform fine-grained feedback solely on the intermediate states during the execution process, mitigating the test bias introduced by potentially flawed tests. It shows the potential to better align the programs with the requirements in real-world scenarios. Nonetheless, self-debugging with in-execution information depends heavily on the LLMs' code reasoning capabilities and lacks formal guarantees of success, as the pass rate drops for LLaMA-3-70B-Instruct on MBPP. We expect that improvements in LLM capabilities will enhance the efficacy of this paradigm.

4 Directions for Future Work

Based on our findings, we deliver a detailed discussion on directions for future work in this Section.

High-Quality Test Generation. In this work, we demonstrate that post-execution self-debugging with self-generated tests struggles on basic problems due to biased evaluations, despite the significant potential shown by LLMs in automated test generation. This highlights the necessity for the research community to focus on the quality of LLM-generated tests before utilizing execution feedback derived from them. Developing techniques that enhance high-quality test synthesis is crucial to mitigate bias for post-execution self-debugging. It could be beneficial to implement an iterative refinement process wherein execution information is leveraged to improve the tests. This could involve using techniques like test-driven development where tests are continuously updated based on code changes and debugging outcomes.

More Deliberate Runtime Information. As demonstrated in Section 3.4, leveraging enriched runtime information from execution is a promising avenue for self-debugging. In particular, in-execution self-debugging has shown superior performance compared to post-execution in certain tasks, suggesting that more nuanced and reliable feedback leads to better performance. Designing more sophisticated methods for collecting and analyzing runtime information is a promising direction for further enhancing self-debugging capabilities. For instance, improving the intelligibility of execution trace representations for LLMs may prove beneficial (Ni et al., 2024). Additionally, beyond variables, other types of runtime information, such as code coverage, execution paths, and memory usage, could be effectively utilized to provide a more comprehensive view of program behavior (Chen et al., 2024a).

Training for More Advanced Capabilities of LLMs. Effective self-debugging with self-generated tests hinges on several core capabilities of LLMs. In terms of refinement, the model should be capable of accurately recognizing and localizing faults within the program. This requires not only an understanding of the code's syntax but also its semantic logic and runtime behavior. Additionally, more advanced reasoning capabilities are needed to thoroughly analyze execution feedback. The model should comprehend the relationship between the code logic and the feedback, thereby deducing the runtime structure of program statements and variables. Training LLMs with diverse debugging scenarios (Ni et al., 2024) and incorporating multi-step reasoning frameworks (Jiang et al., 2024a) could further enhance their ability to diagnose and fix errors autonomously.

Self-Evolution of Code LLMs. Self-debugging opens up possibilities for developing more advanced LLMs without reliance on human supervision or guidance from stronger models (Burns et al., 2024). Traditionally, human-generated test cases serve as a strong supervisory signal for aligning code generation, but the collection of such tests is labor-intensive, leading to a sparsity of labeled data for effective code refinement. Self-generated tests, by contrast, offer a viable path for self-improvement (Tao et al., 2024). They alleviate the burden of manual test generation and pave the way toward truly autonomous self-correcting code generation systems (Chen et al., 2024b). By iteratively refining both the code and the tests, LLMs can evolve into more robust and reliable systems capable of handling increasingly complex programming tasks. This self-evolution paradigm could revolutionize the development of code generation models, making them more adaptable and scalable for real-world applications.

These directions highlight the potential for advancing self-debugging techniques, ultimately leading to more autonomous, accurate, and efficient code generation systems. By addressing these challenges, the research community can unlock new possibilities for LLMs in software development and beyond.

5 Related Work

LLM-Based Code Generation. Code generation is the automatic production of source code based on natural language descriptions. Researchers have proposed various approaches to enhance the quality of code generated by these models. Some works, like LLaMA (Touvron et al., 2023a,b; Dubey et al., 2024) and DeepSeek (DeepSeek-AI et al., 2024b,a) series, focus on optimizing model training, while others aim to improve code quality through post-processing techniques (Chen et al., 2023; Zhang et al., 2023b; Inala et al., 2022). Among these post-processing techniques, methods that involve self-debugging have gained considerable attention. Through feedback from execution results, self-debugging allows models to autonomously debug and refine previously generated code, enhancing the final output. Self-debugging has been integrated into various LLM-based code generation methods (Yang et al., 2024; Zhang et al., 2024; Dong et al., 2023; Huang et al., 2023). In this work, we revisit these techniques, evaluating self-debugging with self-generated tests on both basic and competitive tasks.

Self-Debugging with LLMs. As LLMs have evolved, the idea of using models to refine their own output has become more popular. Most of these methods rely on prompting LLMs with execution results to improve the code. These methods often rely on pre-existing or generated tests to execute the code, capturing execution information that is then used to refine the output code ((Olausson et al., 2024; Wang et al., 2024; Dong et al., 2023; Madaan et al., 2023; Zhang et al., 2023a; Jiang et al., 2024b)). Self-Refine (Madaan et al., 2023) conducts a broad evaluation of self-debugging in code models, highlighting that performance can be improved with higher-quality feedback or human intervention. Self-Debugging (Chen et al., 2024b) and Self-Edit (Zhang et al., 2023a) use the framework in which LLMs can iteratively debug their own generated code by utilizing execution results. LDB (Zhong et al., 2024) utilizes runtime execution information to help debug generated programs. To enhance the execution understanding capability of LLMs, SemCoder (Ding et al., 2024) trains models to simulate execution, enabling abstract reasoning and debugging. NExT (Ni et al., 2024) trains LLMs to generate NL rationales based on execution traces for accurate runtime reasoning. These methods leverage both post and in-execution information for self-debugging since there are predefined oracle tests, which is an idealized setup in real-world scenarios. In this work, we aim to investigate the effect of post-execution and in-execution information respectively, particularly with self-generated tests. We propose a unified framework and provide a detailed analysis of these methods.

6 Conclusion

In this work, we explore self-debugging with self-generated tests to enhance the code generation capabilities of large language models (LLMs). We introduce and evaluate two paradigms: post-execution and in-execution self-debugging. While post-execution methods struggle with simpler tasks due to bias from self-generated tests, in-execution self-debugging leverages intermediate runtime states to mitigate this bias and consistently outperforms post-execution approaches. Our findings highlight the importance of reducing reliance on flawed self-generated tests and incorporating richer runtime feedback for more accurate and robust program refinement. This work underscores the potential of in-execution self-debugging to advance LLMs in code generation, paving the way for more autonomous and reliable code generation systems.

Limitations

We evaluate the efficacy of the proposed self-debugging paradigms on a wide range of LLMs and across three benchmarks covering both basic and competitive programming problems. While our experiments could provide valuable insights into self-debugging for real-world applications, it remains unclear whether these findings would directly translate to more complex software engineering scenarios. For instance, consider repository-level debugging: a model analyzing intermediate API call sequences or variable propagation across modules could mirror the in-execution paradigm. However, questions such as how to efficiently capture and structure runtime data across interconnected modules, or how to handle dependencies and interactions between components, remain open. These complexities highlight the need for further research to adapt self-debugging techniques to large-scale, real-world software systems.

We leave these challenges for future work, as they represent an important next step in advancing the applicability of self-debugging paradigms beyond self-contained programming tasks. By addressing these issues, we can move closer to developing autonomous and self-evolving systems capable of handling the intricacies of modern software engineering.

Acknowledgement

This research was partly supported by the National Natural Science Foundation of China (NSFC, Grant No. W2411051).

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Islem Bouzenia, Yangruibo Ding, Kexin Pei, Baishakhi Ray, and Michael Pradel. 2023. Tracefixer: Execution trace-driven program repair. *CoRR*, abs/2304.12743.

Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeffrey Wu. 2024. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024a. Evaluating large language models with runtime behavior of program execution. *CoRR*, abs/2403.16437.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024b. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024a. Deepseek-v3 technical report. *CoRR*, abs/2412.19437.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. 2024b. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *CoRR*, abs/2406.11931.

Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail E. Kaiser, Junfeng Yang, and Baishakhi Ray. 2024.

Semcoder: Training code language models with comprehensive semantics reasoning. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration code generation via chatgpt. *CoRR*, abs/2304.07590.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Robert Gold. 2010. Control flow graphs and code coverage. *Int. J. Appl. Math. Comput. Sci.*, 20(4):739–749.

Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

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

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codas, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *CoRR*, abs/2403.07974.

Jin Jiang, Yuchen Yan, Yang Liu, Yonggang Jin, Shuai Peng, Mengdi Zhang, Xunliang Cai, Yixin Cao, Liangcai Gao, and Zhi Tang. 2024a. Logicpro: Improving complex logical reasoning via program-guided learning. *arXiv preprint arXiv:2409.12929*.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024b. Training llms to better self-debug and explain code. *CoRR*, abs/2405.18649.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Self-evolve: A code evolution framework via large language models. *CoRR*, abs/2306.02907.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng

Yin. 2024. Next: Teaching large language models to reason about code execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Maxwell I. Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *CoRR*, abs/2112.00114.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *CoRR*, abs/2401.08500.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. 2024. A survey on self-evolution of large language models. *CoRR*, abs/2404.14387.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. INTERVENOR: prompting the coding ability of large language models with the interactive chain of repair. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 2081–2107.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *CoRR*, abs/2405.15793.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 13643–13658. Association for Computational Linguistics.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, pages 41832–41846.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 851–870. Association for Computational Linguistics.

A Benchmark

We utilize HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) and LiveCodeBench (Jain et al., 2024) covering basic and competitive programming problems to comprehensively evaluate the efficacy of self-debugging:

- **HumanEval and MBPP** HumanEval (Chen et al., 2021) consists of 164 programming problems written by humans. Each problem provides a Python function signature and a docstring as its specification. MBPP (Austin et al., 2021) includes 974 programming problems written by contributors through crowdsourcing. Each of these problems features a problem statement, a function signature, and three example tests. To enhance the reliability and accuracy of evaluations, EvalPlus (Liu et al., 2023) extends HumanEval into a more comprehensive version known as HumanEval+ with 80 times more tests than the original HumanEval. Similarly, MBPP+ is an augmentation of the original MBPP, offering 35 times more tests. In our experiments, we use the latest version of MBPP+, which consists of 378 programming problems.
- **LiveCodeBench** LiveCodeBench (Jain et al., 2024) is a contamination-free benchmark that continuously collects new problems from prominent competitive programming platforms. As of now, LiveCodeBench features a collection of over 600 high-quality programming problems. These problems encompass a wide range of difficulty levels and topics, providing a comprehensive evaluation for the coding capabilities of LLMs. In our experiments, we select 450 problems that were published between September 2023 and September 2024.

B Validation of Generated Tests

In Section 3.2, we leverage program contracts and canonical solutions to evaluate the reliability of the generated tests for problems in HumanEval and MBPP[‡]. Program contracts ensure that the generated test inputs adhere to the preconditions specified for each problem, while canonical solutions serve as the ground truth to verify the correctness of the test outputs. We present examples for both benchmarks in Figure 5.

[‡]Neither program contracts nor canonical solutions are provided by LiveCodeBench.

Table 3: Results for increasing number of self-generated tests using GPT-4o-2024-05-13 on HumanEval and MBPP.

(a) HumanEval

#Num of Tests	Input	Output	Suite
10	97.63%	89.77%	59.15%
15	97.89%	88.86%	52.44% (-6.71%)
20	98.11%	86.01%	48.17% (-10.98%)

(b) MBPP

#Num of Tests	Input	Output	Suite
10	94.81%	85.60%	58.73%
15	94.96%	85.27%	53.70% (-5.03%)
20	95.10%	82.94%	50.53% (-8.20%)

Table 4: Results of post-execution self-debugging with detailed feedback and in-execution self-debugging on HumanEval and MBPP when using different sizes of the self-generated test suite. The values highlighted in red or green are changes relative to the initial generation.

Method	#N	HumanEval+	MBPP+
w/o *self-debug*	0	87.8	76.5
post-exec. w/ *detail*	10	$84.1^{-3.7}$	$76.2^{-0.3}$
	15	$84.1^{-3.7}$	$75.9^{-0.6}$
	20	$83.5^{-4.3}$	$75.9^{-0.6}$
in-exec. w/ *trace*	10	$89.0^{+1.2}$	$77.8^{+1.3}$
	15	$88.4^{+0.6}$	$78.0^{+1.5}$
	20	$88.4^{+0.6}$	$77.2^{+0.7}$

To validate test inputs, we place contracts at the beginning of the function and pass the test input to it. If no assertion error occurs, the test input is considered valid. Some functions have additional requirements for inputs beyond just type, so the contracts check for both type mismatches and any other specified constraints. For test output validation, we use canonical solutions to collect the execution output for a given valid input and compare it with the generated test output to confirm correctness.

C Varying Number of Generated Tests

To investigate the effect of the number of self-generated tests, we employ GPT-4o-2024-05-13 to generate varying number of tests $N = [10, 15, 20]$ for each programming problem in HumanEval and MBPP. Following Section 3.2, we calculate the accuracy of these generated tests and the results are summarized in Table 3. As the number of self-generated tests increases, the presence of more challenging edge cases also rises, leading to a reduction in the overall accuracy of the test suites. Specifi-

```
# HumanEval/21

def rescale_to_unit(numbers: List[float]) -> List[float]:
    """ Given list of numbers (of at least two elements), apply a linear transform to
that list, such that the smallest number will become 0 and the largest will become 1
    >>> rescale_to_unit([1.0, 2.0, 3.0, 4.0, 5.0])
    [0.0, 0.25, 0.5, 0.75, 1.0]
    """

    assert all(type(x) in [int, float] for x in numbers), "invalid inputs" #
$_CONTRACT_$
    assert len(numbers) >= 2, "invalid inputs" # $_CONTRACT_$
    assert max(numbers) > min(numbers), "invalid inputs" # $_CONTRACT_$

    ma, mi = max(numbers), min(numbers)
    k = 1 / (ma - mi)
    return list(map(lambda x: (x - mi) * k, numbers))
```

```
# MBPP/439

"""
Write a function to join a list of multiple integers into a single integer.
assert multiple_to_single([11, 33, 50])==113350
"""

def multiple_to_single(L):

    assert isinstance(L, list), "invalid inputs" # $_CONTRACT_$
    assert len(L) > 0, "invalid inputs" # $_CONTRACT_$
    assert all(isinstance(item, int) for item in L), "invalid inputs" # $_CONTRACT_$
    assert all(item > 0 for item in L[1:]), "invalid inputs" # $_CONTRACT_$

    return int(''.join(map(str,L)))
```

Figure 5: Examples of program contracts in HumanEval and MBPP. Program contracts consist of assertions that specify conditions necessary for a valid input.

cally, when the model generates up to 20 tests per problem, the accuracy of the test suite decreases from 59.15% to 48.17% for HumanEval and from 58.73% to 50.53% for MBPP.

We further evaluate the performance of both post-execution self-debugging with detailed feedback and in-execution self-debugging. The results in Table 4 indicate that with an increased number of self-generated tests, the performance of post-execution self-debugging experiences a slight decline on both HumanEval and MBPP. This decline is attributed to the lower accuracy of the test suites, which increases the rate of false negatives and thereby hinders the efficacy of post-execution self-debugging. In contrast, in-execution self-debugging, which leverages intermediate runtime traces, demonstrates consistent improvement over the initial generation.

These findings highlight the robustness of in-execution self-debugging in handling the challenges posed by less accurate test suites, as it relies on precise runtime information rather than potentially flawed self-generated tests. They also underscore the importance of balancing the quantity and quality of self-generated tests to ensure their effectiveness in evaluating and debugging programs. By leveraging runtime insights, in-execution self-debugging provides a more reliable approach to program repair and enhancement, particularly in scenarios where test accuracy is compromised.

D Collection of In-Execution Information

In this work, we partition a program into basic blocks based on the nodes within its control flow graph (CFG) (Gold, 2010; Zhong et al., 2024). A control flow graph is a representation of the different blocks of code in a program and various execution paths that an executor can take through the code. Each block represents a maximal sequence of statements and expressions without any jumps

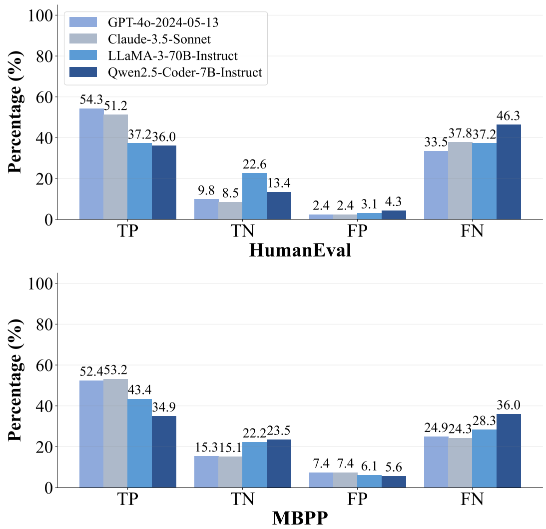

Figure 6: Test bias in labels when evaluating the programs with self-generated tests for HumanEval and MBPP.

or jump targets. Directed edges in the graph reflect the possible execution order among the blocks.

We simply use variables as the key indicators of intermediate states during the execution. To facilitate the visibility of necessary variables, we strategically insert specific lines of code before and after these basic blocks. Given a test input, the program subsequently outputs the variables in execution order. We then organize the content of each reachable basic block and its corresponding variables into execution trace blocks in sequence, which are integrated into the debugging prompt as shown in Figure 10. We provide an example illustrating the representation of in-execution information in Figure 7. For programs containing loops, we are able to capture the change of the intermediate states within each loop iteration. Additionally, in cases where the number of trace blocks exceeds a predefined threshold N, we adopt a strategy to retain the first $N/2$ blocks and the last $N/2$ blocks of the trace. It ensures that both the initial and final segments of the trace, which may contain essential information, are preserved for analysis while maintaining a manageable context length.

E Case Study

In our experiments, we observe that in-execution self-debugging, which leverages runtime information, consistently outperforms post-execution one across various levels of programming tasks. To better understand the unique strengths and weaknesses of these two paradigms, we provide an example involving GPT-4o in Figure 7.

This example illustrates different outcomes of post-execution self-debugging with detailed test feedback and in-execution self-debugging with execution traces. The initial completion for the `can_arrange` function is correct. However, it is evaluated against an erroneous self-generated test that, according to the specification, should return `4` instead of `1`. This discrepancy causes the model to misinterpret the condition in the problem, thereby leading to an incorrect revision of the program. Feedback from post-execution self-debugging, when based on flawed self-generated tests, biases the model away from the problem's specification, resulting in a faulty program.

In contrast, in-execution self-debugging leverages test inputs and their corresponding runtime information to assess program correctness. As depicted in Figure 7, this approach enables the model to perform a fine-grained analysis on the execution trace block by block without being influenced by the potential bias introduced by self-generated tests. By examining the intermediate states in execution order, the model confirms that the trace aligns with the expected behavior of the function, ultimately preserving the correctness of the original program.

This example highlights the critical advantage of in-execution self-debugging: its ability to rely on precise runtime information rather than potentially flawed self-generated tests. By avoiding the test bias, in-execution self-debugging provides a more reliable and effective approach to program repair and enhancement. These findings underscore the importance of leveraging runtime insights to improve the robustness and accuracy of self-debugging methods.

F Prompts

F.1 Prompts for Initial Code Generation

The prompts for initial program generation are provided in Figure 8.

F.2 Prompts for Test Generation

The prompts for test case generation are provided in Figure 9.

F.3 Prompts for Debugging

The prompts for post-execution and in-execution self-debugging are provided in Figure 10 and Figure 11.

Initial Generation (w/o self-debug)

```
========================================= SYSTEM =========================================
You are an expert programming assistant.

========================================== USER ==========================================
Here is the given code to do completion:
```python

def can_arrange(arr):
    """Create a function which returns the largest index of an element which
    is not greater than or equal to the element immediately preceding it. If
    no such element exists then return -1. The given array will not contain
    duplicate values.

    Examples:
    can_arrange([1,2,4,3,5]) = 3
    can_arrange([1,2,3]) = -1
    """

```
Please complete the `can_arrange` function and make sure that it matches the
specification and passes all tests. You are not allowed to modify the given function
signature. Think step by step and provide all completed codes in one code block.

====================================== ASSISTANT ======================================
Sure, let's complete the `can_arrange` function step by step.

1. We need to iterate through the array from the second element to the last element.
2. For each element, we need to check if it is not greater than or equal to the element
immediately preceding it.
3. If such an element is found, we return its index.
4. If no such element is found by the end of the loop, we return -1.

Here is the completed code:

```python
def can_arrange(arr):
    """Create a function which returns the largest index of an element which
    is not greater than or equal to the element immediately preceding it. If
    no such element exists then return -1. The given array will not contain
    duplicate values.

    Examples:
    can_arrange([1,2,4,3,5]) = 3
    can_arrange([1,2,3]) = -1
    """
    for i in range(len(arr) - 1, 0, -1):
        if arr[i] < arr[i - 1]:
            return i
    return -1
```

This function iterates through the array from the end to the beginning, checking if the
current element is less than the previous element. If it finds such an element, it
returns the index of that element. If no such element is found, it returns -1.
```

In-Execution Self-Debugging w/ trace

```
========================================== USER ==========================================
Given an input for the function `can_arrange([5, 4, 3, 2, 1])`, here is the code
execution trace block by block with the intermediate variable values:
[BLOCK-0]
    # arr = [5, 4, 3, 2, 1]
    for i in range(len(arr) - 1, 0, -1):
    # arr = [5, 4, 3, 2, 1]    i = 4
[BLOCK-1]
        # arr = [5, 4, 3, 2, 1]    i = 4
        if arr[i] < arr[i - 1]:
        # arr = [5, 4, 3, 2, 1]    i = 4
[BLOCK-2]
            # arr = [5, 4, 3, 2, 1]    i = 4
            return i
            # arr = [5, 4, 3, 2, 1]    i = 4    _ret = 4


Please explain the execution FOR EACH BLOCK and answer whether this block is correct or
not based on the specifications and given samples in the problem.

======================================= ASSISTANT =======================================
Let's go through the execution trace block by block and explain the intermediate
variable values and the correctness of each block.

### [BLOCK-0]
'''python
# arr = [5, 4, 3, 2, 1]
for i in range(len(arr) - 1, 0, -1):
# arr = [5, 4, 3, 2, 1]   i = 4
'''
**Explanation:**
- The array `arr` is `[5, 4, 3, 2, 1]`.
- The `for` loop is set to iterate from the last index (`len(arr) - 1`, which is 4) to
the first index (1), decrementing by 1 each time.
- In the first iteration, `i` is initialized to 4.

**Correctness:**
- This block is correct. The loop is set up properly to iterate from the end of the
array to the beginning.

### [BLOCK-1]
'''python
# arr = [5, 4, 3, 2, 1]   i = 4
if arr[i] < arr[i - 1]:
# arr = [5, 4, 3, 2, 1]   i = 4
'''
**Explanation:**
- The condition `if arr[i] < arr[i - 1]` is checked.
- For `i = 4`, `arr[4]` is 1 and `arr[3]` is 2.
- The condition `1 < 2` is true.

**Correctness:**
- This block is correct. The condition correctly checks if the current element is less
than the previous element.
```

```
### [BLOCK-2]
```python
# arr = [5, 4, 3, 2, 1]   i = 4
return i
# arr = [5, 4, 3, 2, 1]   i = 4    _ret = 4
```
Explanation:
- Since the condition in [BLOCK-1] is true, the function returns the current index `i`,
which is 4.

Correctness:
- This block is correct. The function correctly returns the index of the element that is
not greater than or equal to the element immediately preceding it.

Conclusion
The execution trace is correct and matches the expected behavior of the function based
on the given specifications and samples. The function correctly identifies that the
element at index 4 (value 1) is less than the element at index 3 (value 2) and returns
the index 4.
```

Post-Execution Self-Debugging w/ detail

````
======================================== USER ========================================
But it failed in the following test case:
assert can_arrange([5, 4, 3, 2, 1]) == 1, "Test Case 3: Descending order array”
# Real Execution Output: 4

Please fix the bug in the `can_arrange` function and make sure that the fixed code
matches the specification and passes all tests. Provide the fixed code in one code block.
If there is no bug in your previous code, repeat the code in a code block again. You are
not allowed to modify the given function signature. Think step by step.

===================================== ASSISTANT =====================================
I see the issue. The function should return the largest index of an element which is not
greater than or equal to the element immediately preceding it. In the case of a
descending order array, the function should return the first index where this condition
is met, which is `1` in the given test case.

To fix this, we need to iterate from the beginning of the array to the end, and return
the first index where the condition is met. Here is the corrected code:

```python
def can_arrange(arr):
    """Create a function which returns the largest index of an element which
    is not greater than or equal to the element immediately preceding it. If
    no such element exists then return -1. The given array will not contain
    duplicate values.

    Examples:
    can_arrange([1,2,4,3,5]) = 3
    can_arrange([1,2,3]) = -1
    """
    for i in range(1, len(arr)):
        if arr[i] < arr[i - 1]:
            return i
    return -1
```

This function now correctly iterates from the beginning of the array to the end, and
returns the first index where the current element is less than the previous element.
This should pass all the given test cases.
````

Figure 7: An example with GPT-4o performing both post and in-execution self-debugging on a problem from HumanEval (HumanEval/135) respectively. Post-execution self-debugging wrongly corrects the program while in-execution self-debugging manages to keep the original answer.

Code Generation Prompt for HumanEval

````
Here is the given code to do completion:
```python
{prompt}
```
Please complete the `{entry_point}` function and make sure that it matches the
specification and passes all tests. You are not allowed to modify the given function
signature. Think step by step and provide all completed codes in one code block.
````

Code Generation Prompt for MBPP

````
Here is the given problem to solve:
```python
{prompt}
```
Please implement the `{entry_point}` function and make sure that it matches the
specification and passes all tests. You are not allowed to modify the given function
name and arguments in the test examples. Think step by step and provide all completed
codes in one code block.
````

Code Generation Prompt for Functional-Input Question in LiveCodeBench

````
Here is the given programming problem to solve:
{content}

Please generate a correct python program that matches the specification and passes all
tests. Think step by step. You will use the following starter code to write the solution
to the problem and enclose your code within delimiters.
```python
{starter_code}
```
````

Code Generation Prompt for Stdin-Input Question in LiveCodeBench

````
Here is the given programming problem to solve:
{content}

Please generate a correct python program that matches the specification and passes all
tests. Read the inputs from stdin solve the problem and write the answer to stdout (do
not directly test on the sample inputs). Think step by step and enclose your code within
delimiters as follows:
```python
# YOUR CODE HERE
```
````

Figure 8: Prompts for initial code generation (w/o *self-debug*).

Test Generation Prompt for HumanEval

````
Here is the given code to do completion:
```python
{prompt}
```
Please provide ten comprehensive and valid test cases to verify whether the
`{entry_point}` function correctly solves the problem. You are not allowed to implement
the function. Think step by step and provide all test cases in one code block.

The format of test cases should be:
```python
assert {entry_point}(input) == expected_output, "Test Case Description"
```
````

Test Generation Prompt for MBPP

````
Here is the given problem to solve:
```python
{prompt}
```
Please provide ten comprehensive and valid test cases to verify whether the
`{entry_point}` function correctly solves the problem. You are not allowed to implement
the function. Think step by step and provide all test cases in one code block.

The format of test cases should be:
```python
assert {entry_point}(input) == expected_output, "Test Case Description"
```
````

Test Generation Prompt for LiveCodeBench

```
Here is the given programming problem to solve:
{content}

Please provide ten comprehensive test samples based on the specification and follow the
format of the given sample.

Your response should be organized like below and no extra information is allowed
(including explanation):
[Input]
<your input here>
[Output]
<your output here>
[Input]
...
```

Figure 9: Prompts for test generation.

Post-Execution Debugging Prompt for HumanEval

```
{error}

Please fix the bug in the `{entry_point}` function and make sure that the fixed code
matches the specification and passes all tests. Provide the fixed code in one code block.
If there is no bug in your previous code, repeat the code in a code block again. You are
not allowed to modify the given function signature. Think step by step.
```

Post-Execution Debugging Prompt for MBPP

```
{error}

Please fix the bug in the `{entry_point}` function and make sure that the fixed code
matches the specification and passes all tests. Provide the fixed code in one code block.
If there is no bug in your previous code, repeat the code in a code block again. You are
not allowed to modify the given function name and arguments in the test examples. Think
step by step.
```

Post-Exec. Debugging Prompt for Functional-Input Questions in LiveCodeBench

````
{error}

Please fix the bug in the code and make sure that the fixed code matches the
specification and passes all tests.
You will use the following starter code to write the solution to the problem and enclose
your code within delimiters.
```python
{starter_code}
```
````

Post-Exec. Debugging Prompt for Stdin-Input Questions in LiveCodeBench

````
{error}

Please fix the bug in the code and make sure that the fixed code matches the
specification and passes all tests.
Read the inputs from stdin solve the problem and write the answer to stdout (do not
directly test on the sample inputs). Enclose your code within delimiters as follows.
```python
# YOUR CODE HERE
```
````

Figure 10: Prompts for post-execution debugging.

In-Execution Debugging Prompt

```
Given an input for the function `{test}`, here is the code execution trace block by
block with the intermediate variable:
{trace}

Please explain the execution FOR EACH BLOCK and answer whether this block is correct or
not based on the specifications and given samples in the problem. If the program is
correct, please restate it in one python code block. If it is incorrect, please fix the
bug and provide the fixed code in a code block.
```

Figure 11: Prompt for in-execution self-debugging.