ConTested: Consistency-Aided Tested Code Generation with LLM

JINHAO DONG*, Peking University, China
JUN SUN, Singapore Management University, Singapore
WENJIE ZHANG, National University of Singapore, Singapore
JIN SONG DONG, National University of Singapore, Singapore
DAN HAO†, Peking University, China

Recent advancements in large language models (LLMs) have significantly improved code generation, which generates code snippets automatically based on natural language requirements. Despite achieving state-of-the-art performance, LLMs often struggle to generate accurate and reliable code, requiring developers to spend substantial effort debugging and evaluating the generated output. Researchers have proposed leveraging *Consistency* to select code that passes more tests (inter-consistency) and demonstrates consistent behavior across more counterparts (intra-consistency). However, since the tests themselves are also generated by LLMs, relying on majority voting based on incorrect tests leads to unreliable results. To address this, we propose a lightweight interaction framework that incorporates user feedback to effectively guide consistency. Our results demonstrate that, with minimal human effort, performance can be significantly improved. In each iteration, we introduce a rank-correct-fix co-evolution process between code and tests. This process iteratively enhances the quality of both, making the consistency voting between code and tests more reliable.
We evaluate ConTested through extensive experiments, demonstrating its effectiveness across multiple LLMs, including GPT-3.5 and GPT-4o. Our results show improvements of 32.9% over GPT-3.5 and 16.97% over GPT-4o. Additionally, ConTested achieves an 11.1% improvement over the SOTA post-processing technique, MPSC. This improvement is achieved with only a 4-round interaction with users, requiring minimal user effort. A user study further confirms the feasibility and cost-effectiveness of ConTested, highlighting its ability to enhance code generation without introducing substantial overhead.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Software development techniques**.

Additional Key Words and Phrases: Code Generation, Self-Consistency, Iterative Interaction

ACM Reference Format:
Jinhao Dong, Jun Sun, Wenjie Zhang, Jin Song Dong, and Dan Hao. 2025. ConTested: Consistency-Aided Tested Code Generation with LLM. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA027 (July 2025), 22 pages. https://doi.org/10.1145/3728902

*This work was done when Jinhao Dong was a visiting student in National University of Singapore (NUS) and Singapore Management University (SMU).
†Dan Hao is the corresponding author.

Authors' Contact Information: Jinhao Dong, Key Laboratory of High Confidence Software Technologies (Peking University), MOE, School of Computer Science, Peking University, Beijing, China, dongjinhao@stu.pku.edu.cn; Jun Sun, School of Computing and Information Systems, Singapore Management University, Singapore, Singapore, junsun@smu.edu.sg; Wenjie Zhang, School of Computing, National University of Singapore, Singapore, Singapore, wjzhang@nus.edu.sg; Jin Song Dong, School of Computing, National University of Singapore, Singapore, Singapore, dcsdjs@nus.edu.sg; Dan Hao, School of Electronic and Computer Engineering, Peking University, Shenzhen, China, haodan@pku.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA027
https://doi.org/10.1145/3728902

1 Introduction

Code generation techniques automatically generate code snippets that implement desired functionality based on natural language requirements. These techniques can reduce the effort required by developers to write code and improve development productivity, as extensively studied in the literature [6, 16, 18, 35]. Recent progress in large language models (LLMs) have significantly impacted the field of code generation. Researchers have introduced various LLMs [2, 11–13, 15, 19, 21, 24] (e.g., GPT-4 [2], DeepSeek-Coder [13], and CodeGen [24]) that achieve SOTA performance, due to their massive parameter scales and pre-training on extensive code-specific corpora.

Although LLMs have shown impressive performance, their outputs are not always reliable. Enhancing the reliability of LLM-generated results is crucial, because developers often invest significant effort to identify and correct errors in the generated code. To increase the reliability of LLM-generated results, *Consistency*[5, 14, 34, 39, 40, 44] has been proposed as an effective, lightweight technique. It generates multiple solutions for each input query and then uses a majority voting to determine the final answer, improving consistency and reliability. *Consistency* is founded on the assumption that the tasks generally have multiple reasoning paths leading to a correct answer [33]. By incorporating *diversity*, *Consistency* increases the likelihood that a consistent result is correct, as the chance of multiple approaches making the same error is low. *Consistency* helps mitigate the randomness, thereby improving output reliability and boosting performance [34, 38–40]. Other post-processing techniques that aim to enhance the reliability of LLMs are generally more resource-intensive compared to *Consistency*. Some approaches, for example, require training additional verifiers [9, 23] or re-rankers [36] to validate LLM outputs. These methods require separate model training, and the reliability of their results may still be questionable. In contrast, *Consistency* operates without the need for additional training or auxiliary models. Recently, *Consistency* has been applied to code generation tasks [5, 14], where the diversity of coding approaches—including various APIs, algorithms, and programming paradigms—offers multiple perspectives for applying consistency.

While *Consistency* improves the reliability of code generated by LLMs, existing approaches face a key limitation. *They overlook the preconditions required for the effective use of Consistency, and relying on Consistency alone is insufficient to guarantee the reliability of LLMs.* Specifically, the effectiveness of *Consistency* depends on the quality of the consistency indicators used to assess consistency. Since consistency indicators are also generated by LLMs, they can be erroneous, leading to unreliable *Consistency* and majority voting. Relying solely on consistency is insufficient to address this issue. Current techniques primarily focus on using tests or specifications as consistency indicators [5, 14], measuring inter-consistency and intra-consistency based on the number of passing tests and functionally equivalent counterparts. As shown in Fig. 3b (which will be discussed in detail in Section 2), codes in group 1 are selected because they pass more tests and have more functionally equivalent counterparts that pass the same set of tests. However, the tests generated by LLMs frequently contain errors, with an average error rate of 37.7% across three widely used code generation datasets, as observed in our experiments. This high error rate significantly reduces the effectiveness of *Consistency*. A buggy code that passes a higher number of erroneous tests may be mistakenly considered as reliable or correct. However, existing techniques neglect this issue when implementing *Consistency*. It is therefore essential to enhance the quality of consistency indicator prior to leveraging consistency. *Furthermore, existing techniques make limited use of Consistency.* Relying solely on majority voting provides only a superficial application of *Consistency*. There are additional ways to leverage *Consistency*, such as using inconsistency to identify potential problems. Once identified, these problems can be fixed to enhance reliability. Moreover, all existing

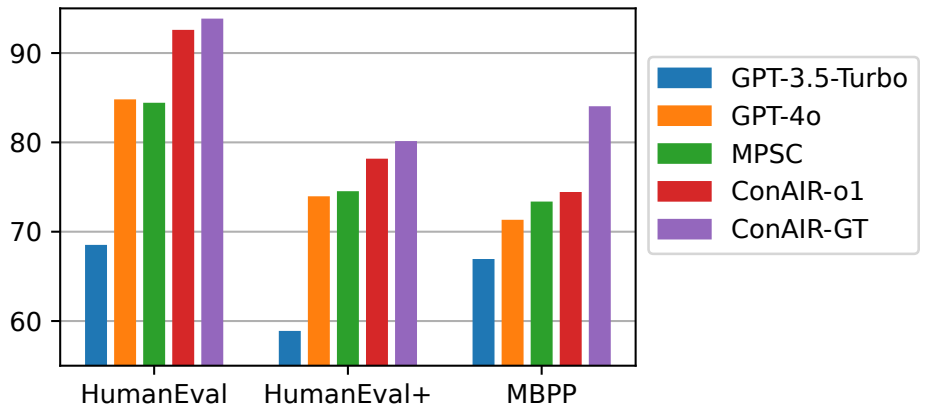


Fig. 1. ConTested built on GPT-3.5 surpasses the SOTA general LLM GPT-4o and the SOTA post-processing technique MPSC on all datasets.

consistency-based techniques lack post-processing steps and simply select a single output. However, the candidate codes might still need additional adjustments.

In this work, we propose ConTested, i.e., Consistency-Aided Tested Code Generation, which introduces a lightweight interaction framework to gather user feedback and a co-evolution process to enhance both code and test quality iteratively. ConTested has two ingredients that distinguish it from existing approaches. *Firstly, it involves developers as the ultimate oracles.* Since both the code and tests are generated by LLMs, there may be errors in either, making it difficult to assess the correctness of the tests based solely on these generated components. Therefore, integrating user feedback is essential to improve the reliability of LLM outputs. To minimize developer efforts, ConTested focuses their involvement on correcting test cases, which is generally easier than fixing the code itself. Additionally, we aim to reduce the number of feedback rounds by leveraging consistency voting from code to tests ($\mathsf{Con}_{c\to t}$), helping to identify tests most likely to contain errors. As a result, the average number of iteration rounds is reduced to just 4, achieving a 33% improvement compared to the baseline model. *Secondly, we propose a dynamic strategy to fix and maintain a set of consistent tests and code candidates.* ConTested operates iteratively through a rank-correct-fix co-evolution process, where both code and test quality are gradually enhanced. This incremental improvement enhances the accuracy of both the consistency voting from code to tests and the consistency voting from tests to code, facilitating more reliable code selection and more precise identification of incorrect tests. The co-evolution process terminates once a code that passes all tests is identified. Due to the high quality of the tests, the resulting code is inherently more robust and reliable.

We conduct a comprehensive evaluation of ConTested. First, we assess its effectiveness in enhancing LLM code generation. With a suboptimal model, GPT-3.5, ConTested achieves an average improvement of 32.9% over GPT-3.5, an 11.1% improvement over the state-of-the-art post-processing technique, MPSC[14], and a 12.32% improvement over the most advanced LLM, GPT-4o, as shown in Fig.1. This improvement is achieved with only a 4-round interaction with users, requiring minimal user effort. Additionally, ConTested demonstrates consistent improvements when built on the state-of-the-art LLM GPT-4o (by 16.97%) and even on the reasoning LLM OpenAI o1 (by 8.85%). We also conduct a user study and examine the overhead of ConTested in terms of time and cost.

Problem Instruction:
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
```

Ground Truth Solution:
```
def can_arrange(arr):
    ind=-1
    i=1
    while i<len(arr):
      if arr[i]<arr[i-1]:
        ind=i
      i+=1
    return ind
```

(a) Problem Instruction

First-Ranked Solutions:
```
def can_arrange(arr):
    n = len(arr)
    largest_index = -1
    for i in range(1, n):
        if arr[i] < arr[i-1]:
            largest_index = i-1
    return largest_index
```

Second-Ranked Solutions:
```
def can_arrange(arr):
    last = arr[0]
    index = -1
    length = len(arr)
    for i in range(1,length):
        if arr[i] >= last:
            last = arr[i]
        else:
            index = i - 1
    return index
```

(b) Top-Ranked and Second-Ranked Solutions

Fig. 2. The motivating example for limitations of relying solely on consistency (HumanEval/135)

In summary, this paper makes the following contributions:

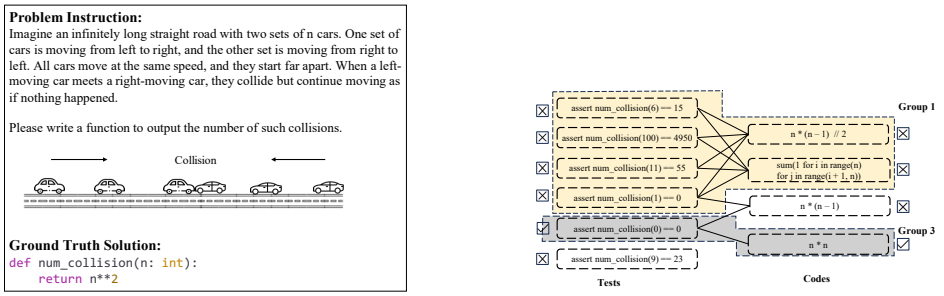
- **A lightweight interaction framework**, which incorporates the user feedback to correct the identified tests and guide the iterative process, resulting in an improvement of 33% towards the base model and 12% towards GPT-4o with only 4 rounds of iteration.
- **A rank-correct-fix co-evolution process** that leverages two forms of consistency voting, with gradually improving code and tests that enhance the reliability of consistency.
- **A finding on consistency techniques** that they often overlook preconditions when applying consistency, which can lead to unreliable results.
- **A comprehensive evaluation**, which evaluates ConTested from both quantitative experiments and a user study.

2 Motivation

In this section, we introduce the motivation for our work, highlighting that consistency can effectively enhance the reliability of LLM-generated outputs. However, relying solely on consistency is insufficient. First, although the most consistent outputs are more likely to be correct, they may still contain issues. Second, if the consistency indicators used to assess consistency are of low quality, the resulting consistency will be unreliable.

2.1 Effectiveness of Consistency and Limitations of Relying Solely on It

Firstly, we discuss the effectiveness of consistency and the limitations of relying solely on it. In Fig. 2a, we present the instructions for HumanEval/135, that is, "finding the largest index of an element that is less than the previous element." In Fig. 2b, we display the solutions generated by LLMs. The left side represents the solution set with the highest consistency. This set contains the most functionally equivalent counterparts and passes the majority of tests. On the right are other, less consistent solutions. The first-ranked solution on the left is very close to the ground truth solution; its overall logic is correct, but the selected index is slightly off. In contrast, the solution on the right is algorithmically incorrect. This demonstrates the effectiveness of consistency in helping us identify solutions that are more likely to be correct.



(a) Problem Instruction (b) Testing Results

Fig. 3. The motivating example illustrating the consequences of ignoring the prerequisite required for consistency (HumanEval/41)

Although consistency can increase confidence in selecting correct solutions, relying solely on it is insufficient. While the first-ranked solutions are close to correct, none are entirely accurate; in fact, all generated code for this problem is incorrect. Therefore, beyond consistency, additional post-processing of the generated code is necessary. All existing consistency-based techniques lack post-processing steps and simply select a single output. In this paper, we focus on fixing the code using tests corrected with user feedback. In this example, after just two correction steps, we obtain accurate and consistent codes. Notably, 66.7% of the final correct solutions originate from the initially top-ranked group, as nearly correct code is easier to fix than entirely incorrect code.

2.2 Consequences of Neglecting Consistency Preconditions

In addition to the need for further processing of outputs obtained through consistency, there are instances where consistency leads to incorrect results when the consistency indicator (in this case, tests) are of low quality. Existing techniques frequently overlook these prerequisites for effectively utilizing consistency. Specifically, the assessment for consistency relies on relatively good-quality consistency indicators; without this, consistency achieved through inaccurate consistency indicators is unreliable, leading to potentially incorrect outputs. To illustrate, consider an example from HumanEval [7], as shown in Fig. 3 (HumanEval/41). The left side (Fig. 3a) presents the problem description and a ground truth solution, while the right side (Fig. 3b) shows simplified testing results. In this case, the top-ranked group, containing 34 tests and 44 codes (simplified as Group 1 in Fig. 3b), ranks highest by consistency, while a group of 2 tests and 2 codes (simplified as Group 3) ranks lowest. To simplify the graph, we omit Group 2, which consists of the 3-rd code and the 4-th and 5-th tests. The highest-ranked group passes the most tests and contains the most functionality-equivalent counterparts, selected as the final output by existing techniques[5, 14]. However, all codes in this group are incorrect, while the correct codes are ranked last. This mistake arises because most tests passed by the top group (32 out of 34) are incorrect. Fig. 3b illustrates this with a simplified testing results: here, Group 1 of 4 tests and 2 codes are all incorrect, while the correct code "`n ** 2`" only passes one test. Therefore, when test quality is low, majority voting based on testing results is unreliable, a limitation overlooked by current methods. Existing techniques[5, 14] that rely solely on consistency are unable to select correct answers due to the lowest consistency level of the correct answers. Thus, incorporating user feedback is essential to enhance test quality and reliability.

We leverage consistency voting from codes to tests $\mathsf{Con}_{c \to t}$ to identify the most inconsistent test—that is, the test that most codes fail to pass. In this example, the identified test is the sixth

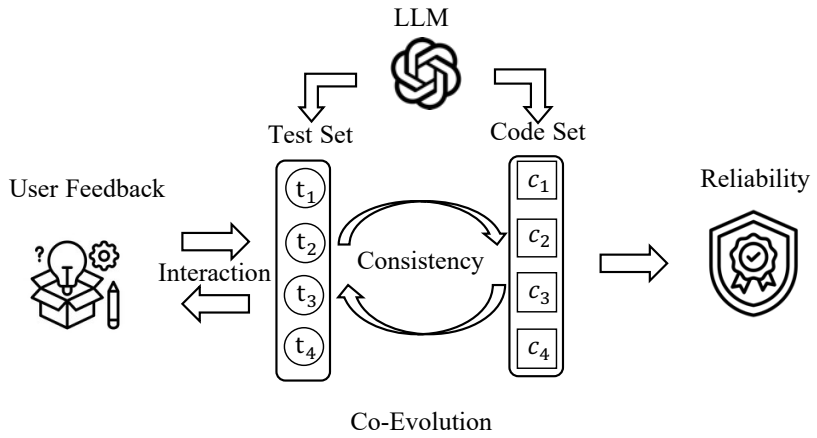


Fig. 4. The overview of ConTested

test in Fig. 3b. We then prompt users to correct the output, which they adjust from "23" to "81" based on the instruction requirements. With this corrected test, we can identify the codes that fail it and fix these codes. Here, the initially top-ranked codes fail this test, while the lowest-ranked codes pass. After this fixing, codes that cannot be fixed are discarded. With just one step of user feedback and code fix, all remaining codes align perfectly, producing consistent outputs across all tests. ConTested demonstrates promise by achieving correct results with only minimal human interaction.

3 Approach

In this paper, we introduce a new consistency-aided technique, ConTested, which incorporates a lightweight interaction framework to gather user feedback and a co-evolution process to iteratively enhance the quality of both tests and codes. An overview of the workflow of ConTested is illustrated in Fig. 4. ConTested has two ingredients that distinguish it from existing approaches. Firstly, the developers are involved as the ultimate oracle, and we propose a lightweight interaction framework that incorporates user feedback to correct the identified tests and guide the iterative process. Secondly, ConTested uses a rank-correct-fix co-evolution process in each iteration to gradually improve the quality of both code and tests, which makes $\mathsf{Con}_{c\to t}$ and $\mathsf{Con}_{t\to c}$ increasingly reliable. In the end, we achieve a more reliable code through improved consistency. In the following, we introduce the details of ConTested. Specifically, we will introduce the task definition in Section 3.1, the overall interaction framework in Section 3.2, and the specific co-evolution process in each iteration in Section 3.3.

3.1 Task Description

Assumption. Firstly, the use scenario of ConTested is that when the users have a requirement in natural language and leverages ConTested to help implement the code. Following the standard practice outlined by the HumanEval benchmark creators [7] and all existing techniques [5, 13–15, 19, 21, 26, 27, 30], we assume the problem description d and the method signature s are provided as input for ConTested. Developers often begin by defining method signatures. It is essential to clearly define the input and output types. This aids in the creation of test cases, clarifies the overall program structure and the scope of each function, and enhances communication among

team members. Although natural language descriptions can serve this purpose, using a method signature is more precise and straightforward. Secondly, each benchmark provides a hidden test set to assess the correctness of the generated code, and these tests remain unseen during the code generation process. The hidden tests are only used to assess the results, and revealing them would expose the ground truths to LLM. The problem we aim to solve is code generation, where the input is solely a natural language problem description. This typical code generation task has been addressed in many prior works [5, 13–15, 19, 21, 26, 27, 30]. Therefore, there are no ground truth test cases provided as input. If we choose to use tests to assist in generating code, we would need to generate these tests ourselves based on the problem description, similar to the SOTA techniques MPSC, CodeT, and our proposed approach.

Problem Definition. We introduce the definition and setup of this task, that is, leveraging consistency to improve LLM code generation results. First of all, in our paper, "code" refers specifically to production code and does not include test case code. "Test" refers to test case code, typically in the form of assertions, such as "assert method_name(...) == ...". The code generation task aims to generate a code solution, c, based on a problem description, d, using a large language model, $\mathcal{M}$. Formally, this is represented as $c = \mathcal{M}(d)$. The problem description, d, provides the requirements in natural language and includes the function signature, specifying the function name and parameters, as shown in the example in Fig 2a. Generating correct code in a single attempt is challenging for LLMs [5, 14]. To address this, researchers propose sampling multiple code solutions from LLMs, denoted as $\mathbf{C} = \{c_1, c_2, ..., c_n\}$, and obtaining a code $\hat{c}$ based on $\mathbf{C}$ that is more likely to be correct. In addition to generating code, researchers also use the same LLM, $\mathcal{M}$, to generate a set of tests, $\mathbf{T} = \{t_1, t_2, ..., t_m\}$, to aid in obtaining the best code, $\hat{c}$. These tests, $\mathbf{T}$, serve as consistency indicators to evaluate the consistency of the generated codes, $\mathbf{C}$. A test case t is defined as a pair of input and expected output (i.e., $t = (x, y)$), which verifies whether the output of the code c meets the requirements specified in the problem description d. Existing consistency-based LLM code generation methods [5, 14] work by selecting the code that passes the most tests (inter-consistency) and has the highest number of functionally equivalent counterparts (intra-consistency). However, this approach can be problematic: while tests help verify code correctness, they may also be incorrect since they are generated by the same LLM. In this paper, we introduce a new consistency-augmented technique, ConTested, which incorporates an interaction framework to gather user feedback $\mathbf{F}= \{f_1, f_2, ..., f_k\}$ and a co-evolution process to iteratively enhance the quality of both tests and codes. In our approach, the input and output are defined as $\hat{c}$ = ConTested($\mathbf{C}$, $\mathbf{T}$, $\mathbf{F}$).

3.2 Lightweight Interaction Framework to Gather User Feedback

In this section, we present the overall framework of ConTested. ConTested is a lightweight interaction framework that collects user feedback to correct identified tests and guide the iterative improvement process. To satisfy the prerequisite necessary for consistency, we incorporate user feedback in a developer-friendly, lightweight manner. In our task, tests act as consistency indicators that verify code correctness and assess consistency. However, our experiments reveal that 37.7% of tests generated by the LLM are incorrect. With only the tests $\mathbf{T}$ and the codes $\mathbf{C}$, we cannot ensure test accuracy or determine their correct outputs, as both tests and codes are generated by the same LLM, making them potentially unreliable. Additionally, in many cases, only the developer can determine the correctness of specific test results. In real-world development, users must ensure the correctness of tests; otherwise, the quality of the code may be compromise. Keeping the developer in the loop also fosters a sense of code ownership and ensures that they maintain a clear understanding of the code generation process, enabling them to respond more swiftly to future bugs. Therefore, integrating user feedback is essential—a need also supported by many studies [8, 25].

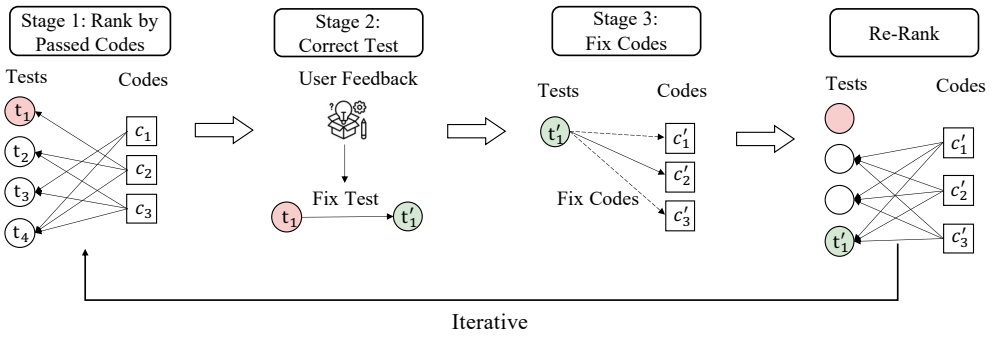


Fig. 5. The co-evolution process of ConTested

However, it is crucial to minimize the need for developer consultation. First, we aim to simplify the questions users need to answer. Therefore, ConTested prompts users to check and correct the tests rather than directly fixing the code. In other words, the feedback we ask users to provide is the correction to the tests. As shown in Fig. 7a, we provide the prompt to correct the tests. Given the problem description and input, users only need to provide the correct output as feedback. Second, we limit the user check to the fewest possible rounds. To achieve this, we use consistency voting from codes to tests ($\mathsf{Con}_{c\to t}$) to identify tests that are passed by fewer codes, as these are more likely to contain errors. Following existing work [5, 14], we formalize the consistency relationship between code c and test t as

$$\mathsf{Con}(c,t) = \mathsf{Con}(c,(x,y)) = \begin{cases} True, c(x) = y \\ False, c(x) \neq y \end{cases} \tag{1}$$

Tests and codes are implementations of the same problem requirements from two different perspectives. The code c and the test t are considered consistent when c passes t, indicating that the functionality aligns from both perspectives. The consistency voting $\mathsf{Con}_{c\to t}$ represents the degree of consistency between each test t and the entire code set $\mathbf{C}$, serving as a measure of the test's reliability from the perspective of the codes and is denoted as

$$\mathsf{Con}_{c\to t}(t,\mathbf{C}) = \sum_{c} \mathsf{Con}(t,c) \tag{2}$$

The lower the consistency voting ($\mathsf{Con}_{c\to t}$), the more likely it is that the test is incorrect. We rank the tests based on $\mathsf{Con}_{c\to t}$ and select the most inconsistent test for correction, which can enhance the quality of both the test set and code set and thus reduces the number of iteration rounds. Additionally, when the test is consistent with all codes—i.e., $\mathsf{Con}_{c\to t}(t,\mathbf{C}) = \text{size}(\mathbf{C})$—we skip the correction process and directly use the code outputs as the test output. Although individual codes may be incorrect, the likelihood of all codes producing the same incorrect result is low. In addition, as iterations proceed, code quality steadily improves, making consistency voting increasingly reliable. By leveraging these two methods, we decrease the number of iteration rounds and reduce the need for human feedback. With the support of consistency, the average number of iteration rounds is reduced to just 4, with an improvement of 33% to the base model.

3.3 Co-Evolution Process between Codes and Tests

In Section 3.2, we introduced the overall interaction framework. Here, we detail the specific process within each iteration. Each iteration leverages a rank-correct-fix co-evolution process between

codes and tests to iteratively improve their quality, as illustrated in Fig.5. The co-evolution algorithm is outlined in Algorithm 1.

Before initiating the co-evolution process between code and test cases, we first need to generate a set of tests, denoted as **T**, and a set of codes, denoted as **C**. As shown in Algorithm 1, these sets serve as the input for the subsequent co-evolution process. The prompts used for generating these codes and tests are illustrated in Fig 7. To ensure a fair comparison, we employ the same prompts as those used in the state-of-the-art post-processing techniques. These prompts are straightforward and directly instruct the models to generate tests and codes. The prompt for generating code consists of two parts: the first part is an instruction for LLM to generate codes, and the second part is the problem description. When generating tests, an additional component is included: the format for the test, which is specified as "*assert method_name(...) == ...*". We adopt a high temperature for LLM (0.8 in this paper) to ensure that the models generate sufficiently diverse codes and tests.

During the co-evolution process, we employ two types of consistency voting. The consistency voting from codes to tests ($\mathsf{Con}_{c\to t}$) identifies tests most likely to be erroneous, as discussed in Section 3.2. The consistency voting from tests to codes ($\mathsf{Con}_{t\to c}$) selects the code consistent with all tests and, therefore, most likely to be correct—i.e., the code that passes all tests, serving as the process's termination condition. Through this interaction, codes and tests co-evolve, enhancing each other. The tests help identify erroneous cases, and once corrected, they further aid in refining the codes. As the quality of both codes and tests improves, the reliability of $\mathsf{Con}_{c\to t}$ and $\mathsf{Con}_{t\to c}$ increases. Higher-quality codes make it more likely that tests they cannot pass are buggy, while improved tests increase the likelihood that codes passing more tests are correct.

Next, I will provide a detailed introduction to the co-evolution algorithm, using Algorithm 1 and Fig. 5 to offer a more intuitive explanation. As shown in Algorithm 1, the input consists of the test set **T** and code set **C** generated by the LLM $\mathcal{M}$, with the output being a more reliable and accurate code, $\hat{c}$, than any individual code in **C**. Before the algorithm begins, each code is executed on each test to gather execution information. We obtain the information whether each code can pass each test. The entire execution process includes both compiling and executing the code, and failure at either stage results in an overall failure. This process is efficient, as it can be parallelized. Next, we compute the consistency voting between codes and tests, as illustrated in Equation 1 and Equation 2. The main body of the algorithm consists of iterations, where tests are divided into two groups: *corrected tests*, $\mathbf{T}_{\text{cor}}$, and *unknown tests*, $\mathbf{T}_{\text{unk}}$. The iteration terminates once all unknown tests are corrected or a code that passes all tests is found. Each iteration involves a rank-correct-fix co-evolution process between codes and tests, divided into three stages: "ranking", "correcting", and "fixing".

Ranking. The first stage, "ranking," is outlined in Line 3 of Algorithm 1. As already introduced in Section 3.2, we leverage the consistency voting from codes to tests (i.e., $\mathsf{Con}_{c\to t}$) to rank the tests and identify the one most likely to be incorrect, i.e., the test passed by the fewest codes, denoted as t_{w}. As the co-evolution process iterates, the quality of codes improves, leading to increasingly accurate identification of incorrect tests. As shown in Stage 1 of Fig. 5, t_4 has three codes that pass it, t_3 and t_2 have two codes that pass each, and t_1 only has one code that passes it. Based on the consistency voting from codes to tests, $\mathsf{Con}_{c\to t}$, t_1 is most inconsistent with the codes, making it more likely to contain an error and thus selected.

Correcting. In Line 4-5, describes the second stage, "correcting", where we utilize an interaction process to incorporate user feedback. Specifically, the feedback we request from users is the correction of the tests. Given the problem description and input, users only need to provide the correct output as feedback. Once corrected, t_{w} is updated to t_{cor}. t_{cor} will then be removed from the unknown tests group $\mathbf{T}_{\text{unk}}$ and added to the corrected tests group $\mathbf{T}_{\text{cor}}$. As shown in Stage 2 of

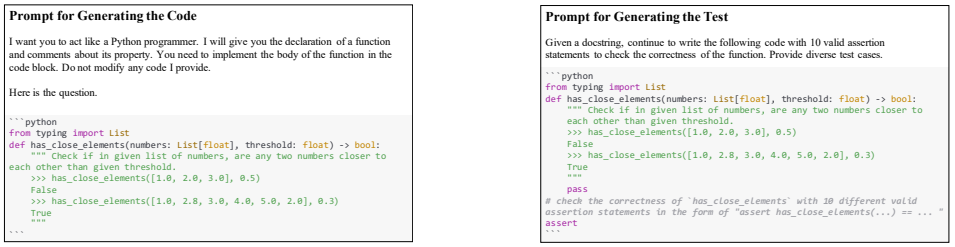
Algorithm 1: Co-Evolution Algorithm between Codes and Tests

Input: test case set $\mathbf{T}$; code set $\mathbf{C}$

1 $\mathbf{T}_{\text{unk}} \leftarrow \mathbf{T}$; $\mathbf{T}_{\text{cor}} \leftarrow \{\}$; $\mathbf{C}_{\text{dis}} \leftarrow \{\}$; ▷ initializing sets
2 **while** *true* **do**
3 $t_{\text{w}} \leftarrow \text{argmax}_{t \in \mathbf{T}_{\text{unk}}} \text{Con}_{c \rightarrow t}(t, \mathbf{C})$; ▷ rank and localize a worst test case
4 $t_{\text{cor}} \leftarrow \text{InteractivelyCorrectTestCase}(t_{\text{w}})$; ▷ interactively correct the worst test case
5 $\mathbf{T}_{\text{unk}}$.remove(t_{w}); $\mathbf{T}_{\text{cor}}$.add(t_{cor});
6 $\mathbf{C}_{\text{rem}} \leftarrow \{\}$;
7 **for** $c \in \mathbf{C}$ **do**
8 **if** $\text{Con}(t_{cor}, c)$ **then**
9 $\mathbf{C}_{\text{rem}}$.add(c);
10 **else**
11 $c' \leftarrow \text{LLMFixCode}(c)$; ▷ fix the code by LLM
12 **if** $\text{Con}_{t \rightarrow c}(\mathbf{T}_{cor}, c') = \text{size}(\mathbf{T}_{cor})$ **then**
13 $\mathbf{C}_{\text{rem}}$.add(c'); ; ▷ the fixed code can pass all the corrected tests
14 **else**
15 $\mathbf{C}_{\text{dis}}$.add(c');
16 **if** $\mathbf{C}_{rem}$.isEmpty() **then** ▷ if no code passes corrected test cases
17 **return** $\text{argmax}_{c \in \mathbf{C}_{\text{dis}}} \text{Con}_{t \rightarrow c}(\mathbf{T}_{\text{unk}} \cup \mathbf{T}_{\text{cor}}, c)$;
18 $\mathbf{C} \leftarrow \mathbf{C}_{\text{rem}}$;
19 **if** $\mathbf{T}_{unk}$.isEmpty() **then** ▷ if all test cases are corrected
20 **return** $\text{argmax}_{c \in \mathbf{C}} \text{Con}_{t \rightarrow c}(\mathbf{T}_{\text{cor}}, c)$;
21 **for** $c \in \mathbf{C}$ **do** ▷ find a code that passes all test cases
22 **if** $\text{Con}_{t \rightarrow c}(\mathbf{T}_{unk} \cup \mathbf{T}_{cor}, c) = \text{size}(\mathbf{T}_{unk} \cup \mathbf{T}_{cor})$ **then**
23 **return** c;

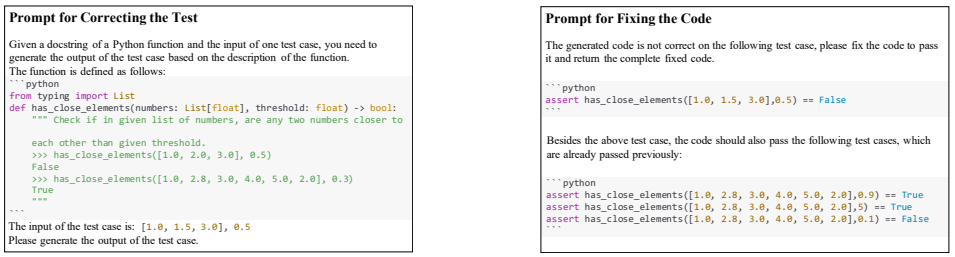
Fig. 5, t_1 is updated to $t_1^{'}$. This test is then removed from the unknown set $\mathbf{T}_{\text{unk}}$ and added to the corrected set $\mathbf{T}_{\text{cor}}$. Once corrected, we also update the consistency relationships between tests and code. This stage produces the first update of the consistency voting results, making the consistency voting from tests to code $\text{Con}_{t \rightarrow c}$ more reliable. As shown in Fig.7a, we present the prompt used to correct tests. In Section 4.1, we will introduce a setting where OpenAI o1 is used to simulate user feedback (i.e., correcting the tests), with this prompt provided to OpenAI o1. The prompt consists of three parts: (1) The first part contains instructions asking LLM to generate the test output based on the input. (2) The second part provides the problem description, and we do not include a concrete implementation. (3) The third part includes the test input, with the LLM being asked to generate the output based on the problem description.

Fixing. The third stage, "fixing", described in Lines 7-15 of Algorithm 1, involves fixing the codes that fail on the corrected test t_{cor}. For this, we employ the same model $\mathcal{M}$ that initially generated codes to fix the codes, allowing us to demonstrate the improvements brought by ConTested. As shown in Stage 3 of Fig. 5, c_1, c_2, and c_3 initially fail to pass $t_1^{'}$. We leverage $\mathcal{M}$ to fix them, resulting in the corrected versions $c_1^{'}$, $c_2^{'}$, and $c_3^{'}$. The prompt for the model to fix the code is presented in Figure 7b. We provide the test t_{cor} that the initial code c fails to pass, and instruct the model to modify the code so it can successfully pass t_{cor}. Additionally, if there are other tests that have already been corrected, denoted as $\mathbf{T}_{\text{cor}}$, we include these in the prompt as well. The model is



(a) Prompt for generating codes (b) Prompt for generating tests

Fig. 6. Prompts for generating codes and tests (the example data is from HumanEval/1)



(a) Prompt for correcting tests (b) Prompt for fixing codes

Fig. 7. Prompts in the co-evolution process between codes and tests (the illustrative problem is from HumanEval/1)

expected to ensure that the modified code passes all of these tests. After the fix process, we obtain the fixed version of the code, denoted as $c^{'}$.

We check whether the fixed code $c^{'}$ can pass all tests in the corrected set $\mathbf{T}_{\text{cor}}$ (Line 12). We divide the code set $\mathbf{C}$ into *discarded codes* $\mathbf{C}_{\text{dis}}$ and *retained codes* $\mathbf{C}_{\text{rem}}$. If the fixed code $c^{'}$ cannot pass $\mathbf{T}_{\text{cor}}$, it is removed from $\mathbf{C}_{\text{rem}}$ and added to $\mathbf{C}_{\text{dis}}$. This process highlights the advantage of introducing diversity in the generated codes. Some implementations may contain fundamental logical errors that are difficult to fix, while others may only fail on certain edge cases, making them fixable. Diversity thus enhances the chances of obtaining a viable code solution. If all codes end up in $\mathbf{C}_{\text{dis}}$, indicating the absence of a code that fully satisfies all tests, we select and output the code from $\mathbf{C}_{\text{dis}}$ that passes the most tests (Line 16-17).

After fixing all codes, we execute each fixed code c_{fix} on the unknown test set $\mathbf{T}_{\text{unk}}$ and update the consistency relationships to re-rank the tests. This produces the second update of the consistency relationship. After updating, if we identify a code that passes all tests, we output this code as $\hat{c}$ (Line 21-23). With the improved quality of tests, consistency voting from tests to code becomes increasingly reliable, meaning a code that passes all tests is more likely to be correct. If no such code exists, we proceed to the next iteration. On the other hand, as the quality of the codes improves, the consistency voting from code to tests becomes more reliable, enabling more precise identification of incorrect tests in subsequent iterations. As shown in the "re-rank" section of Fig. 5, t_3 has no passing codes, while other tests are passed by all codes. Therefore, t_3 is selected for further inspection, initiating the next iteration.

4 Experimental Setup

4.1 Research Questions

We begin by evaluating the performance of ConTested through two automated simulated experiments: one where user feedback is simulated using LLM OpenAI o1, which has advanced reasoning capabilities, and another where feedback is simulated via the ground truth solution. Following these, we examine the user efforts required to use ConTested and conduct a user study to assess the user experience in a real interactive setting. Lastly, we analyze the overhead of ConTested.

- **RQ1. Overall Effectiveness** How effective is ConTestedin improving LLM code generation? We conduct two simulated experiments to automatically evaluate ConTested, enabling extensive quantitative analysis. In the first experiment, OpenAI o1 is used to simulate providing user feedback, while in the second experiment, the ground truth solution is used to simulate providing user feedback.
- **RQ2. User Efforts and User Study** What is the user effort required by ConTested? We provide and discuss the user effort required by ConTested in the experiments. We further report the user study here, which assesses the user experience in a real interactive setting.
- **RQ3. Time and Cost Overhead** What is the overhead of ConTested? We analysis the time and cost of ConTested.

4.2 Dataset

We adopt three standard code generation datasets that are widely-used by code generation techniques [2, 11–13, 15, 19, 21, 24]. HumanEval [7] and MBPP [4] comprise a set of hand-crafted Python programming problems. For each problem, the dataset provides the problem description in natural language, the tests to check the correctness of given output, and the ground truth solution. HumanEval+ [20] adds more tests to the HumanEval dataset, which makes the check more strict. The statistics of the three datasets are as follows. The number of problems in HumanEval, HumanEval+, and MBPP are 164, 164, 427 respectively. The average number of tests in three datasets are 7.77, 764.74, 3.1 respectively.

4.3 Evaluation Metrics

Following all the code generation techniques, we use the Pass@k metric. For each problem, we may generate multiple code solutions and select the top k as the final candidates. Among these k solutions, if at least one successfully passes all the tests for the problem, the problem is considered solved. Pass@k represents the ratio of successfully solved problems to the total number of problems.

4.4 Compared Techniques

Firstly, we compare ConTested with OpenAI models, including GPT-3.5, and the most advanced general model, GPT-4o. Additionally, we compare with other LLMs for code generation, such as Code Llama [30], WizardCode [21], and Deepseek Code [13]. Since ConTested functions as a post-processing technique for outputs generated by LLM, we also evaluate it against other post-processing methods, including the state-of-the-art MPSC[14] and CodeT[5]. These two techniques are also based on consistency. CodeT uses tests as consistency indicators, selecting the code that satisfies the most tests and has the highest number of functionally equivalent counterparts based on test results. MPSC further incorporates specifications as additional consistency indicators; however, the benefits brought by specification are limited. Other post-processing techniques in the comparison include Self-Consistency [39], MBR-EXEC [31], and Self-Collaboration [10].

4.5 Implementation

In our experiments, we choose GPT-3.5, GPT-4o and OpenAI o1 as the base foundation model. We leverage the same model to generate codes and tests, and also fix the codes. ConTested achieves a consistent improvement on three base models. To keep consistent with the state-of-the-art technique MPSC, the GPT-3.5 version we adopt is gpt-3.5-turbo-0613 API, which is released on 2023-06-13. GPT-3.5 is an old and less powerful model, and the context size is only 4,096. The GPT-4o version we adopt is gpt-4o-2024-08-06, which is the latest and most powerful general LLM. The context size is 128,000. The temperature for the models is set to 0.8, allowing the model to generate sufficiently diverse code solutions and tests. OpenAI o1 [28] is the latest LLM released by OpenAI, noted for its strong reasoning abilities and its use of chain-of-thought processes to enhance generation quality. ConTested can achieve consistent improvement on both poor and powerful model, which indicates the effectiveness of ConTested. The performance of ConTested based on GPT-3.5 is even better than GPT-4o. In RQ1 (Section 5.1), we also investigate scenarios where user feedback is simulated with OpenAI o1. OpenAI o1 can solve the reasoning and logic problems with chain-of-thought, which can simulate humans. The version we use is o1-preview-2024-09-12.

5 Results and Analysis

In this section, we present the experimental results along with an in-depth analysis. First, we conduct two simulated experiments to automatically evaluate ConTested's effectiveness, as detailed in Section 5.1. Next, we discuss the user effort involved in using ConTested during these experiments and include a user study assessing user experience in a real interactive setting in Section 5.2. Finally, we analyze the time and cost overhead associated with ConTested in Section 5.3.

5.1 RQ1: Overall Effectiveness

5.1.1 Experimental Design. In this paper, we propose a lightweight interaction framework to gather user feedback, which enhances the quality of consistency indicator by asking users to validate the correctness of selected tests. In this research question (RQ1), we investigate the effectiveness of ConTested through large-scale, automated experiments. These experiments simulate user feedback, enabling extensive quantitative analysis. In Section 5.2 (RQ2), we conduct a user study to evaluate ConTested in a real interactive setting. Given the limitations in scaling the user study to a large scale, we leverage simulated user interactions in RQ1. Specifically, we use two simulation methods: (1) OpenAI o1 is used to simulate providing user feedback, (2) ground truth solution is used to simulate providing user feedback, which can be regarded as *novice users* and *experienced users* respectively. Accordingly, the two variants of ConTested are referred to as $\text{ConTested}_{\text{o1}}$ and $\text{ConTested}_{\text{GT}}$, respectively.

Experienced users rarely make mistakes, while novice users are more prone to errors. In real development scenarios, tests should be error-free, as they serve to ensure code correctness. They are typically provided by users, who are responsible for ensuring their accuracy. If tests contain errors, the quality of the generated code is compromised. Additionally, it is often easier for users to provide expected test outputs rather than writing code, as they already have a clear understanding of the requirements. In simulations using OpenAI o1, we employ OpenAI o1 to correct the identified tests. This process is fully automated, requiring no user involvement. Therefore, $\text{ConTested}_{\text{o1}}$is an automated variant of ConTested. In simulations using ground truth solution, we utilize the ground truth solution provided by the benchmark to execute the identified tests and generate ground truth outputs as user feedback.

For the base model, we ensure fair comparison by following the setup in SOTA[14], using GPT-3.5 for code generation and fixing, as shown in Table 1. Besides, we also build ConTested based on the

Table 1. The performance of ConTested and other baselines on three benchmarks. The best and second-best performances for each dataset are highlighted in **bold** and underlined, respectively.

Benchmark	**HumanEval**			**HumanEval+**		
Metric	**Pass@1**	**Pass@2**	**Pass@5**	**Pass@1**	**Pass@2**	**Pass@5**
GPT-4o	84.67	89.69	92.50	73.82	81.03	85.80
GPT-3.5-Turbo	68.38	76.24	83.15	58.75	66.58	73.96
DeepSeekCoder	79.30	-	-	-	-	-
WizardCoder	73.20	-	-	-	-	-
Code Llama	62.20	-	-	-	-	-
Self-consistency	$73.86_{+5.48}$	$73.93_{-2.31}$	$74.10_{-9.05}$	$63.50_{+4.75}$	$64.70_{-1.88}$	$65.67_{-8.29}$
MBR-EXEC	$72.96_{+4.58}$	$76.47_{+0.23}$	$79.00_{-0.45}$	$62.12_{+3.37}$	$67.08_{+0.50}$	$71.38_{-2.58}$
CodeT	$78.05_{+9.67}$	$78.05_{+1.81}$	$78.30_{-4.85}$	$67.87_{+9.12}$	$68.75_{+2.17}$	$69.65_{-4.31}$
Self-collaboration	$74.40_{+6.02}$	-	-	-	-	-
MPSC	$84.29_{+15.91}$	$86.79_{+10.55}$	$87.13_{+3.98}$	$74.39_{+15.64}$	$76.66_{+10.08}$	$77.25_{+3.29}$
ConTested$_{o1}$ (GPT-3.5-Based)	$\underline{92.45}_{+24.07}$	$\underline{94.45}_{+18.21}$	$\underline{95.71}_{+12.56}$	$\underline{78.03}_{+19.28}$	$\underline{82.65}_{+16.07}$	$\underline{86.35}_{+12.39}$
ConTested$_{GT}$ (GPT-3.5-Based)	$\mathbf{93.71}_{+25.33}$	$\mathbf{94.86}_{+18.62}$	$\mathbf{96.32}_{+13.17}$	$\mathbf{80.00}_{+21.25}$	$\mathbf{84.67}_{+18.09}$	$\mathbf{88.42}_{+14.46}$

Benchmark	**MBPP**		
Metric	**Pass@1**	**Pass@2**	**Pass@5**
GPT-4o	71.19	$\underline{76.49}$	$\underline{79.89}$
GPT-3.5-Turbo	66.80	72.34	76.60
DeepSeekCoder	70.00	-	-
WizardCoder	61.20	-	-
Code Llama	61.20	-	-
Self-consistency	$71.70_{+4.90}$	$71.73_{-0.61}$	$71.82_{-4.78}$
MBR-EXEC	$70.79_{+3.99}$	$73.14_{+0.80}$	$74.85_{-1.75}$
CodeT	$71.90_{+5.10}$	$71.95_{-0.39}$	$72.02_{-4.58}$
Self-collaboration	$68.20_{+1.40}$	-	-
MPSC	$73.23_{+6.43}$	$73.29_{+0.95}$	$73.55_{-3.50}$
ConTested$_{o1}$ (GPT-3.5-Based)	$\underline{74.30}_{+7.50}$	$74.53_{+2.19}$	$75.14_{-1.46}$
ConTested$_{GT}$ (GPT-3.5-Based)	$\mathbf{83.90}_{+17.10}$	$\mathbf{86.80}_{+14.46}$	$\mathbf{87.45}_{+10.85}$

most advanced general LLM, GPT-4o, and the latest reasoning LLM, OpenAI o1, where ConTested consistently shows performance improvements, as shown in Table 2. Due to the significant time and cost associated with OpenAI o1 (detailed in Section 5.3), we sampled 25 problems from HumanEval for code generation and fixing using OpenAI o1. OpenAI o1 is more suitable for complex logic and math tasks, and its cost makes it impractical for routine code generation.

5.1.2 Experimental Results. In Table 1, we present the effectiveness of ConTested across two simulated experiments. In these experiments, ConTested$_{o1}$uses OpenAI o1 to simulate feedback, while ConTested$_{GT}$ employs ground truth solution for feedback simulation. We conduct experiments on three datasets and compare the results with various baselines. These comparison techniques are grouped into three categories: the top part in Table 1 leverages only the LLM, the middle part applies post-processing techniques, and the bottom part represents our proposed approach. The increases shown to the right of the post-processing technique results indicate their improvements over the base model, GPT-3.5. As shown in Table 1, both variants of ConTested achieve SOTA performance across all benchmarks, demonstrating the effectiveness of ConTested. ConTested$_{GT}$ outperforms ConTested$_{o1}$, as the outputs assigned to tests in ConTested$_{o1}$ may be incorrect, reducing the model's performance. Built on a suboptimal model, GPT-3.5, ConTested achieves an average improvement of 32.9% over GPT-3.5, an 11.1% gain over the state-of-the-art post-processing technique, MPSC [14], and a 12.32% improvement over the most advanced LLM, GPT-4o.

Table 2. The performance of ConTested built on GPT-4o and OpenAI o1

Models	HumanEval			HumanEval+			MBPP		
	Pass@1	Pass@2	Pass@5	Pass@1	Pass@2	Pass@5	Pass@1	Pass@1	Pass@5
GPT-4o	84.67	89.69	92.5	73.82	81.03	85.80	71.19	76.49	79.89
ConTested$_{o1}$ (GPT-4o-Based)	94.96	96.10	97.64	81.95	86.4	90.9	75.36	75.94	76.20
ConTested$_{GT}$ (GPT-4o-Based)	97.59	99.15	99.03	85.50	88.42	91.58	85.30	86.56	88.13
OpenAI o1	90.96	92.59	93.95	93.60	95.63	96.00	-	-	-
ConTested$_{GT}$ (o1-Based)	100.0	100.0	100.0	95.26	99.95	100.0	-	-	-

Next, we compare the performance of ConTested$_{GT}$ and ConTested$_{o1}$. The results of ConTested$_{o1}$ on HumanEval and HumanEval+ are similar, indicating that the outputs generated by OpenAI o1 are close to the ground truth outputs. ConTested$_{o1}$ performs worse than ConTested$_{GT}$ on MBPP, but it still surpasses the SOTA baseline, MPSC. This difference arises because many of the outputs generated by OpenAI o1 on MBPP are incorrect. We compute the error rate of the outputs generated by OpenAI o1 for tests; the error rate on HumanEval and MBPP are 8.6% and 48.3%, respectively. Since the generated outputs guide code fixing to meet their specifications, any inaccuracies can degrade code quality. The HumanEval dataset includes comprehensive problem instructions, complete function signatures, and parameter and return value types, which aids understanding. Conversely, the MBPP dataset often lacks parameter and return types, and its problem descriptions can be misleading. In addition, there are no input-output examples to clarify requirements. This ambiguity brings challenges even for human understanding, as noted by participants in the user study. For instance, in MBPP/299, the instruction is "Write a function to calculate the maximum aggregate from the list of tuples," which is vague. Given the test input, [(1, 40), (2, 50), (3, 60), (1, 70), (2, 80), (3, 90)], it is particularly challenging to deduce the intended output based solely on the description.

Finally, to demonstrate the generalization ability of ConTested, we further implement it using more powerful LLMs, GPT-4o and OpenAI o1. The results are shown in Table 2. In the o1-based ConTested, since the code is already generated by OpenAI o1, we evaluate only ConTested$_{GT}$. As shown in Table 2, ConTested achieves notable gains with these advanced models as the base. For example, ConTested$_{o1}$ based on GPT-4o increases performance on HumanEval from 84.67 to 97.59. Specifically, built on GPT-4o, the pass@1 improvements of ConTested$_{o1}$ across three datasets are 12.2%, 11.0%, and 5.9%, while ConTested$_{GT}$ achieves 15.3%, 15.82%, and 19.8% improvements, respectively. When using OpenAI o1 as the base model, ConTested$_{GT}$ achieves improvements of 9.94% and 1.77%. These increases are smaller than those on GPT-4o due to the higher initial performance of OpenAI o1, which limits the room for further gains.

5.2 RQ2: Users Efforts and User Study

Since ConTested is designed as an interaction framework, it is essential to minimize the need for developer intervention. In this section, we focus mainly on the effort required from users. First, we show the interaction rounds in the simulated experiments to reflect user effort from a quantitative perspective. Additionally, we conduct a user study to evaluate ConTested in interactive scenarios to evaluate the efforts from user experience.

5.2.1 Experimental Design of User Study. In this section, we introduce the design of the user study. ***Settings.*** In our user study, there are three different settings. (1) *Writing Code.* In the first setting, given the problem instruction in natural language, the users are asked to implement the function completely. (2) *Fixing Code.* In the second setting, given the instruction, we will leverage the LLM to generate a code. Then the instruction and the code will be given to the users, and they will be asked to check the correctness of the code and fix it if any bugs exists. (3) *Fixing Tests.* The third

Table 3. The number of interaction rounds

Benchmark	HumanEval	HumanEval+	MBPP	Average
ConTested	2.83	4.80	5.95	4.53

setting is ConTested, and we will let the users use ConTested. Each iteration ConTested will identify a problematic test and ask the users to check it and fix it if any problem exist.

Metrics. During the three settings, we will record the time that the users spend on each setting and ask the users to give a difficulty score representing the difficulty degree to complete the task in each setting (1 is easiest, and 5 is highest). Finally, we will compute Pass@1 for the solutions obtained from each setting.

Participants. The implementation language for the three benchmarks is Python. Given the widespread use of code generation in enhancing coding efficiency across both academia and industry, we selected participants from both domains. Specifically, we selected 6 PhD students from universities and 6 industry developers working with Python from industry as participants. Of the 6 PhD students, 5 had prior industry experience in Python development through internships. The industry developers had an average of 2.83 years of working experience.

Prior to the experiment, participants were not informed about the problems in HumanEval and MBPP. To assess their familiarity with Python, we conducted a survey. Among the PhD students, their average Python experience is 4.83 years. When asked to rate their proficiency, the average score was 4 out of 5. The 6 PhDs had similar levels of Python experience. Additionally, 50% of the PhDs code almost every day, and 100% code at least 4 days a week. For the industry developers, the average Python experience was 6 years, with an average proficiency score of 4.5 out of 5. All of them code almost every day, and the 6 industry developers had similar levels of experience.

To ensure that all participants were familiar with the proposed method and the various experimental settings, we conducted a training session that covered the details of each setting. Only after confirming that all participants fully understood the settings did we proceed with the user study. Furthermore, participants were asked to rate their familiarity with the settings, and all provided a score of 5 out of 5.

Procedure. Considering the efforts for participants, we randomly[1] sample 40 examples, with 20 from HumanEval and 20 from MBPP. We asked both the academia and industry groups to complete the 40 problems individually. We will have three different users complete each of the three settings for a given problem to prevent any user from becoming familiar with the problem after working on one setting. The number of problems completed by each participant across different settings was approximately balanced(e.g., 7 problems in Setting-1, 7 problems in Setting-2, and 6 problems in Setting-3), with each participant working on 20 problems in total. Detailed task assignments for each participant are available on our website [1].

5.2.2 Experimental Results. Firstly, we present the interaction rounds in the simulated experiments in Table 3. The average number of rounds across all benchmarks is 4.53, indicating that users need to be consulted only four times on average to achieve a 33% performance improvement over the base model, which is a worthwhile efforts. HumanEval and HumanEval+ require only 2.83 and 4.80 rounds, a relatively low number. In contrast, MBPP involves more rounds, that is, 5.95. As discussed

[1]We selected these samples using a completely random method. Specifically, we used one of the most commonly used pseudo-random generators in C++ code, `std::minstd_rand`, with a random seed of 0 to sample from both HumanEval and MBPP.

Table 4. The results of user study

Groups	Settings	Pass@1	Time (s)	Difficulty (1-5)
PhDs	Writing Code	70.0	270.4	2.6
	Fixing Code	87.5	178.1	2.4
	Fixing Tests	**90.0**	**101.9**	**1.5**
Industry Developers	Writing Code	70.0	210.5	2.5
	Fixing Code	87.5	140.6	2.2
	Fixing Tests	**92.5**	**98.6**	**1.9**
Overall	Writing Code	70.0	240.5	2.6
	Fixing Code	87.5	159.4	2.3
	Fixing Tests	**91.3**	**100.2**	**1.7**

Table 5. Time and cost analysis of ConTested and OpenAI o1.

Models	Datasets	Performance			Avg Time (min)				Avg Cost ($)			
		Pass@1	Pass@2	Pass@5	Overall	Gen	Correct	Fix	Overall	Gen	Correct	Fix
ConTested$_{o1}$	MBPP	73.68	74.66	75.76	1.08	0.21	0.48	0.39	0.30	0.03	0.21	0.06
(GPT-3.5-based)	HumanEval	92.00	92.00	95.98	2.26	0.28	1.10	0.88	0.56	0.04	0.40	0.12
OpenAI o1	HumanEval	90.96	92.59	93.9	25.44	25.44	-	-	5.87	5.87	-	-

in RQ1, MBPP's instructions are somewhat ambiguous, making it more challenging for models to correctly refine the code, thus requiring additional tests for better understanding.

Next, we present the results of the user study, summarized in Table 4. We present the results for three groups: PhDs, Industry Developers, and Overall, with each group's performance averaged across 40 problems. Among the three metrics evaluated, ConTested (Fixing Tests) demonstrates the best performance for both the PhD group and the Industry Developers group. For Pass@1, time spent, and difficulty scores, users report the highest Pass@1, spend the least time on ConTested, and assign it the lowest difficulty scores, aligning with our expectations. This suggests that ConTested not only achieves the best performance but also offers the most efficient user experience with minimal effort. Additionally, ConTested proves effective for both academia and industry. Notably, industry developers tend to spend less time than PhDs across all three settings, indicating their superior coding abilities.

To confirm our observations, we further conducted the Wilcoxon signed-rank tests on time consumption and difficulty score between ConTested and the other two methods (i.e., writing code and fixing code). Regarding time consumption, the tests yielded a p-value of 3.4e-6 between ConTested and writing code (Setting-1) and a p-value of 7.3e-5 between ConTested and fixing code (Setting-2). As for the difficulty score, the p-value was 2.4e-4 between ConTested and writing code and 2.2e-3 between ConTested and fixing code. These results indicate that ConTested is statistically more time-efficient and less difficult to use, with both comparisons reaching significance at the 99% confidence level.

5.3 RQ3: Time and Cost Overhead

5.3.1 Experimental Design. In this RQ, we analyze the overhead of ConTested in terms of time and cost. The primary expense for ConTested arises from invoking the OpenAI API. The tool comprises three stages: generating code, correcting tests, and fixing code. Since the human efforts are already

studied in the user study, we mainly focus on evaluating the time and cost of ConTested$_{o1}$ in this RQ. We present results for HumanEval and MBPP only since the problem instructions in HumanEval+ and HumanEval are identical. To provide intuitive comparisons, we also report the time and cost of using OpenAI o1 solely without ConTested. Given the substantial costs of OpenAI o1, when evaluating OpenAI o1, we sample 25 problems, generating 50 code candidates per problem using OpenAI o1. Results are summarized in Table 5.

5.3.2 Experimental Results. As shown in Table 5, ConTested achieves an overall runtime of 1 to 2 minutes, with a total cost of only $0.30 to $0.56. Among the three tasks, test correction demands most time and cost, as it relies on OpenAI o1 and other two tasks use GPT-3.5. The OpenAI o1 API is approximately 20 times more expensive than the GPT-3.5 API. However, because ConTested requires only about four rounds of feedback, the overall cost of invoking OpenAI o1 remains manageable. In comparison, using solely OpenAI o1 results in an average runtime of 25.44 minutes and a cost of $5.87 per problem, exceeding ConTested's cost by over tenfold. Moreover, ConTested even outperforms OpenAI o1. Thus, ConTested efficiently achieves superior performance with minimal time and budget compared to the most advanced model alone.

6 Discussion

Integrating User Feedback We introduce a lightweight interaction framework to incorporate user feedback for test correction. Human involvement is essential for two main reasons. First, in many cases, only the developer can verify the correctness of specific test results. Since both tests and code are generated by LLMs, they may each contain inaccuracies, making it challenging to determine correct test outputs from these sources alone. In real-world development, developers must write tests to ensure the correctness of the codes. Second, maintaining the developer's involvement fosters a sense of code ownership and ensures they maintain a clear understanding of the code generation process. In software development, the process itself often carries more significance than the final result. This engagement allows developers to gain a deeper understanding of code details, enhancing their ability to address future bugs swiftly. However, it is important to limit developer involvement. We ask users only to validate test results, not the code itself. ConTested can achieve a 33% improvement in performance with only four rounds of interaction. Additionally, we propose an automated variant, ConTested$_{o1}$, which also outperforms the standard SOTA approach.

Usability of ConTested in Complex Scenarios ConTested can correct the tests and ensure their accuracy. The tests serve as a strong signal, providing feedback on the specific situations where the code fails and helping to locate and fix bugs. This is why ConTested can work well. Therefore, when using ConTested, the performance will not become worse at the very least, regardless of the difficulty of the problem. The proposed method addresses a fundamental challenge related to inaccuracies in test cases; however, other challenges remain. One such challenge is that the effectiveness of code generation is influenced by the capability of the base model (LLM) and the complexity of the given problem. If the problem is highly complex, and the base model generates totally poor responses, it indicates that the model fails to comprehend the problem entirely. In such cases, providing corrected test cases may not have great improvement, but the performance will not become worse, at the very least. As demonstrated in Table 1 and Table 2 of the paper, ConTested consistently enhances performance towards LLMs with different abilities (GPT-3.5, GPT-4o, and OpenAI o1), regardless of whether it is applied to a weak or strong model. On the other hand, as long as the base model demonstrates some capacity to solve the problem, ConTested can enhance its performance based on the corrected tests.

Threats to Validity *Threats to Internal Validity* mainly lie in the randomness introduced by LLMs. To address this issue, we use identical code and tests generated by LLMs for different techniques

to maintain consistency. Additionally, for fair comparison, we employ the same ChatGPT API as the state-of-the-art technique MPSC [14]. To ensure accuracy, the first two authors meticulously review the code to prevent bugs. *Threats to external validity* mainly lie in the benchmark used. To prevent data leakage issues with LLMs, we use hand-written datasets that are not part of the LLM training data. These datasets are consistently used across all code generation techniques. Another external validity threat arises from the user study. Each problem requires developers to perform three different tasks, and to minimize bias, we assign different developers to each task within a problem. This introduces the potential threat that the three developers may have varying levels of expertise. To mitigate this, we select users with comparable development experience.

Limitations The first limitation of ConTested is that its performance depends on the quality of the corrected tests. If tests are incorrectly corrected, the model may be misled—a limitation shared by all human-involving techniques. However, correcting tests is generally less error-prone than directly correcting code. Additionally, in real-world development, users must ensure the correctness of tests; otherwise, the quality of the code may be compromised. While formal verification could be used to ensure test correctness, it is often too time-consuming for practical use.

7 Related Work

In this section, we present the related work relevant to ConTested. We cover three types of related work: LLM for code generation, *Consistency*, and other post-hoc techniques.

LLM for Code Generation Code generation techniques automatically produce code snippets based on natural language descriptions, which has high practical value and has been studied for decades. Early methods relied on templates to generate code [17, 42, 43]. With the appearance of neural networks, many learning-based techniques [29, 35, 41] emerged, using natural language requirements as input and generating code as output. Recently, LLMs have shown strong performance in various tasks, including code generation. General-purpose LLMs, such as ChatGPT[27] and Claude[3], are trained on diverse types of textual corpora, achieving high performance across NLP tasks, including code generation. In addition, code-specific LLMs, trained on extensive public code datasets, such as DeepSeek-Coder [13], WizardCoder [21], and StarCoder [19], have been developed. All these techniques utilize a similar network architecture based on the Transformer model [37], with only minor modifications, such as adjustments in layer count and positional embeddings. These techniques commonly adopt a two-stage training process: (1) next token prediction with "fill-in-the-middle" (FIM) and (2) instruction tuning. FIM enables models to predict masked snippets using surrounding code, enhancing their understanding of code context and generation ability. Instruction tuning further improves the model's capability to follow human instructions, essential for generating code based on natural language commands.

Consistency Despite achieving promising performance, LLM outputs remain unreliable, particularly for complex tasks where we cannot guarantee output accuracy. This requires users to verify LLM outputs, producing additional human efforts. To improve output reliability, researchers have proposed leveraging *Consistency* to post-process results. By sampling multiple outputs from LLMs and using majority voting, the approach selects the most consistent response as the final answer. This *Consistency* method is theoretically based on the notion that the tasks may allow multiple valid paths to the correct answer [33]. Rooted in the principle of *diversity*, *Consistency* assumes that when diverse reasoning methods converge on a single answer, it is more likely accurate. For instance, Wang et al. [39] samples multiple "chain-of-thought" paths, choosing the most frequent answer, while Sun et al. [34] generates varying outputs by first reciting different relevant knowledge. These methods judge consistency directly from output agreement. However, code generation is an open-ended problem, so *Consistency* techniques in code generation domain rely on different consistency indicators to assess consistency. CodeT[5] generates both codes and multiple tests, selecting the

code that passes the most tests (inter-consistency) and shares the most functionally equivalent counterparts (intra-consistency). Both types of consistency depend heavily on test quality, yet these tests, generated by LLMs, may contain errors. MPSC[14] considers specification alongside tests, though this approach also faces limitations. As shown in their paper [14], specification achieves only around 50% accuracy, and has small impact on results after removing specification. To address these issues, ConTested enhances the quality of tests and codes iteratively through a co-evolution process. Other consistency types involve cross-model consistency, where different LLMs debate when their answers are inconsistent [40]. ALGO [44] introduces an additional brute-force implementation, using its output as an oracle to verify the correctness of other generated outputs. However, the accuracy of this brute-force algorithm itself is not guaranteed.

Other Post-Process Techniques Besides *Consistency*, there are also other post-process techniques improving the quality of LLMs' outputs. Self-refine [22] prompts the LLM to assess its output, provide feedback, and refine based on that feedback. However, these methods lack explicit guidance for modification, requiring the LLM to self-reflect, which can be challenging. In contrast, we provide LLMs with failed test cases as direct feedback. Reflexion [32] uses environmental feedback, such as error messages during test execution, to refine outputs, though the authors acknowledge that performance relies on test quality. Other techniques [23, 45] introduce a separate verifier or reviewer to score and re-rank outputs, yet these approaches also lack a specific direction for evaluation. In our work, tests offer the most precise guidance for evaluation and refinement.

8 Conclusion

LLMs have shown promising performance in code generation; however, they struggle to produce flawless code in a single attempt. Researchers leverage *Consistency* to enhance code quality. Nevertheless, current methods overlook a crucial aspect of using *Consistency*: the consistency indicators should be of good quality. Without this, the achieved consistency remains unreliable. In this work, we introduce Consistency-Aided Tested Code Generation (ConTested), an approach designed to enhance code generator performance through two key components: (1) lightweight user effort for validating the correctness of selected tests, and (2) a dynamic strategy for ranking, localizing, and correcting multiple tests and codes. Our framework enables a lightweight, interactive process that incorporates user feedback to address identified tests and guide the iterative improvement process. Notably, the iteration rounds average only four with the support of consistency, requiring minimal human effort to achieve a performance improvement of approximately 30%. Each iteration follows a co-evolution process involving codes and tests. This process iteratively refines code and test quality, making consistency voting from codes to tests and vice versa increasingly reliable. More reliable $\mathsf{Con}_{t\to c}$ allows us to select better code, while more reliable $\mathsf{Con}_{c\to t}$ enhances the accuracy in identifying incorrect tests. The co-evolution process terminates when we identify a code that passes all tests. Given the high quality of the tests, this selected code is more reliable. We evaluate ConTested through extensive experiments, demonstrating its effectiveness across multiple LLMs, including GPT-3.5 and GPT-4o.

9 Data Availability

Our package is available at [1], which contains the data and scripts for reproduction.

Acknowledgments

This work was supported by National Natural Science Foundation of China under Grant No. 62372005 and 62232001.

References

[1] 2024. Replication package. https://github.com/DJjjjhao/replication_package.

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

[3] anthropic 2024. *Meet Claude.* anthropic. https://www.anthropic.com/claude

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

[5] 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. https://openreview.net/forum?id=ktrw68Cmu9c

[6] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to accelerate symbolic execution via code transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 6–1.

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

[8] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

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

[11] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2021. Glm: General language model pretraining with autoregressive blank infilling. *arXiv preprint arXiv:2103.10360* (2021).

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[13] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[14] Baizhou Huang, Shuai Lu, Xiaojun Wan, and Nan Duan. 2024. Enhancing Large Language Models in Coding Through Multi-Perspective Self-Consistency. 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*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 1429–1450. https://doi.org/10.18653/V1/2024.ACL-LONG.78

[15] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

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

[17] Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. North American Chapter of the Association for Computational Linguistics (NAACL).

[18] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. Codeeditor: Learning to edit source code with pre-trained models. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–22.

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

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

[21] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[22] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems* 36 (2024).
[23] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.
[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[25] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.
[26] OpenAI 2023. *GPT-3.5 Turbo*. OpenAI. https://platform.openai.com/docs/models/gpt-3-5-turbo
[27] OpenAI 2023. *Introducing GPT-4o and more tools to ChatGPT free users*. OpenAI. https://openai.com/index/gpt-4o-and-more-tools-to-chatgpt-free/
[28] OpenAI 2024. *GIntroducing OpenAI o1-preview*. OpenAI. https://openai.com/index/introducing-openai-o1-preview/
[29] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).
[30] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[31] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454* (2022).
[32] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*. 8634–8652.
[33] Keith E Stanovich and Richard F West. 2000. Advancing the rationality debate. *Behavioral and brain sciences* 23, 5 (2000), 701–717.
[34] Zhiqing Sun, Xuezhi Wang, Yi Tay, Yiming Yang, and Denny Zhou. 2022. Recitation-augmented language models. *arXiv preprint arXiv:2210.01296* (2022).
[35] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 8984–8991.
[36] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (2022).
[37] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
[38] Han Wang, Archiki Prasad, Elias Stengel-Eskin, and Mohit Bansal. 2024. Soft Self-Consistency Improves Language Model Agents. *arXiv preprint arXiv:2402.13212* (2024).
[39] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=1PL1NIMMrw
[40] Kai Xiong, Xiao Ding, Yixin Cao, Ting Liu, and Bing Qin. 2023. Examining Inter-Consistency of Large Language Models Collaboration: An In-depth Analysis via Debate. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 7572–7590.
[41] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).
[42] Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 678–687.
[43] Luke S Zettlemoyer and Michael Collins. 2012. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420* (2012).
[44] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems* 36 (2023), 54769–54784.
[45] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*. PMLR, 41832–41846.

Received 2025-02-27; accepted 2025-03-31