A Pair Programming Framework for Code Generation via Multi-Plan Exploration and Feedback-Driven Refinement

Huan Zhang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
zhanghuan.nju@gmail.com

Wei Cheng
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
wchengcs.nju@gmail.com

Yuhan Wu
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
yhwu.nju@gmail.com

Wei Hu*
State Key Laboratory for Novel Software Technology
National Institute of Healthcare Data Science
Nanjing University
Nanjing, China
whu@nju.edu.cn

ABSTRACT

Large language models (LLMs) have achieved impressive performance on code generation. Although prior studies enhanced LLMs with prompting techniques and code refinement, they still struggle with complex programming problems due to rigid solution plans. In this paper, we draw on pair programming practices to propose PairCoder, a novel LLM-based framework for code generation. PairCoder incorporates two collaborative LLM agents, namely a Navigator agent for high-level planning and a Driver agent for specific implementation. The Navigator is responsible for proposing promising solution plans, selecting the current optimal plan, and directing the next iteration round based on execution feedback. The Driver follows the guidance of Navigator to undertake initial code generation, code testing, and refinement. This interleaved and iterative workflow involves multi-plan exploration and feedback-based refinement, which mimics the collaboration of pair programmers. We evaluate PairCoder with both open-source and closed-source LLMs on various code generation benchmarks. Extensive experimental results demonstrate the superior accuracy of PairCoder, achieving relative pass@1 improvements of 12.00%–162.43% compared to prompting LLMs directly.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**.

KEYWORDS

Code generation, Large language model, Agent, Pair programming

*Corresponding author

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695506

ACM Reference Format:
Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A Pair Programming Framework for Code Generation via Multi-Plan Exploration and Feedback-Driven Refinement. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695506

1 INTRODUCTION

Code generation aims to automatically generate executable source code that conforms to given requirements, typically expressed in natural language. Recent progress in large language models (LLMs) has significantly improved software development productivity by reducing repetitive programming efforts [23, 35]. The success of commercial models like ChatGPT [33] and Claude [1], along with powerful open-source models like Code Llama [40] and DeepSeek Coder [14], has attracted substantial interest from both academia and industry. These advancements demonstrate the remarkable capabilities of LLMs in code generation and have great potential to influence the field of intelligent software engineering [9, 13, 48].

As requirements become more complex, it becomes challenging for LLMs (and even humans) to directly generate code that meets the given requirements [10]. One key focus of existing work is prompting techniques, which guide LLMs to produce intermediate reasoning steps for problem descriptions. This line of work [22, 25, 46] focuses on designing different types of prompts to stimulate the reasoning abilities of LLMs, enabling them to generate intermediate steps before producing the final code. Another important aspect is that generating the correct code is rarely a one-time effort [7]. Several studies [5, 27, 31] employ sampling-based approaches to filter or rank the numerous responses generated by LLMs, relying on a substantial number of samples. Other works [7, 32, 45] attempt to refine the generated code using feedback from LLMs themselves or external sources. They introduce a debugging process to make the generated program behave as expected. Furthermore, a few works [10, 16, 37] explore the use of collaborative LLM agents to simulate human software development processes.

Although existing approaches have improved code generation, they still have significant limitations. First, most approaches focus

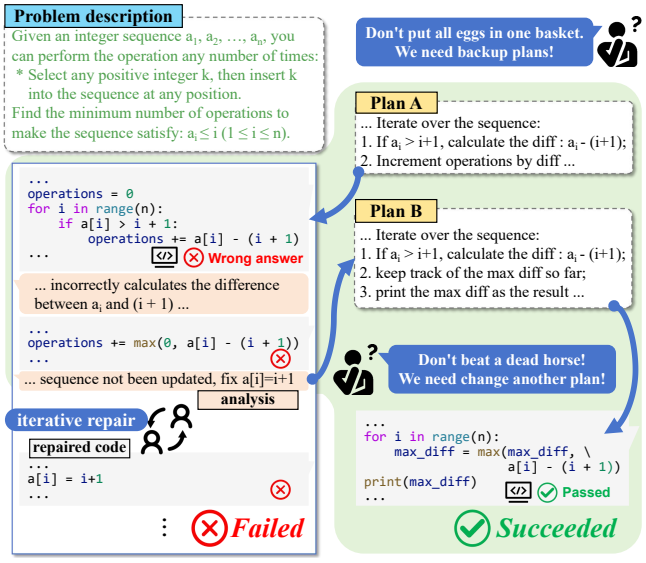

Figure 1: A competitive programming problem excerpted from the CodeContest benchmark (#test87) [27]. Both the plans and codes are generated by GPT-3.5-Turbo.

too narrowly on generating a single solution plan, lacking high-level planning and problem decomposition capabilities. Due to the inherent complexity of real-world problems, a single solving path often fails to produce correct code directly. Second, if the initially generated code follows a flawed plan, the feedback-based repair process is likely to be misguided by the initial incorrect direction, failing to address fundamental issues. This deviates from how human developers tackle programming problems in practice. Human developers typically start with high-level problem analysis and propose multiple potential solution plans. Then, they evaluate these plans based on continuous implementation and testing feedback. If the current plan is deemed infeasible or reaches an impasse, they will decisively abandon it and explore alternatives. This iterative cycle of "multi-plan exploration and practical feedback" ensures that human developers can efficiently solve complex problems without being constrained by a single flawed plan.

To overcome these limitations, we propose PairCoder, a novel code generation framework inspired by pair programming practices [47] in software engineering. In pair programming, two developers play the roles of "navigator" and "driver": the former is responsible for high-level planning and direction, while the latter focuses on implementing the current task. We adapt pair programming practices to LLM-based code generation by introducing two intelligent agents: Navigator and Driver. The Navigator's role includes starting with high-level problem analysis (referred to as reflection), generating multiple potential solution plans, selecting the best current plan, and directing the subsequent process based on the Driver's execution feedback. The Driver generates or repairs the code according to the Navigator's guidance and provides execution feedback for subsequent adjustment.

This Navigator-Driver framework mimics the iterative and adaptive strategies employed by human developers. By starting with reflection and generating multiple plans, it avoids the constraints of a single flawed plan. Continuous feedback interaction between the Navigator and the Driver allows for dynamic plan adjustment, enabling the abandonment of infeasible plans and exploring new directions when necessary. This iterative cycle significantly improves the robustness and quality of code generation for real-world tasks. We conduct extensive experiments to evaluate our PairCoder on five diverse code generation benchmarks using both open-source (DeepSeek-Coder [14]) and closed-source (GPT-3.5-Turbo [34]) LLMs. The results show that on the pass@1 metric, PairCoder achieves superior accuracy over competitive baselines across all benchmarks for both LLMs.

In summary, the key contributions in this paper are outlined as follows:

- We are the first to adapt pair programming practices into LLM-based code generation and propose a new framework called PairCoder that comprises two collaborative agents: a Navigator for high-level planning and a Driver for specific implementation. (Sect. 3.1)
- PairCoder integrates two key mechanisms: (i) multi-plan exploration achieved by the Navigator through adjusting diverse solution plans, and (ii) feedback-driven refinement based on execution feedback from the Driver and historical memory. (Sects. 3.2 and 3.3)
- PairCoder significantly outperforms all competitive baselines on five benchmarks. It gains relative improvements of 16.97%–162.43% on GPT-3.5-Turbo and 12.00%–128.76% on DeepSeek-Coder-Instruct 33B compared to prompting LLMs directly. (Sect. 4) The source code is publicly available on GitHub.[1]

2 MOTIVATING EXAMPLE

Fig. 1 shows a programming problem from the CodeContest benchmark [27]: given an integer sequence $a_1, a_2, \ldots, a_n$, find the minimum number of insertions to ensure $\forall 1 \le i \le n, a_i \le i$.

The prompting approaches [22, 25, 46] generate intermediate reasoning steps based on the problem description to guide code generation, as excerpted in Plan A. Considering an input sequence "1, 3, 4" in public test cases, it is evident that a single operation is sufficient to satisfy the requirement, i.e., inserting '2' between '1' and '3'. Guided by Plan A, the first code snippet on the left side of Fig. 1 incorrectly assumes that two insertions are required. This feedback from code testing can help LLMs repair the generated code [7, 32, 45]. However, iterative repairs do not yield the right answers as they stubbornly follow the flawed Plan A, which incorrectly accumulates `operations`. For example, LLMs would mistake the bug as an incorrect calculation of the difference between a_i and $(i + 1)$, or that the sequence has not been updated. We believe this dilemma arises from an inherent pitfall of the single-path approach: once an incorrect blueprint is initially established, LLMs struggle to identify the root cause of the error and thus mislead subsequent repairs.

In contrast, human programmers do not put all eggs in one basket. If the current plan is deemed ineffective, they will explore alternative plans. Plan B correctly solves the problem by tracking

[1] https://github.com/nju-websoft/PairCoder

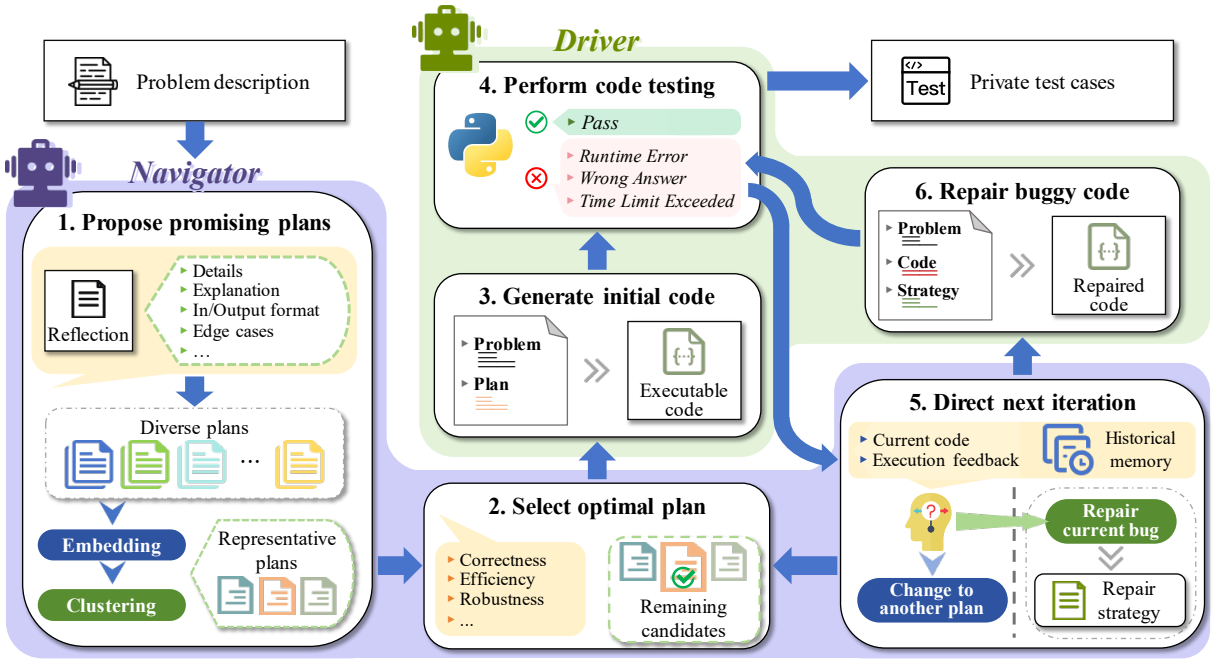

Figure 2: Overview of our PairCoder, in which a Navigator agent and a Driver agent collaborate on code generation.

the maximum difference between a_i and $(i+1)$. With the proper solution plan, LLMs generate the correct code instead of getting stuck in a refinement loop under Plan A.

In practice, solving complex problems often requires exploring multiple solution plans, and timely adjustment can mitigate wasted effort on flawed plans. The role of multi-plan exploration and flexible adjustment is analogous to that of the "navigator-driver" in pair programming, serving as the key inspiration for our work.

3 FRAMEWORK

3.1 Overview

We first formulate the realistic problem of code generation as generating a program C from a natural language description Q and a set of public (visible) test cases $\mathcal{T}_v = \{(I_i, O_i)\}_{i=1}^{m_v}$, where O_i denotes the desired output for the input I_i. Based on the execution feedback from $\mathcal{T}_v$, LLMs can iteratively refine the generated program C, having a maximum number of iterations r to control cost and efficiency. Finally, C is considered correct if its behavior is consistent with the test oracle [17], which is usually represented by a set of private (hidden) test cases $\mathcal{T}_h = \{(I_i, O_i)\}_{i=1}^{m_h}$, i.e., satisfying that $\forall (I_i, O_i) \in \mathcal{T}_h, C(I_i) = O_i$. Note that the accessible $\mathcal{T}_v$ is not a complete test, while $\mathcal{T}_h$ is invisible during the code generation and refinement stages.

Fig. 2 illustrates the workflow of PairCoder. Both the Navigator and Driver agents are powered by an LLM with general-purpose capabilities, such as GPT-3.5-Turbo [33]. The Navigator guides the Driver by generating solution plans and repair strategies. Therefore, the Driver focuses all its attention on specific code tasks, including code generation, code testing, and refinement. The workflow of the two agents is interleaved and iterates until the generated program C passes all public test cases $\mathcal{T}_v$ or the maximum number of iterations r is reached. To clearly describe the details of PairCoder, we present Algorithm 1 throughout this section.

3.2 Navigator Agent

The Navigator agent serves as the main controller in deeply understanding the problem and providing strategic direction. Its role is to propose multiple promising plans (Step 1 in Fig. 2), select the currently best solution plan (Step 2), and direct the next iteration based on execution feedback and historical memory (Step 5).

Propose promising plans. The Navigator first reflects on the given natural language description Q (Line 1 in Algorithm 1). The prompt for LLMs is shown in Fig. 3. It stimulates LLMs to explicitly analyze the details of the problem, consider possible valid inputs and edge cases, and explain public test cases. This reflection process enables the Navigator to gain a comprehensive understanding of the core logic, constraints, and requirements for effective problem solving.

With the comprehensive reflection on the problem, the Navigator further comes up with specific solution plans (Line 2). As shown in Fig. 4, we include brief examples in prompts to guide the proposal and emphasize the functional correctness of the proposed plans. Each plan outlines a high-level solution and key implementation steps in concise natural language. To obtain diverse solution plans, we set a non-zero temperature for multiple nucleus sampling [15] and ask LLMs to generate multiple plans in each batch [8], which improves sampling efficiency while reducing duplicate plans. After brainstorming through multiple sampling, we select k representative plans as candidates. Specifically, we first divide

Algorithm 1: PairCoder

Input: problem description Q, public test cases $\mathcal{T}_v$
[*Agents*]: Navigator model $\mathcal{M}_N$, Driver model $\mathcal{M}_D$
[*Memories*]: code history $\mathcal{H}_c$, feedback history $\mathcal{H}_f$
[*Hyperparameters*]: the maximum number of iterations r, the number of sampled plans n, the number of clusters k
Output: generated code C

1 $reflection \leftarrow \mathcal{M}_N(\cdot \mid \texttt{ReflectPrompt}, Q)$;
2 $\{P_i\}_{i=1}^{n} \leftarrow \mathcal{M}_N(\cdot \mid \texttt{PlanPrompt}, Q, reflection)$;
3 $S \leftarrow \textsc{ClusterPlans}(\{P_i\}_{i=1}^{n}, k)$; // $S \subseteq \{P_i\}_{i=1}^{n} \wedge \|S\| = k$
4 $C \leftarrow NULL$; $F \leftarrow NULL$;
5 **for** $j := 1$ **to** r **do**
6 **if** $(j = 1) \vee (C \in \mathcal{H}_c) \vee (F \in \mathcal{H}_f)$ **then**
 // select a new solution plan
7 $plan \leftarrow \mathcal{M}_N(\cdot \mid \texttt{SelectPrompt}, Q, reflection, S)$;
8 $S \leftarrow S \setminus \{plan\}$;
9 $\mathcal{H}_c \leftarrow \emptyset$; $\mathcal{H}_f \leftarrow \emptyset$;
10 $C \leftarrow \mathcal{M}_D(\cdot \mid \texttt{CodePrompt}, Q, plan)$;
11 **else** // stick to the current plan
12 $\mathcal{H}_c \leftarrow \mathcal{H}_c \cup \{C\}$; $\mathcal{H}_f \leftarrow \mathcal{H}_f \cup \{F\}$;
13 $strategy \leftarrow \mathcal{M}_N(\cdot \mid \texttt{AnalyzePrompt}, Q, C, F)$;
14 $C \leftarrow \mathcal{M}_D(\cdot \mid \texttt{RepairPrompt}, Q, C, strategy)$;
15 $F \leftarrow \textsc{ExecuteTests}(C, \mathcal{T}_v)$;
16 **if** $F = Pass$ **then break** ;
17 **return** C;

all n samples into k clusters using a text embedding model and the classical k-means++ algorithm [2], and then select the plan closest to the cluster centroid from each cluster (Line 3). Intuitively, the Navigator groups similar plans together and selects representative ones, ensuring a diverse set of high-level solution plans. These can involve techniques like brute force search, greedy algorithm, or other problem-solving strategies.

Select optimal plan. In each iteration, there is a determined solution plan to guide code generation or refinement. Whenever a plan needs to be initially selected or discarded, the Navigator selects the optimal plan from the remaining candidates (Line 7). Following the previous studies [21, 52], we also make choices through LLMs, where the prompt template is shown in Fig. 5. Considering the problem description and reflection together, the Navigator leverages the reasoning capabilities of LLMs to consolidate multiple key factors in code quality, including correctness, efficiency, and robustness. Note that functional correctness takes precedence over efficiency. We prefer to try an inefficient but intuitive solution plan first, such as a direct brute force method, which is in line with the common practice of human programmers to prioritize problem-solving before optimization. Once a plan is selected, it is removed from the candidates, preventing it from being selected again in subsequent iterations (Line 8).

Direct next iteration. Once the generated code C in the last iteration does not pass all the public test cases $\mathcal{T}_v$, it is the Navigator's turn to direct the next iteration. Instead of stubbornly persisting in a single solving path to repair the incorrect code [7, 42, 45, 54], the Navigator can timely adjust the solution plan to seek a turnaround. We observe that code refinement tends to get stuck in a dead-end

ReflectPrompt

```
You are given a coding problem: <<Problem Description>>
Given the coding problem, you have two tasks using natural
language:
1. reflect on the problem:
- For simple problems, briefly provide a concise explanation
and note all possible valid inputs and edge cases. Avoid
overthinking.
- For complex problems, provide a comprehensive analysis
covering all aspects, details, nuances, and how to properly
handle all possible valid inputs and edge cases per problem
description.
2. For provided public test case(s) in description, briefly
explain how the specified input yields the expected output
based on the problem description.
```

```
<<Reflection>>
This problem requires finding the minimum operations to
modify a sequence so each element is less than or equal to
its index ...
Simple inputs where the sequence meets the condition,
complex inputs needing multiple operations, and edge cases
like an empty sequence ...
For input [1, 3, 4], 1 operation is needed. Insert 2 to get
[1, 2, 3, 4].
```

Figure 3: The prompt template used by the Navigator to reflect on the given problem description and an output example of the reflection.

PlanPrompt

```
You are given a coding problem: <<Problem Description>>
Self-reflection on the problem: <<Relfection>>
Provide up to 3 possible solution plans to the problem.
Each solution plan should:
1.Have a descriptive name.
2.Outline the solution approach:
- For simple problems (e.g., reversing a list), provide a
concise solution (e.g., we can directly use list1[::-1] to
do it.).
- For complex problems (e.g., finding the shortest path in a
graph), provide a problem-solving plan with high-level steps
(e.g., we can use BFS to solve it. First, Initialize a queue
with... Second, ...).
  If necessary, select an appropriate algorithm through
problem analysis such as brute force, simulation, greedy,
hash map, two pointers, DFS/BFS, stack/queue, DP, etc.
3.Ensure functional correctness by addressing all possible
valid inputs and edge cases per problem description.
```

```
<<Solution Plans>>
(1) Greedy Approach
 - Initialize `operations` to 0.
 - Iterate through the sequence.
 - For each a_i at position i, if a_i > i, calculate `diff` as
a_i - i and track the max`diff`.
 - Output the final value of `operations`.
(2) Brute Force Approach
 - Set `min_operations` to infinity.
 - Generate all possible sequences.
 - For each sequence, check conditions and count operations.
 - Update `min_operations` if fewer operations are needed.
(3) ...
```

Figure 4: The prompt template for sampling multiple plans and an output example of solution plans.

SelectPrompt

```
You are given a coding problem: <<Problem Description>>
Self-reflection on the problem: <<Relfection>>
Here is a list of <<k>> possible solutions to the problem:
<<k candidates>>
Choose the most robust and correct solution and provide a
brief explanation for your choice. The selected solution
should:
- Prioritize functional correctness over efficiency. If
there is a simulation method or a direct brute force method
available, prefer it.
- Fully solve the problem and correctly handle all possible
valid inputs and edge cases as per the problem description.
- Consider more efficient methods only if they do not
compromise correctness.
```

Figure 5: The prompt template for selecting the optimal solution plan from the candidates.

AnalyzePrompt

```
You are given a coding problem: <<Problem Description>>
A Python code solution for the problem: <<Buggy Code>>
However, the code solution failed to produce the expected
output: <<Execution feedback>>
Identify the specific part(s) of the code that contain
logical errors or incorrect functional function(s) in the
code. Briefly explain the root causes of the identified code
section(s) that prevent it from producing the expected
output for the specified input.
Provide a step-by-step approach to fix the issues, ensuring
the corrected code can handle all valid inputs correctly.
Keep the fix steps concise for minor issues, but provide
more detailed steps if major revisions are required.
```

Figure 6: The prompt template used to propose a repair strategy. This is customized for *Wrong Answer* and is slightly different from the other types of execution feedback.

loop if the generated code or execution feedback has already occurred in the past. Therefore, we introduce a long-term memory module to systematically store and maintain the coding and execution history under the current solution plan (Line 12). It consists of the generated programs $\mathcal{H}_c = \{C^i\}_{i=1}^r$ and the execution feedback $\mathcal{H}_f = \{F^i\}_{i=1}^r$, where F^i is the execution feedback of the generated code C^i in the i-th round of iteration, comprising specific error type and execution details. We apply a simple but effective *heuristic strategy* to determine whether to change the solution plan. Given the buggy code and its execution feedback, the current solution plan will be considered unpromising if any of them has already occurred in the historical memory $\mathcal{H}_c$ and $\mathcal{H}_f$ (Line 6), leading to a re-selection of the optimal plan in Step 2. Whenever a new solution plan is selected, the historical memory module is cleared to start fresh (Line 9).

Another potential iteration direction is to repair the buggy code, which pursues gradual progress without abandoning a promising solution plan. Based on the execution feedback, the Navigator leverages the reasoning ability of LLMs to propose a directive repair strategy (Line 13). As shown in Fig. 6, the prompt for LLMs comprises the problem description, the buggy code, and specific execution feedback. For different types of execution feedback, we slightly adjust the prompts to remind LLMs to focus on specific aspects. Specifically, *Runtime Error* can suggest repairs that involve syntax error or exception handling, e.g., array index out of bounds. *Wrong Answer* indicates logic errors in the code, which may inspire repair strategies like adjusting condition handling and data flow. For the inefficient solution plans flagged by *Time Limit Exceeded*, the Navigator may recommend optimizations to improve computational performance. The customized repair strategy will guide the code refinement in Step 6 of the next iteration.

3.3 Driver Agent

In contrast to the high-level planning of the Navigator, the Driver agent focuses all its attention on specific code tasks, including generating initial code guided by a new plan (Step 3), testing code on public test cases (Step 4), and repairing the buggy code (Step 6).

Generate initial code. Once a new solution plan is selected, the Driver first generates an initial code implementation guided by the plan (Line 10). To align with the high-level planning of the Navigator, the Driver concatenates the problem description and current solution plan together, making the prompt shown in Fig. 7. Existing LLMs [14, 33] already have impressive abilities to comprehend the context in prompts and convert solution plans expressed in natural language into corresponding executable programs [22, 26, 46]. We expect, but do not mandate, that the generated code is an exact match to the solution plan. Incorrect implementation is tolerated at this step, which would be recognized and refined in subsequent iterations.

Perform code testing. Generating correct code is rarely a one-time effort, and recent studies indicate that LLMs struggle to self-correct their responses without external feedback [19, 44]. Therefore, we follow previous works [7, 32] to introduce an executor that evaluates the generated program C on the public test cases $\mathcal{T}_v$ (Line 15). We categorize the execution feedback into four types:

- *Pass.* It indicates that C successfully passes all public test cases, i.e., satisfying that $\forall (I_i, O_i) \in \mathcal{T}_v, C(I_i) = O_i$.
- *Runtime Error.* It indicates that the execution is terminated prematurely due to unhandled exceptions or errors.
- *Wrong Answer.* It indicates that C gives unexpected outputs in some cases, i.e., satisfying that $\exists (I_i, O_i) \in \mathcal{T}_v, C(I_i) \neq O_i$. This type takes precedence over *Time Limit Exceeded*.
- *Time Limit Exceeded.* It indicates that C fails to produce outputs within the specified time limit.

If the execution feedback is *Pass*, we will terminate the iterative process and consider C as the final output (Line 16); Otherwise, the Driver will deliver the current program and execution feedback to the Navigator, which are used to direct the next iteration in Step 5. Code testing marks the end of an iteration round, so the iterative process will also be terminated after performing the r-th test, outputting the last generated code (Line 17).

Repair buggy code. If the Navigator assumes that the current solution plan remains promising, the Driver will attempt to repair the buggy code based on the given repair strategy (Line 14). As shown in Fig. 8, the prompt for LLMs comprises the problem description, the buggy code, the execution feedback, and the repair strategy. The Driver aims to address the issues identified in the repair strategy, such as logic errors and performance bottlenecks.

CodePrompt

```
You are given a coding problem: <<Problem Description>>
Please generate a Python code to fully solve the problem
using the following solution: <<Solution Plan>>
```

Figure 7: The prompt template used by the DRIVER to generate initial code guided by a new solution plan.

RepairPrompt

```
You are given a coding problem: <<Problem Description>>
A Python code solution for the problem: <<Buggy Code>>
However, when running the following input example, the code
solution above failed to produce the expected output:
<<Execution Feedback>>
Please fix the code using the following approach:
<<Repair Strategy>>
```

Figure 8: The prompt template for repairing buggy code.

Similar to the code generation in Step 3, we do not claim that the repair is a complete success, and the generated code can be further refined in subsequent iterations.

Complexity analysis. The time complexity of Algorithm 1 is determined by the number of iterations r and the cost of operations within each iteration. Let c denote the constant factor representing the cost of operations, such as model inference and code testing within each iteration. Then, the overall time complexity of PAIRCODER is $O(r \times c)$. Similarly, the space complexity of PAIRCODER is $O(r)$, since the NAVIGATOR needs to store the historical memory $\mathcal{H}_c$ and $\mathcal{H}_f$, which grow linearly with the iteration count. While the multi-plan exploration and iterative refinement introduce additional computational overhead, the superior accuracy of PAIRCODER in code generation justifies the trade-off in Sect. 4.

4 EXPERIMENTS AND RESULTS

We evaluate PAIRCODER by defining the following research questions (RQs) and outlining how we propose to answer them:

- **RQ1. How does the accuracy of PAIRCODER in code generation compare to other approaches?** We aim to evaluate the effectiveness of our PAIRCODER framework in code generation compared to existing approaches. We conduct comprehensive experiments across diverse benchmarks and foundation models.
- **RQ2. How do critical hyperparameters impact the accuracy of PAIRCODER?** We thoroughly investigate the effect of iteration count on PAIRCODER compared to other iterative refinement-based approaches, as well as the impact of cluster number on PAIRCODER.
- **RQ3. What are the individual contributions of the major components in PAIRCODER?** We aim to analyze the effectiveness of two major components in PAIRCODER: multi-plan exploration and feedback-driven refinement facilitated by NAVIGATOR-DRIVER collaboration. By disabling each component in ablation studies, we isolate their effects and validate their contributions to the overall accuracy.
- **RQ4. What are the findings of cost and error analyses for PAIRCODER?** The cost analysis focuses on quantifying the API calls and token consumption, providing insights into the computational resources required to deploy PAIRCODER in real-world scenarios. We also analyze the causes of errors in failed test cases, indicating potential future improvements in code generation.

Table 1: Benchmark statistics, including the number of programming problems Q, the average number of public test cases m_v, and the average number of private test cases m_h.

Features	HumanEval		MBPP		CodeContest	
	Orig	Plus	Orig	Plus	Valid	Test
# Q	164	164	500	399	117	165
Avg. m_v	2.8	2.8	1.0	3.0	1.5	1.7
Avg. m_h	9.6	764.1	3.1	108.5	202.9	202.1

4.1 Experiment Settings

Benchmarks. Following the prior works [7, 45], we conduct comprehensive experiments on five widely used benchmarks of code generation: HumanEval [6], HumanEval+ [28], MBPP [4], MBPP+ [28], and CodeContest [27]. The statistics of these benchmarks are shown in Table 1. HumanEval, MBPP, and their extended versions (Plus) aim at simple function-level code generation, while CodeContest consists of competition-level programming problems. Both the validation and test sets of CodeContest are considered.

Furthermore, we put effort into providing public test cases $\mathcal{T}_v$ for execution feedback. For the benchmarks [6, 27, 28] where public test cases are provided in problem descriptions, we extract $\mathcal{T}_v$ using hand-written rules. For the benchmarks lacking public test cases in the descriptions, we follow [7, 31, 54] by treating the first private case as $\mathcal{T}_v$ for MBPP, while for MBPP+, we use the original three private cases before extension as $\mathcal{T}_v$.

Metrics. In line with previous works [7, 10, 22, 30], we use the greedy pass@1 [6, 49] to assess the functional correctness of the generated program. A program is regarded correct only if it passes all private test cases T_h. Compared to pass@K with multiple nucleus sampling, the greedy pass@1 represents a more realistic scenario, where developers are not required to review the correct one from multiple solutions.

Comparative methods. We compare PAIRCODER with two main categories of existing approaches for code generation. We briefly describe them as follows.

Prompting techniques. This category focuses on prompts to steer LLMs towards generating more accurate code solutions for requirements. Notable approaches include:

- **Direct prompting** [6] takes the original requirements directly as inputs to prompt LLMs for code generation.
- **Chain-of-Thought (CoT) prompting** [46] elicits LLMs to generate a chain of intermediate natural language reasoning steps before producing the final code. We use the classical CoT instruction "Let's think step by step." to guide LLMs in zero-shot [24].
- **SCoT prompting** [25] asks LLMs using three basic program structures (i.e., sequence, branch, and loop structures) to

build intermediate reasoning steps. Then, LLMs generate the final code based on the structured CoT.
- **Self-planning** [22] consists of planning and implementation phases. In the planning phase, LLMs plan out the solution steps from the intent with few-shot demonstrations. In the implementation phase, LLMs generate code step by step, guided by the planned solution steps.

Refinement-based approaches. Approaches in this category refine the generated code based on feedback, either from the LLM itself or external sources, such as compilers and interpreters.

- **Self-collaboration** [10] enables multiple LLM agents to act as distinct roles (i.e., analyst, coder, and tester) within a virtual team. These roles interact and collaborate in a specified manner to address code generation tasks.
- **Self-repair** [32] employs a feedback model to generate textual explanations for errors encountered during unit test execution. The code model then uses these explanations to repair the initial code.
- **Self-debugging** [7] teaches LLMs to perform iterative *rubber duck debugging* by explaining the generated code line-by-line with few-shot demonstrations.
- **INTERVENOR** [45] prompts LLMs to play two distinct roles. The Code Teacher iteratively crafts the interactive chain of repair based on compiler feedback, which guides the Code Learner to generate or repair code.
- **Reflexion** [42] uses CoT prompting to generate its own test suites. It then iteratively generates code and verbal self-reflections based on this self-generated test feedback, which guide subsequent implementations.
- **LDB** [54] segments programs into basic blocks and tracks intermediate variable values during runtime execution for iterative program repair.
- **MetaGPT** [16] is a multi-agent system that simulates a complete software company by defining five roles and leveraging human-like standard operating procedures.

Implementation details. We apply both open-source and closed-source LLMs to PairCoder, including DeepSeek-Coder-Instruct with 33B parameters [14] and GPT-3.5-Turbo (gpt-3.5-turbo-0613) [33], where the maximum context window is 16K tokens for all approaches. Moreover, the Navigator employs text-embedding-3-large[2] to vectorize the $n = 15$ sampled solution plans and divides them into $k = 3$ clusters. Programs taking longer than 3 seconds to execute on any test case are marked as *Time Limit Exceeded*. We adopt a temperature of 0.8 to sample diverse solution plans and use greedy decoding by setting the temperature to 0 in other steps. For the comparative methods, we reproduce them according to the source code or provided prompts; otherwise, we quote the results directly from their papers [10, 16, 54]. Following the prior works [7, 22, 30, 54], we use greedy decoding in the reproduced methods unless otherwise specified. The maximum number of iterations r is set to 10 for PairCoder and other iterative refinement-based approaches, i.e., up to 10 times the code is generated or refined.

To ensure a fair comparison, all approaches are given identical problem descriptions and public test cases for code generation. The generated code is then executed and evaluated in a consistent Python 3.9 environment. This guarantees that the code generated by different approaches will receive consistent external feedback, thereby enabling unbiased and rigorous comparisons.

[2]https://platform.openai.com/docs/guides/embeddings

4.2 RQ1: Accuracy Comparison

The accuracy comparison results are presented in Table 2. Our PairCoder achieves the best pass@1 scores across all benchmarks and foundation models. In comparison to prompting LLMs directly, PairCoder shows significant relative improvement of 12.00% to 162.43%. The prompting techniques would accumulate errors in intermediate thoughts and single code generation, causing relatively weak accuracy on code generation. CoT and SCoT prompting are even worse than direct generation in some settings, which is consistent with the findings of prior works [20, 26]. The poor performance of Reflexion is likely due to the model generating incorrect test cases, which leads to self-reflections based on false negative evaluations of the code [42]. In contrast, other refinement-based approaches using provided public test cases achieve overall accuracy gains, since the reliable test feedback can guide the refinement in more promising directions. However, they are confined to a single solving path, lacking the flexibility to explore alternative solution plans when stuck. To overcome these limitations, PairCoder combines the advantages of multi-plan exploration and feedback-driven refinement.

Accuracy discrepancies across benchmarks are worth examining. Most approaches perform well on the relatively simple HumanEval and MBPP benchmarks. However, all approaches exhibit a substantial accuracy decrease on the challenging CodeContest benchmark. This reflects that current code generation techniques still have room for improvement in tackling complex programming problems. Note that the direct prompting with DeepSeek-Coder even outperforms that with GPT-3.5-Turbo on HumanEval, HumanEval+, and CodeContest-test. We speculate this may be due to data leakage issues, which will be further analyzed in Sect. 4.6.

We further evaluate PairCoder with one of the most advanced LLMs, GPT-4 (gpt-4-0613) [34], and cite the results of several powerful approaches [10, 16, 42] from their original papers. The comparison results are shown in Table 3. PairCoder still significantly improves accuracy over direct prompting and outperforms existing multi-agent approaches.

> **Answer to RQ1:** PairCoder outperforms all baselines across five benchmarks with two advanced LLMs. Compared with direct generation, it gains remarkable relative improvements ranging from 16.97% to 162.43% on GPT-3.5-Turbo, and from 12.00% to 128.76% on DeepSeek-Coder. The largest improvement is achieved on the most challenging CodeContest benchmark.

4.3 RQ2: Hyperparameter Impact

We investigate the impact of two critical hyperparameters: the maximum number of iterations r and the number of clusters k.

Table 2: Comparison of pass@1 with two LLMs on code generation benchmarks. "†" denotes the value is directly cited from the respective original work, and "-" denotes the empty result due to reproducibility issues.

Approaches	GPT-3.5-Turbo						DeepSeek-Coder					
	HumanEval		MBPP		CodeContest		HumanEval		MBPP		CodeContest	
	Orig	Plus	Orig	Plus	Valid	Test	Orig	Plus	Orig	Plus	Valid	Test
Direct prompting	67.68	60.98	66.80	66.42	6.84	6.06	76.22	67.78	66.40	64.41	5.98	6.67
CoT prompting	68.90	62.80	69.00	67.17	5.13	5.45	77.27	68.90	67.60	67.67	5.45	6.67
SCoT prompting	68.29	61.59	62.60	61.40	5.98	4.24	73.17	65.85	61.80	60.15	4.27	6.06
Self-planning	72.56	64.63	69.60	67.67	7.69	6.06	74.39	68.90	65.80	68.17	6.84	9.09
Self-collaboration†	74.40	-	68.20	-	-	-	-	-	-	-	-	-
Self-repair	73.17	65.24	70.60	69.42	7.69	6.67	77.44	70.12	70.00	70.43	5.98	7.27
Self-debugging	78.05	72.56	72.80	70.43	9.40	11.52	79.27	73.78	72.20	71.12	8.55	13.33
INTERVENOR	77.44	69.51	73.40	71.93	8.55	9.09	79.88	72.56	72.60	72.43	7.69	10.30
Reflexion	69.57	-	67.76	-	-	-	81.99	-	74.31	-	-	-
LDB†	82.90	-	76.00	-	-	-	-	-	-	-	-	-
PairCoder (ours)	**87.80**	**77.44**	**80.60**	**77.69**	**17.95**	**15.15**	**85.37**	**76.22**	**78.80**	**75.69**	**13.68**	**14.55**
Relative improvement ↑	29.73%	26.99%	20.66%	16.97%	162.43%	150.00%	12.00%	12.45%	18.67%	17.51%	128.76%	118.14%

Table 3: Additional comparison of pass@1 with GPT-4 on the original HumanEval and the sanitized MBPP [4].

Approaches	HumanEval	MBPP
Direct prompting	84.76	82.71
MetaGPT†	85.9	87.7
Reflexion†	91.0	77.1
Self-collaboration†	90.2	78.9
PairCoder	**93.90**	**91.23**

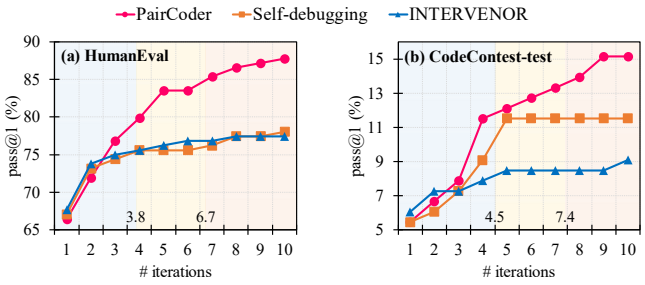

Figure 9: Accuracy changes with the maximum number of iterations, which evaluates PairCoder, Self-debugging, and INTERVENOR using GPT-3.5-Turbo on (a) HumanEval and (b) CodeContest-test benchmarks. The colored demarcations denote the average iteration counts at which PairCoder transitions to its next plan.

Iteration count impact. Given $1 \leq r \leq 10$, we compare PairCoder with two iterative refinement-based approaches, INTERVENOR and Self-debugging. We report the experimental results on HumanEval and the test set of CodeContest with GPT-3.5-Turbo.

Fig. 9 depicts the line plots that show the variation in the pass@1 metric with increasing iteration count for different approaches. As the iteration count increases, all approaches exhibit accuracy improvements, but the extent of these improvements varies. On HumanEval, while Self-debugging and INTERVENOR show modest gains of around 10 percentage points in pass@1 scores after 10 iterations, PairCoder shows a substantially larger improvement of over 21 points. A similar pattern emerges on CodeContest-test, where PairCoder's pass@1 score raises by nearly 10 points, outpacing the limited improvements of the two baselines.

Notably, Self-debugging and INTERVENOR appear to reach an accuracy plateau after a certain number of iterations, e.g., $r \geq 5$ on CodeContest-test. This observation aligns with prior findings [7, 54]. In contrast, PairCoder consistently maintains an upward trajectory across all iteration counts on both benchmarks, suggesting its ability to leverage more iterations for continuous accuracy improvement. Furthermore, we examine when PairCoder typically switches to a new solution plan during iterative refinement. Fig. 9 also labels the average iteration counts at which PairCoder transitions to its next plan, denoted by the colored demarcations, e.g., 3.8 and 6.7 on HumanEval. They also reflect the average count of code repairs under each plan. We observe that the demarcations roughly align with the iteration counts where Self-debugging and INTERVENOR appear to plateau. This correlation suggests that multi-plan exploration based on the long-term memory module enables PairCoder to effectively identify and abandon unpromising plans. Within the same maximum number of iterations, PairCoder can timely explore multiple candidate plans based on execution feedback, achieving superior accuracy by expanding the search space away from local optima.

Cluster number impact. To investigate the impact of the cluster number k on PairCoder, we vary k from 1 to 5. We also report the results on HumanEval and CodeContest-test with GPT-3.5-Turbo.

As shown in Fig. 10, the line plots reflect the changes in pass@1, and the bar charts are for the proportion of problems where all k plans are attempted. As the cluster number k increases, the pass@1 score of PairCoder improves but reaches a bottleneck after $k = 3$.

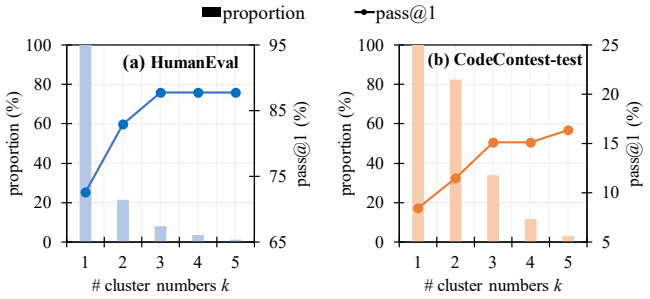

Figure 10: Accuracy changes with the number of clusters, which evaluates PairCoder by pass@1 score and the proportion of problems where all k plans are attempted.

Intuitively, a larger k can bring more candidate solution plans for iterative refinement, leading to a positive correlation with the accuracy of PairCoder. However, only 4.88% and 14.55% of the problems in HumanEval and CodeContest-test, respectively, attempt more than three plans. This is because the iterative process would stop early due to passing public test cases or reaching the maximum number of iterations, resulting in a plateau in accuracy. It suggests that a moderate cluster number $k = 3$ appears to be optimal for PairCoder with the maximum number of iterations $r = 10$. We think that a larger iteration count and cluster number may bring more improvement for PairCoder, but it is a trade-off between effectiveness and cost.

The proportions of attempting all k plans differ significantly between the two benchmarks. For the simpler HumanEval, only 21.34% of problems attempt the second plan with $k = 2$, while this value is 82.42% for CodeContest-test. CodeContest consists of competition-level programming problems, where an initial solution plan is likely to fail to solve the problem. This observation on complex problems supports our design for multi-plan exploration.

> **Answer to RQ2:** The iteration count and cluster number significantly affect the accuracy of PairCoder. The increase in iteration count brings a substantial and consistent improvement to PairCoder versus the baselines. With a maximum of 10 iteration rounds, the optimal cluster number is 3, which is a trade-off for cost efficiency.

4.4 RQ3: Ablation Study

To analyze the individual effectiveness of multi-plan exploration and feedback-driven refinement in PairCoder, we conduct ablation studies in Table 4. "w/o MP" disables the capability of multi-plan exploration, making the Navigator always choose to repair the current code rather than adjust the solution plan in Step 5. "w/o RF" is the opposite, which disables the feedback-driven refinement process, making the Navigator always choose to attempt another candidate plan. The Driver's behavior changes according to different directions of the Navigator.

The ablation results demonstrate that the complete PairCoder achieves the best accuracy, and both multi-plan exploration and feedback-driven refinement play a positive role in code generation.

Table 4: Ablation results for multi-plan exploration (MP) and feedback-driven refinement (RF).

Models	Variants	HumanEval		MBPP		CodeContest	
		Orig	Plus	Orig	Plus	Valid	Test
GPT-3.5 -Turbo	PairCoder	**87.80**	**77.44**	**80.60**	**77.69**	**17.95**	**15.15**
	w/o MP	81.10	73.78	76.20	71.68	11.97	10.91
	w/o RF	74.39	68.29	72.20	68.92	8.55	9.69
DeepSeek -Coder	PairCoder	**85.37**	**76.22**	**78.80**	**75.69**	**13.68**	**14.55**
	w/o MP	78.86	73.17	73.40	71.93	9.40	12.12
	w/o RF	75.61	69.51	68.60	67.92	7.69	10.91

Multi-plan exploration effectively expands the search space away from local optima, leading to considerable improvements. Feedback-driven refinement brings more significant improvement than multi-plan exploration across all benchmarks and foundation models. It indicates that even with a proper solution plan, advanced LLMs still struggle to generate the correct code in one attempt. The two components are more effective for complex problems, yielding substantial relative improvements of 38.86% and 56.35% using GPT-3.5-Turbo on CodeContest-test, respectively, compared to 8.26% and 18.03% on the simpler HumanEval. This is in line with real-world programming practices, where complex problem-solving requires more exploration and refinement.

> **Answer to RQ3:** Both multi-plan exploration and feedback-driven refinement contribute to PairCoder for code generation. They bring more significant improvements for complex problems in CodeContest, which is in line with real-world programming practices.

4.5 RQ4: Cost and Error Analyses

For this RQ, we further investigate the usage of PairCoder.

Cost analysis. We perform a cost analysis for PairCoder and all reproduced approaches on HumanEval, MBPP, and the test set of CodeContest. The cost is measured by two key metrics: the average number of API calls per problem and the average token consumption per problem. For each approach, we record its API requests and responses using GPT-3.5-Turbo. This provides the number of API calls and token consumption, including input tokens and generated output tokens of LLMs. Note that we count API calls to assess the efficiency in code generation, since the time spent is susceptible to uncontrollable factors such as network fluctuation.

For the fairness of comparison, we extend the comparative methods to conduct additional experiments using GPT-3.5-turbo: (i) For prompting techniques, we repeat sampling with a temperature of 0.8 until the generated code passes all public test cases or 10 attempts are reached. (ii) For Self-repair, we also allow up to 10 iterations. This setup ensures that these approaches have the same maximum number of attempts as the iterative approaches in Sect. 4.1. As shown in Table 5, the results demonstrate that simple repetitive sampling can indeed enhance these approaches, but PairCoder remains dominant. It confirms that the effectiveness of PairCoder beyond merely increased computation.

Table 5: Accuracy comparison of prompting approaches and Self-repair using GPT-3.5-Turbo with up to 10 iterations. HumanEval and MBPP are the original versions (the same below unless otherwise specified).

Approaches	HumanEval	MBPP	CodeContest-test
Direct prompting	75.61	72.80	9.09
CoT prompting	76.83	73.00	7.88
SCoT prompting	75.61	73.94	8.48
Self-planning	79.27	76.36	10.90
Self-repair	78.65	75.00	9.09
PairCoder	**87.80**	**80.60**	**15.15**

Table 6: Average number of API calls and tokens (in thousands) using GPT-3.5-Turbo. Note that all approaches are allowed up to 10 iterations.

Approaches	HumanEval		MBPP		CodeContest-test	
	API	Token	API	Token	API	Token
Direct prompting	2.43	1.19	2.31	1.23	8.73	9.79
CoT prompting	2.43	1.36	2.37	1.41	8.62	10.16
SCoT prompting	4.63	4.46	4.61	4.49	17.30	26.57
Self-planning	4.16	3.27	4.67	2.78	17.02	20.03
Self-repair	2.37	2.40	2.36	2.02	8.85	19.82
Reflexion	**8.47**	9.68	**9.58**	**10.28**	-	-
LDB†	-	**23**	-	-	-	-
Self-debugging	2.74	7.74	2.55	6.47	9.07	**38.40**
INTERVENOR	3.85	2.05	3.77	1.83	17.06	13.72
PairCoder	5.28	5.98	5.37	5.09	**17.56**	24.06

Table 6 presents the cost comparison results. Among prompting techniques, Direct and CoT prompting generally have lower costs compared to other approaches, while SCoT and Self-planning demonstrate higher costs due to their complex prompting strategies. For refinement-based approached, Reflexion shows high API call and token consumption on both benchmarks. LDB, reported only for HumanEval, demonstrates the highest token consumption among all approaches. Collaborating the Navigator and the Driver, PairCoder requires more API calls than most approaches except Reflexion. Nevertheless, PairCoder maintains moderate token consumption, particularly when compared to LDB, Reflexion, and Self-debugging across different benchmarks. Furthermore, we observe that all approaches spend higher cost on the challenging CodeContest-test than simpler benchmarks, due to the fact that complex problem-solving requires more model interactions. Despite moderate cost increase, PairCoder significantly improves accuracy across all settings (as shown in Tables 2 and 5), justifying the minor increase cost. Overall, PairCoder achieves better accuracy in code generation while maintaining reasonable costs compared to existing approaches.

Error analysis. We conduct a detailed analysis of the error types encountered by PairCoder using GPT-3.5-Turbo on three benchmarks. Table 7 presents the analysis results. Overall, *Wrong Answer* is the most common error type, indicating that generating functionally correct programs remains a key challenge for code generation. On the relatively simple HumanEval and MBPP benchmarks, PairCoder solves over 80% programming problems, where *Runtime Error* and *Time Limit Exceeded* are rare. However, on the challenging CodeContest-test benchmark, the accuracy largely decreases. Although the causes of errors are still dominated by *Wrong Answer*, there is a notable increase in *Runtime Error* and *Time Limit Exceeded*. The analysis results are consistent with realistic programming practices, where simple problems generally do not encounter unexpected exceptions and efficiency issues. We emphasize the urgent need to further improve the functional correctness of code generation, and also focus on the efficiency and robustness for complex programming problems.

Table 7: Analysis of passes on public test cases T_v, private test cases T_h, and specific error types on T_h, i.e., *Runtime Error* (RE), *Wrong Answer* (WA), and *Time Limit Exceeded* (TLE).

Benchmarks	Pass T_v	Pass T_h	Not pass T_h		
			RE	WA	TLE
HumanEval	95.73%	87.80%	0	11.59%	0.61%
MBPP	93.20%	80.60%	0.80%	18.60%	0
CodeContest-test	21.21%	15.15%	16.36%	60.00%	8.48%

Since only the public test cases T_v are visible during the code generation and refinement stages, their quality directly impacts the final performance. On HumanEval and MBPP, the pass rates on public test cases are over 93%, and 91.72% (87.80/95.73) and 86.48% (80.60/93.20) of them also pass the private test cases T_h, respectively. However, on the challenging CodeContest-test, the pass rate on T_v is only 21.21%, and 71.43% (15.15/21.21) of them eventually pass T_h. As with code testing, public test cases can provide real feedback from the executor, revealing the vulnerability of a program to specific inputs. Referring to Table 1, the low coverage of public test cases limits the ability of PairCoder to facilitate code generation.

Based on the above findings, the evolution of code generation approaches and the expansion of public test cases are both crucial and orthogonal. PairCoder seems powerful enough for simple programming problems, and broader public test cases such as edge cases would bring the accuracy closer to that on the current T_v. For complex problems, it is imperative to enhance the reasoning and programming capabilities inherent in LLMs. Besides, retrieval augmentation [20, 38, 43] and test case generation [5, 39, 42, 51] are potential future improvements in code generation.

Answer to RQ4: Compared to existing approaches, PairCoder achieves superior accuracy with comparable and reasonable cost. *Wrong Answer* is the main error cause, yet efficiency and robustness also deserve concerns for complex problems. In addition to enhancing code generation, test case generation is a promising orthogonal direction.

4.6 Threats to Validity

The first potential threat relates to the generalizability of our evaluation. To mitigate this concern, we carefully select five widely used and representative benchmarks and both closed-source and open-source LLMs for our experiments. Under all these settings, consistently superior accuracy demonstrates the effectiveness of PairCoder. In future work, we plan to further validate the generalizability of PairCoder across a broader range of LLMs and benchmarks, such as multilingual code generation [3, 53].

Another potential threat is data leakage in pre-trained LLMs. For example, DeepSeek-Coder was released after benchmarks like HumanEval were collected, raising the possibility that benchmark samples were unintentionally included in its pre-training corpus. However, any such leakage would affect all baselines equally since they use the same model. Therefore, while leakage may inflate overall accuracy, it does not affect the fairness of our comparative analysis and the relative gains of PairCoder, which consistently shows the largest improvements across all settings.

The third potential threat arises from the zero-shot prompting used in our experiments. Although zero-shot prompting achieves superior accuracy while largely reducing token consumption, we do not rule out the possibility that other instructions or few-shot demonstrations could further improve performance. However, the selection of demonstrations for in-context learning poses a significant challenge, which can greatly influence the behavior of LLMs [12]. We leave the exploration of effective few-shot prompting techniques in future work.

5 RELATED WORK

Recent advancements in LLMs have shown remarkable capabilities in code generation tasks by training on vast amounts of code-containing corpora. Open-source models like InCoder [11], Code Llama [40], WizardCoder [29], and DeepSeek-Coder [14], have depicted performance that matches or even surpasses closed-source commercial models like ChatGPT [33], GPT-4 [34], and Claude [1]. This development has significantly improved software productivity [23, 35] and profoundly affected the progress of intelligent software engineering, attracting substantial work focused on enhancing the code generation capabilities of LLMs.

A key focus is on prompting techniques that guide LLMs to produce intermediate reasoning steps from problem descriptions. Prompting techniques have been proven to effectively improve the code generation performance of LLMs in a plug-and-play manner [18, 22, 25, 26, 46]. SCoT [25] and Self-planning [22] design different formats of intermediate steps, while BRAINSTORM [26] trains a neural ranker model to select the best thought. They leverage prompting techniques to stimulate the reasoning capabilities of LLMs, guiding them to generate more accurate code. However, generating correct code is rarely a one-time effort [7]. Some approaches [5, 21, 27, 31, 41, 52] first generate multiple code solutions and then filter or rank them based on consistency or execution results to obtain the final code. They require substantial computational resources to generate code candidates, which is inefficient and orthogonal to our framework.

Another line of work focuses on refining the generated code [7, 32, 45, 50, 54], where the feedback is obtained from LLMs themselves or external sources. Self-edit [50] trains a fault-aware code editor that employs error messages to refine the generated code. Self-repair [32] investigates the effects of leveraging feedback from diverse sources for code repair, such as humans or LLMs. Self-debugging [7] utilizes explanations generated by LLMs to repair code. INTERVENOR [45] emulates the interactive code repair processes based on compiler feedback. LDB [54] mainly focuses on debugging by leveraging runtime execution information, which is orthogonal to our work. It can be integrated into our PairCoder framework to enhance code repair in Steps 5 and 6. We include competitive baselines in our experiments [7, 22, 25, 32, 45, 54], as shown in Table 2, exclude Self-Edit [50] due to its additional training requirements.

Additionally, several works [10, 16, 37] employ collaborative LLM agents to simulate the full software development lifecycle such as the waterfall model [36], spanning from high-level tasks like requirements analysis and architecture design [16, 37], to low-level roles like coder and tester [10]. A notable example is MetaGPT [16]. In contrast, our PairCoder focuses specifically on the critical coding stage, essentially corresponding to the Engineer agent in MetaGPT. Instead of attempting to simulate the entire development lifecycle, PairCoder uses collaborative agents to emulate pair programming, a well-established and proven software practice directly targeting efficient and high-quality code generation. While differing in scope, our work complements those broader development lifecycle simulations by concentrating on the essential coding component.

6 CONCLUSION

In this paper, we propose the PairCoder framework, which is the first to adapt pair programming practices into LLM-based code generation. It comprises a Navigator agent for high-level planning and a Driver agent for specific implementation, collaborating on code generation via multi-plan exploration and feedback-driven refinement. The Navigator explores multiple plans based on execution feedback from the Driver and historical memory. The Driver follows the guidance of the Navigator to undertake initial code generation, code testing, and refinement. Extensive experiments on diverse benchmarks and LLMs demonstrate the superior accuracy of PairCoder. Our work represents a promising step towards leveraging collaborative agents to facilitate intelligent software development. In future work, we plan to integrate human feedback or external knowledge sources to further enhance the high-level planning capabilities of the Navigator. We will also explore applications of the PairCoder framework to other domains beyond code generation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (No. 62272219) and the Collaborative Innovation Center of Novel Software Technology & Industrialization.

REFERENCES

[1] Anthropic. 2023. Meet Claude. https://www.anthropic.com/claude/. Accessed May 30, 2024.

[2] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The Advantages of Careful Seeding. In *SODA*. SIAM, New Orleans, LA, USA, 1027–1035.

[3] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. Multi-lingual Evaluation of Code Generation Models. In *ICLR*. OpenReview.net, Kigali, Rwanda, 1–19.

[4] 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* 2108.07732 (2021), 1–34.

[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 *ICLR*. OpenReview.net, Kigali, Rwanda, 1–19.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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* 2107.03374 (2021), 1–35.

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *ICLR*. OpenReview.net, Vienna, Austria, 1–80.

[8] Zhoujun Cheng, Jungo Kasai, and Tao Yu. 2023. Batch Prompting: Efficient Inference with Large Language Model APIs. In *EMNLP*. ACL, Singapore, 792–810.

[9] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *ICSE*. IEEE/ACM, Lisbon, Portugal, 70:1–70:13.

[10] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *CoRR* 2304.07590 (2023), 1–38.

[11] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *ICLR*. OpenReview.net, Kigali, Rwanda, 1–26.

[12] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?. In *ASE*. IEEE, Kirchberg, Luxembourg, 761–773.

[13] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *ICSE*. IEEE/ACM, Lisbon, Portugal, 39:1–39:13.

[14] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* 2401.14196 (2024), 1–23.

[15] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *ICLR*. OpenReview.net, Addis Ababa, Ethiopia, 1–16.

[16] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *ICLR*. OpenReview.net, Vienna, Austria, 1–26.

[17] William E. Howden. 1978. Theoretical and Empirical Studies of Program Testing. *IEEE Trans. Softw.* 4 (1978), 293–298.

[18] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation. *CoRR* 2308.08784 (2023), 1–20.

[19] 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 *ICLR*. OpenReview.net, Kigali, Rwanda, 1–17.

[20] Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. 2024. Knowledge-Aware Code Generation with Large Language Models. In *ICPC*. IEEE/ACM, Lisbon, Portugal, 52–63.

[21] 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 *NeurIPS*. Curran Associates, Inc., New Orleans, LA, USA, 13419–13432.

[22] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. *CoRR* 2303.06689 (2023), 1–19.

[23] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *CHI*. ACM, Hamburg, Germany, 455:1–455:23.

[24] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *NeurIPS*. Curran Associates, Inc., New Orleans, LA, USA, 22199–22213.

[25] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured Chain-of-Thought Prompting for Code Generation. *CoRR* 2305.06599 (2023), 1–12.

[26] Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023. Think Outside the Code: Brainstorming Boosts Large Language Models in Code Generation. *CoRR* 2305.10679 (2023), 1–13.

[27] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science* 378 (2022), 1092–1097.

[28] 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 *NeurIPS*. Curran Associates, Inc., New Orleans, LA, USA, 21558–21572.

[29] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *ICLR*. OpenReview.net, Kigali, Rwanda, 1–21.

[30] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. 2023. ClarifyGPT: Empowering LLM-based Code Generation with Intention Clarification. *CoRR* 2310.10996 (2023), 1–21.

[31] 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 *ICML*. PMLR, Honolulu, HI, USA, 26106–26128.

[32] 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 *ICLR*. OpenReview.net, Vienna, Austria, 1–49.

[33] OpenAI. 2022. Introducing ChatGPT. https://openai.com/index/chatgpt/. Accessed May 30, 2024.

[34] OpenAI. 2023. GPT-4 Technical Report. *CoRR* 2303.08774 (2023), 1–100.

[35] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *CoRR* 2302.06590 (2023), 1–19.

[36] Kai Petersen, Claes Wohlin, and Dejan Baca. 2009. The Waterfall Model in Large-Scale Development. In *PROFES*. Springer, Oulu, Finland, 386–400.

[37] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *ACL*. ACL, Bangkok, Thailand, 15174–15186.

[38] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. 2023. From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining. In *ASE*. IEEE/ACM, Luxembourg, 976–987.

[39] Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering. *CoRR* 2401.08500 (2024), 1–10.

[40] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* 2308.12950 (2023), 1–48.

[41] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural Language to Code Translation with Execution. In *EMNLP*. ACL, Abu Dhabi, United Arab Emirates, 3533–3546.

[42] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *NeurIPS*, Vol. 36. Curran Associates, Inc., New Orleans, LA, USA, 8634–8652.

[43] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. ARKS: Active Retrieval in Knowledge Soup for Code

Generation. *CoRR* 2402.12317 (2024), 1–16.

[44] Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. 2023. Can Large Language Models Really Improve by Self-critiquing Their Own Plans? *CoRR* 2310.08118 (2023), 1–6.

[45] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. INTERVENOR: Prompt the Coding Ability of Large Language Models with the Interactive Chain of Repairing. In *Findings of ACL*. ACL, Bangkok, Thailand, 2081–2107.

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*. Curran Associates, Inc., New Orleans, LA, USA, 24824–24837.

[47] L. Williams. 2001. Integrating pair programming into a software development process. In *14th Conference on Software Engineering Education and Training. 'In search of a software engineering profession' (Cat. No. PR01059)*. IEEE, Charlotte, NC, USA, 27–36.

[48] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1585–1608.

[49] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *ICSE*. ACM, Lisbon, Portugal, 37:1–37:12.

[50] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *ACL*. ACL, Toronto, Canada, 769–787.

[51] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. ALGO: Synthesizing Algorithmic Programs with LLM-generated Oracle Verifiers. In *NeurIPS*. Curran Associates Inc., New Orleans, LA, USA, 54769–54784.

[52] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023. Coder Reviewer Reranking for Code Generation. In *ICML*, Vol. 202. PMLR, Honolulu, HI, USA, 41832–41846.

[53] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *KDD*. ACM, Long Beach, CA, USA, 5673–5684.

[54] 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 ACL*. ACL, Bangkok, Thailand, 851–870.