

Code Reffix: A Benchmark for Reflection-Guided Code Repair with Large Language Models

Zaiyuan Di Jianting Chen* Yunxiao Yang Xiaoying Gao

Li Yang Zhihao Wang Yang Xiang

Tongji University, Shanghai, China

{dizaiyuan, tj_chenjt, shxiangyang}@tongji.edu.cn

Abstract

While recent studies have increasingly emphasized the role of reflection in code repair tasks, existing benchmarks still target the repair generation capability of LLMs, lacking fine-grained evaluation of reflection generation capability. To this end, we propose Code Reffix, a benchmark featuring an automated pipeline with oracle reflections and a dual-task protocol to decouple the evaluation of reflection from repair. Through extensive experiments on 14 LLMs and fine-tuning analysis, we aim to pinpoint performance bottlenecks of code repair, quantify reflection quality, and verify the value of reflection optimization. Evaluations reveal that underperforming reflection capabilities of small-scale LLMs remain a major bottleneck for code repair. By quantifying this gap, Code Reffix provides a critical foundation for optimizing LLMs to achieve superior repair performance.

1 Introduction

Large language models (LLMs) have made significant progress in the field of code generation (Peng et al., 2023; Huang et al., 2023; Hong et al., 2023; Qian et al., 2024), but they still struggle to guarantee the correctness of generated code when facing complex programming scenarios, making code repair tasks increasingly important.

Early work (Zhang et al., 2023; Paul et al., 2023; Fan et al., 2023; Jiang et al., 2023) typically adopted a direct repair paradigm, taking error code and auxiliary context (such as defect reports and test feedback) as input to generate repaired code. Such methods view repair as a single-step task, relying excessively on the model’s ability to directly map input to correct code, making them difficult to apply to complex scenarios.

Inspired by the *diagnose-correct* process of human developers, recent work has shifted towards

a reflection repair paradigm (Chen et al., 2023b; Madaan et al., 2024). This paradigm is divided into two phases: the **reflection phase** uses LLMs to analyze error code and auxiliary context, identify root causes, and provide repair suggestions; the **repair phase** uses reflection information to guide the generation of repaired code. Under this paradigm, many studies focus on enhancing repair capabilities by optimizing reflection generation, achieving notable improvements (Ni et al., 2024; Jiang et al., 2024).

Although the code repair paradigm has upgraded from direct repair to reflection repair, code repair benchmarks have not yet shifted. Commonly used benchmarks such as HumanEvalFix (Muennighoff et al., 2023) typically only verify the correctness of the repaired code, which are only applied to evaluate the generation capability in the repair phase, while ignoring the fine-grained quality analysis of the reflection phase. This lack of evaluation dimension makes it difficult for researchers to independently assess the contribution of reflection results to the final repair.

To address this gap, we introduce Code Reffix, a benchmark designed to achieve precise evaluation of code repair through process decomposition and multi-perspective metrics, while providing high-quality corpora to enhance model reflection capabilities. For data collection, we sample a large number of error codes from two public code generation benchmarks (Zhuo et al., 2024; Jain et al., 2024). In the evaluation process, Code Reffix requires models to first generate reflections based on error codes, then leverage the reflections to guide the repair process, and finally validate the repaired codes. To evaluate reflection quality, we construct high-quality oracle reflections for each error code by leveraging the corresponding correct code. This reflection can serve as a standard for measuring the quality of generated reflections and as a label for optimizing model reflection capability. Finally,

*Corresponding author.

we build a dataset containing 7,633 samples. Each sample contains an error code, a task description, test feedback, and an oracle reflection.

Our advantage lies in extending the outcome-based evaluation on final repair to a process-based evaluation targeting the reflection repair paradigm, precisely locating the model’s capability bottleneck. This goal is refined into three progressive levels: 1) **Identification**: Based on the reflection repair paradigm, precisely locate the bottleneck phase restricting model performance; 2) **Measurement**: Targeting the reflection phase identified as the bottleneck, construct multi-perspective metrics to quantify generation quality; 3) **Optimization**: Utilize oracle reflections to explore the potential of reflection capability optimization. Experiments on various closed-source or open-source LLMs indicate that the reflection capability of existing models has obvious space for optimization. Meanwhile, model optimization based on oracle reflections can improve their code repair performance.

In summary, our contributions are as follows:

1. We propose Code Reffix, a code repair benchmark that decouples the evaluation of code reflection and repair capability, supporting fine-grained analysis of LLM repair process.
2. We construct a code repair dataset containing 7,633 samples, each equipped with structured oracle reflections generated under the guidance of correct codes, providing valuable supervision data for the community.
3. We conduct extensive evaluations on 14 LLMs and optimize 2 models using oracle reflections. Experiments show that reflection is a key bottleneck phase currently limiting model repair performance, proving the dual value of oracle reflections in evaluation and optimization.

2 Related Work

Direct Repair. Code repair aims to automatically correct errors in code, typically adopting a direct repair paradigm that maps input data to repaired code. Within this paradigm, many studies improve repair performance by enhancing input data, such as adding execution traces (Gupta et al., 2020; Bouzenia et al., 2023) and test feedback (Zhang et al., 2023; Zheng et al., 2024). However, such a method can only provide shallow information, relying excessively on the model’s generation capability, which is limited in complex scenarios.

Reflection Repair. Some studies have turned

to reflection repair paradigms (Yao et al., 2022; Chen et al., 2023b; Joshi et al., 2023; Shinn et al., 2024; Madaan et al., 2024; Zhong et al., 2024). This paradigm uses LLMs to analyze input content such as error code and provide suggestions to guide subsequent repair. By introducing a reflection process, this paradigm utilizes prior knowledge in LLMs and can effectively improve their repair performance.

Given its potential, recent research has concentrated on a deeper exploration of this reflection process. This includes empirical studies investigating the impact of reflection quality on repair performance (Olausson et al., 2023; Kamoi et al., 2024), efforts to integrate reflection as a supervisory signal (Ni et al., 2024; Jiang et al., 2024; Yang et al., 2024; Zhou et al., 2025), and deployment of reflection during inference to iteratively refine code (Zhong et al., 2024; Shi et al., 2024; Li et al., 2025).

Code Repair Benchmarks. While many studies have proven the critical role of reflection quality in this paradigm, existing code repair benchmarks still pay little attention to the evaluation of reflection. These benchmarks typically only consider the correctness of repaired code, lacking fine-grained analysis of reflection quality (Just et al., 2014; Muenighoff et al., 2023; Jimenez et al., 2023). Recently, some researchers have attempted to build specialized benchmarks for reflection quality evaluation and optimization (Chae et al., 2024).

In comparison, our proposed Code Reffix targets more complex code repair scenarios, introducing fine-grained structured reflection design and diverse evaluation perspectives, which can not only identify key bottlenecks in the repair process but also explore the actual impact of reflection optimization on repair performance.

3 Code Reffix

3.1 Definition and Motivation

Code repair refers to fixing error code into correct code, defined as follows:

Definition 1 (Code Repair). Let \tilde{C} be the given error code, E represent the auxiliary context (e.g., test feedback), and the test suite $T = \{t_1, t_2, \dots, t_n\}$ be a set of test cases. The task aims to construct a generation function that maps \tilde{C} and E to repaired code C :

$$\mathcal{F} : (\tilde{C}, E) \rightarrow C. \quad (1)$$

C is correctly repaired if and only if it passes T , i.e., $\varepsilon(C, T) = 1$. ε is the verification function.

However, the repair work of human developers actually corresponds to a complex cognitive process, which usually includes multiple steps such as understanding errors and developing plans before implementing repair (Brooks, 1999). To facilitate modeling this process, we summarize these preliminary steps as reflection and define reflection repair as follows:

Definition 2 (Reflection Repair). Following Definition 1, this task aims to model code repair as a two-phase generation task. The **reflection phase** aims to construct a reflection generation function $\mathcal{R} : (\tilde{C}, E) \rightarrow I$, mapping error code and auxiliary context to reflection I ; the **repair phase** aims to construct a repair generation function $\mathcal{F} : (\tilde{C}, E, I) \rightarrow C$, using reflection as prior guidance to generate repaired code C . The entire process can be formalized as solving a composite mapping function:

$$C = \mathcal{F}(\tilde{C}, E, \mathcal{R}(\tilde{C}, E)). \quad (2)$$

Recent studies show that incorporating this reflection process can improve repair performance. However, traditional benchmarks typically focus on the evaluation of repair effects, ignoring in-depth research on reflection quality. Consequently, researchers face challenges in discerning the precise cause of repair failures: whether the model failed to comprehend the error, lacked viable improvement ideas (reflection defects), or simply erred during code generation (repair defects). This inability to precisely pinpoint shortcomings hinders targeted model improvement efforts.

To this end, we propose Code Reffix, aimed at decoupling reflection generation from code repair. We are committed to exploring the following three progressive core research questions:

RQ1: Identification. In reflection repair, is the main bottleneck restricting model performance reflection capability or repair capability?

RQ2: Measurement. How to construct metrics from multiple perspectives to quantify reflection quality in a fine-grained way?

RQ3: Optimization. Does targeted optimization of reflection capability have both significant effectiveness and unique value in code repair?

3.2 Dataset Construction

Error Code Sampling. We collected submissions released by BigCodeBench (Zhuo et al.,

Table 1: CODE REFFIX statistics.

| Split | Easy | Hard | Overall |
|--|------|------|---------|
| Train | | | |
| #Tasks | 927 | 275 | 1202 |
| #Samples | 5567 | 1773 | 7340 |
| Avg. #Error Code Len. | 1163 | 1967 | 1357 |
| Avg. #Task Description Len. | 1024 | 1354 | 1104 |
| Avg. #Test Feedback Len. | 486 | 588 | 511 |
| Avg. #Oracle Reflection (Failure Trace) Len. | 853 | 892 | 862 |
| Avg. #Oracle Reflection (Cause Diagnosis) Len. | 147 | 150 | 148 |
| Avg. #Oracle Reflection (Repair Guidance) Len. | 409 | 489 | 428 |
| Avg. #Oracle Reflection Total Len. | 1408 | 1531 | 1438 |
| Eval | | | |
| #Tasks | 217 | 76 | 293 |
| #Samples | 217 | 76 | 293 |
| Avg. #Error Code Len. | 1319 | 2137 | 1531 |
| Avg. #Task Description Len. | 939 | 1308 | 1035 |
| Avg. #Test Feedback Len. | 493 | 499 | 495 |
| Avg. #Oracle Reflection (Failure Trace) Len. | 511 | 527 | 515 |
| Avg. #Oracle Reflection (Cause Diagnosis) Len. | 136 | 142 | 137 |
| Avg. #Oracle Reflection (Repair Guidance) Len. | 346 | 424 | 367 |
| Avg. #Oracle Reflection Total Len. | 993 | 1093 | 1019 |

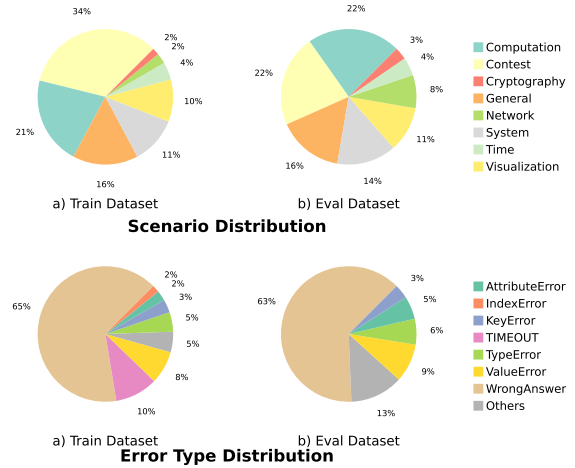


Figure 1: CODE REFFIX distributions. Infrequent error types are grouped into the Others.

2024) and LiveCodeBench (Jain et al., 2024) officials, screened error codes among them, and constructed auxiliary contexts for the error codes. To make auxiliary information unified and detailed, we formalized it as a quadruple $E = (E_{desc}, E_{type}, E_{msg}, E_{case})$, representing task description, error type, error message, and failed test case, respectively, as shown in Fig. 3. Finally, we divided the processed data into training and evaluation sets according to task ID.

Reflection Design. Inspired by the cognitive process of human developers (Weiser, 1982; Brooks, 1999; Vessey, 2007), we formalized the reflection I as a triple $I = (I_{trace}, I_{cause}, I_{guide})$, where I_{trace} is the failure trace, aimed at analyzing code behavior and identifying error locations; I_{cause} is the cause diagnosis, aimed at concisely pointing out the root cause of failure; I_{guide} is the repair guid-

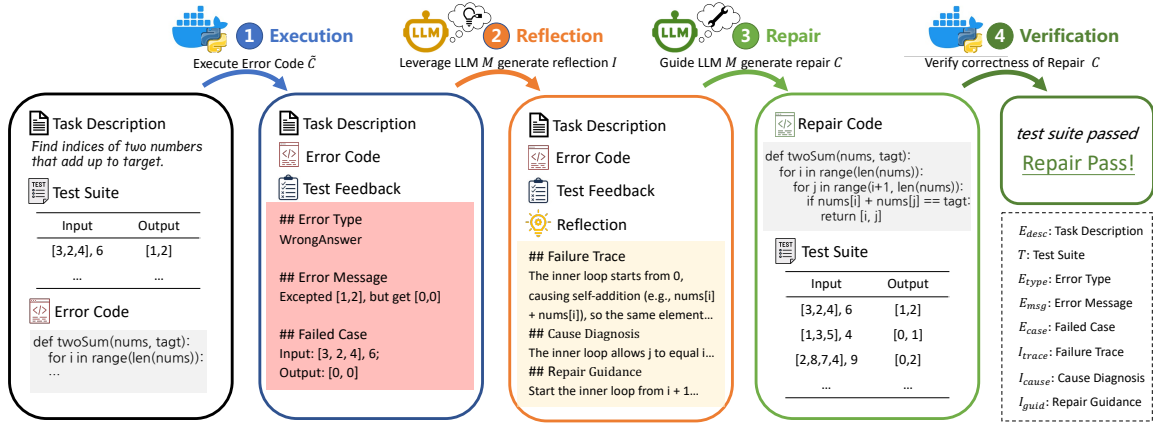


Figure 2: CODE REFFIX pipeline.

ance, aimed at providing specific suggestions to guide subsequent repair. This design can simulate human cognition in a fine-grained way and enhance the quality and interpretability of generated content (Wei et al., 2022).

Code Reffix further introduces an **oracle reflection** \hat{I} as a reference for evaluating I (Fig. 3). Specifically, we use DeepSeek-V3.2 combined with error code \tilde{C} , auxiliary context E , and correct code \hat{C} (where $\varepsilon(\hat{C}, T) = 1$) to synthesize oracle reflection, and prevent data distortion caused by \hat{I} leaking code fragments from \tilde{C} , through checking and rewriting mechanisms.

Finally, we completed the data construction of Code Reffix, covering a wide range of difficulty levels, task scenarios, and error types, as shown in Tab. 1 and Fig. 1. More details are presented in Appendix A.

3.3 Benchmarking Pipeline

As shown in Fig. 2, the benchmarking pipeline includes 4 steps.

Step 1: Execution. Code Reffix executes the error code \tilde{C} within a Docker-based sandbox to capture test feedback and construct the auxiliary context E . As detailed in Section 3.2, this step is integrated into our data construction process.

Step 2: Reflection. Utilizing a LLM M , Code Reffix generates a reflection I by processing \tilde{C} and E through a specialized prompt (see Appendix E), formulated as $I = \mathcal{R}(\tilde{C}, E)$.

Step 3: Repair. With a LLM M , Code Reffix incorporates the reflection I into the prompt to guide the repair process. The repaired code C is

| |
|---|
| ✗ Error Code: \tilde{C} |
| <pre>def twoSum(self, nums, target): for i in range(len(nums)): for j in range(len(nums)): if nums[i] + nums[j] == target: return [i, j]</pre> |
| 📄 Auxiliary Context: E |
| <pre>Task Description (E_{desc}): Find indices of two numbers that add up to target. Error Type (E_{type}): AssertionError Error Message (E_{msg}): Expected [1, 2], but output: [0, 0] Failed Case (E_{case}): Input: [3, 2, 4], 6; Output: [0, 0]</pre> |
| 🗨️ Oracle Reflection: \hat{I} |
| <pre>## Failure Trace (I_{trace}) The inner loop starts from 0, causing self-addition(e.g., nums[i] + nums[i]),so the same element to be used twice (as in the failed output [0, 0]), which violates the constraint. ## Cause Diagnosis (I_{cause}) The inner loop allows j to equal i , violating the requirement for distinct elements. ## Repair Guidance (I_{guid}) Start the inner loop from i + 1 instead of 0 to ensure only distinct pairs are compared.</pre> |

Figure 3: CODE REFFIX sample example

generated as $C = \mathcal{F}(\tilde{C}, E, I)$.

Step 4: Verification. Code Reffix verifies the repaired code C in the sandbox by executing the test suite T , denoted as $\varepsilon(C, T)$.

3.4 Evaluation Protocol

Based on Code Reffix, we evaluate model capabilities from two dimensions: repair and reflection.

Repair Capability Evaluation. We take the repair pass rate P as the metric for this dimension, defining the P of model M under the guidance of reflection I as:

$$P(M, I) = \frac{1}{N_{\tilde{C}}} \sum_i \mathbb{I}(\varepsilon(C_i, T_i) = 1) \quad (3)$$

where $N_{\tilde{C}}$ represents the number of error codes, and $\mathbb{I}(\cdot)$ is the indicator function. In cases where I

Table 2: Validity of reflection and bottleneck identification of reflection repair (RQ1).

| Category | Model | Scale | $P_{\text{fix}}(\%)$ | | | $P_{\text{self}}(\%)$ | | | $P_{\text{guid}}(\%)$ | | | $\Delta P_{\text{self,fix}}(\%)$ | | | $\Delta P_{\text{guid,self}}(\%)$ | | |
|------------------------|-----------------------|-------|----------------------|--------------|--------------|-----------------------|--------------|--------------|-----------------------|--------------|--------------|----------------------------------|-------------|--------------|-----------------------------------|--------------|--------------|
| | | | Easy | Hard | All | Easy | Hard | All | Easy | Hard | All | Easy | Hard | All | Easy | Hard | All |
| Closed-Source | Gemini-3-Flash | - | 62.67 | 30.26 | 54.27 | 65.90 | 30.26 | 56.66 | 71.43 | 36.84 | 62.46 | 3.23 | 0.00 | 2.39 | 5.53 | 6.58 | 5.80 |
| | GPT-5 | - | 54.38 | 31.58 | 48.46 | 57.54 | 35.53 | 51.54 | 71.43 | 43.42 | 64.16 | 3.16 | 3.95 | 3.08 | 13.89 | 7.89 | 12.62 |
| | Claude-Sonnet-4.5 | - | 55.76 | 27.63 | 48.46 | 56.22 | 30.26 | 49.49 | 67.28 | 39.47 | 60.07 | 0.46 | 2.63 | 1.03 | 11.06 | 9.21 | 10.58 |
| | GPT-4o | - | 47.93 | 31.58 | 43.69 | 41.01 | 19.74 | 35.49 | 64.52 | 36.84 | 57.34 | -6.92 | -11.84 | -8.20 | 23.51 | 17.10 | 21.85 |
| | Avg. | - | 55.19 | 30.26 | 48.72 | 55.17 | 28.95 | 48.30 | 68.67 | 39.14 | 61.01 | -0.02 | -1.32 | -0.43 | 13.50 | 10.20 | 12.71 |
| Open-Source General | DeepSeek-V3.2 | 685B | 52.07 | 27.63 | 45.73 | 55.76 | 31.48 | 49.49 | 70.05 | 53.95 | 65.87 | 3.69 | 3.85 | 3.76 | 14.29 | 22.47 | 16.38 |
| | Qwen2.5 | 14B | 27.19 | 21.05 | 25.60 | 32.26 | 10.53 | 26.62 | 67.28 | 43.42 | 61.09 | 5.07 | -10.52 | 1.02 | 35.02 | 32.89 | 34.47 |
| | Llama3 | 8B | 12.90 | 7.89 | 11.60 | 13.82 | 6.58 | 11.95 | 47.93 | 31.58 | 43.69 | 0.92 | -1.31 | 0.35 | 34.11 | 25.00 | 31.74 |
| | Mistral-v0.3 | 7B | 7.37 | 2.30 | 6.14 | 7.83 | 3.95 | 6.83 | 37.33 | 17.11 | 32.08 | 0.46 | 1.65 | 0.69 | 29.50 | 13.16 | 25.25 |
| | Avg. | - | 19.91 | 11.77 | 17.81 | 21.93 | 10.51 | 18.98 | 44.52 | 29.21 | 40.55 | 2.03 | -1.27 | 1.16 | 22.58 | 18.70 | 21.57 |
| Open-Source Code | Qwen2.5-Coder | 14B | 30.41 | 18.42 | 27.30 | 27.65 | 14.47 | 24.23 | 67.28 | 43.42 | 61.09 | -2.76 | -3.95 | -3.07 | 39.63 | 28.95 | 36.86 |
| | Qwen2.5-Coder | 7B | 12.90 | 9.21 | 11.95 | 16.59 | 11.84 | 15.36 | 65.44 | 38.16 | 58.36 | 3.69 | 2.63 | 3.41 | 48.85 | 26.32 | 43.00 |
| | CodeGemma | 7B | 6.45 | 3.95 | 5.80 | 6.91 | 6.58 | 6.83 | 47.93 | 27.63 | 42.66 | 0.46 | 2.63 | 1.03 | 41.02 | 21.05 | 35.83 |
| | DeepSeek-Coder | 6.7B | 11.52 | 6.58 | 10.24 | 11.52 | 3.95 | 9.56 | 55.76 | 28.95 | 48.81 | 0.00 | -2.63 | -0.68 | 44.24 | 25.00 | 39.25 |
| | Avg. | - | 15.32 | 9.54 | 13.82 | 15.67 | 9.21 | 14.00 | 59.10 | 34.54 | 52.73 | 0.35 | -0.33 | 0.17 | 43.44 | 25.33 | 38.74 |
| Community Work | CoffeeEval | 6.7B | 12.44 | 9.21 | 11.60 | 10.60 | 6.58 | 9.56 | 57.14 | 32.98 | 50.85 | -1.84 | -2.63 | -2.04 | 46.54 | 26.40 | 41.29 |
| | ReflectionCoder | 6.7B | 5.53 | 1.32 | 4.44 | 8.76 | 1.32 | 6.83 | 50.23 | 21.05 | 42.66 | 3.23 | 0.00 | 2.39 | 41.47 | 19.73 | 35.83 |
| | Qwen-Reffix(Ours) | 7B | 20.28 | 6.58 | 16.72 | 28.11 | 13.16 | 24.23 | 63.13 | 39.47 | 57.00 | 7.83 | 6.58 | 7.51 | 35.02 | 26.31 | 32.77 |
| | DeepSeek-Reffix(Ours) | 6.7B | 8.76 | 5.26 | 7.85 | 28.11 | 13.16 | 24.23 | 60.83 | 32.98 | 53.58 | 19.35 | 7.9 | 16.38 | 32.72 | 19.82 | 29.35 |
| | Avg. | - | 11.75 | 5.59 | 10.15 | 18.90 | 8.56 | 16.21 | 57.83 | 31.62 | 51.02 | 7.14 | 2.96 | 6.06 | 38.94 | 23.07 | 34.81 |

* Bold indicates global optimal, underline indicates intra-category optimal.

is an empty string (or null), the metric is simplified to $P(M)$, representing the direct repair paradigm.

We further set up the following three repair settings and denote their repair rates as P_{fix} , P_{self} , P_{guid} , respectively:

- **Direct Repair:** Generate repair based on \tilde{C} and E , i.e., $C = \mathcal{F}(\tilde{C}, E)$
- **Self-Reflection:** Repair based on model-generated reflection, i.e., $I = \mathcal{R}(\tilde{C}, E)$, $C = \mathcal{F}(\tilde{C}, E, I)$.
- **Oracle-Guided:** Repair based on oracle reflection, i.e., $C = \mathcal{F}(\tilde{C}, E, \hat{I})$.

By comparing P_{self} with P_{fix} , we can identify the contribution of the reflection phase to repair, and use $\Delta P_{\text{self,fix}} = P_{\text{self}} - P_{\text{fix}}$ to roughly estimate the model’s reflection capability. By leveraging oracle reflection, we treat the model’s performance under oracle guidance (P_{guid}) as the upper bound of its repair capability. Consequently, the optimization space for reflection capability can be quantified by the performance gap $\Delta P_{\text{guid,self}} = P_{\text{guid}} - P_{\text{self}}$.

In Section 4.2, we will separately evaluate the model’s capabilities in each phase and locate its bottlenecks.

Reflection Quality Evaluation. Given reflection I and oracle reflection \hat{I} , we evaluate its quality from two perspectives: downstream performance and semantic quality.

To evaluate **downstream performance**, we introduce the relative pass rate gain G to quantify the improvement over the baseline repair pass rate of

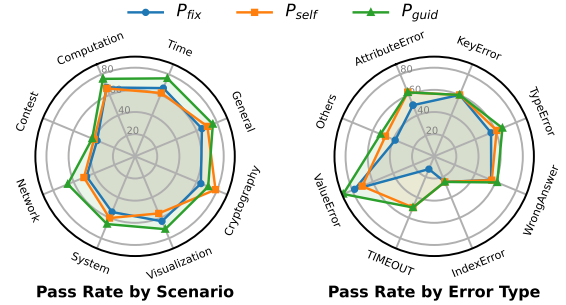


Figure 4: Pass rate distribution of Gemini-3-Flash.

model M . It is computed as:

$$G = \frac{P(M, I) - P(M)}{P(M, \hat{I}) - P(M)} \quad (4)$$

This metric isolates the impact of I through differencing and employs the upper bound \hat{I} for normalization. By doing so, G measures how closely the performance approaches its upper bound, while providing a standardized scale for averaging results across multiple models.

Regarding **semantic quality**, we employ the similarity metric $S = \text{CosineSim}(I_*, \hat{I}_*)$ to evaluate the semantic proximity of I_{trace} , I_{cause} , and I_{guide} to the oracle reflection. This allows for a quantitative assessment of how closely these components align with the ground truth. The correlation between these two perspectives is further explored in Section 4.3.

¹The denominator is always positive in our experiments.

Table 3: Downstream performance of reflection (RQ2). Red indicates negative G .

| Category | Reflector | Scale | DeepSeek-V3.2 | | Qwen2.5-Coder-7B | | DeepSeek-Coder-6.7B | | Avg. | |
|------------------------|-----------------------|-------|---------------|--------------|------------------|--------------|---------------------|--------------|--------------|--------------|
| | | | $P(\%)$ | G | $P(\%)$ | G | $P(\%)$ | G | $P(\%)$ | G |
| Closed-Source | Gemini-3-Flash | - | 54.95 | 0.46 | 45.73 | 0.73 | 36.86 | 0.69 | 45.85 | 0.63 |
| | GPT-5 | - | 53.92 | 0.41 | 45.73 | 0.73 | 37.54 | 0.71 | 45.73 | 0.61 |
| | Claude-Sonnet-4.5 | - | 52.56 | 0.34 | 44.03 | 0.69 | 37.20 | 0.70 | 44.60 | 0.58 |
| | GPT-4o | - | 45.05 | -0.03 | 34.81 | 0.49 | 26.96 | 0.43 | 35.61 | 0.30 |
| | Avg. | - | 51.62 | 0.29 | 42.58 | 0.66 | 34.64 | 0.63 | 42.95 | 0.53 |
| Open-Source General | DeepSeek-V3.2 | 685B | 49.49 | 0.19 | 38.57 | 0.57 | <u>32.08</u> | 0.57 | 40.05 | 0.44 |
| | Qwen2.5 | 14B | 37.88 | -0.39 | 24.91 | 0.28 | 18.43 | 0.21 | 27.07 | 0.03 |
| | Llama3 | 8B | 36.52 | -0.46 | 15.02 | 0.07 | 12.63 | 0.06 | 21.39 | -0.11 |
| | Mistral-v0.3 | 7B | 33.11 | -0.63 | 11.95 | 0.00 | 7.51 | -0.07 | 17.52 | -0.23 |
| | Avg. | - | 39.25 | -0.32 | 22.61 | 0.23 | 17.66 | 0.19 | 26.51 | 0.03 |
| Open-Source Code | Qwen2.5-Coder | 14B | 40.96 | -0.24 | <u>22.53</u> | <u>0.23</u> | <u>16.72</u> | <u>0.17</u> | 26.74 | <u>0.05</u> |
| | Qwen2.5-Coder | 7B | 37.20 | -0.42 | 15.36 | 0.07 | 15.36 | 0.13 | 22.64 | -0.07 |
| | DeepSeek-Coder | 6.7B | 38.23 | -0.37 | 12.97 | 0.02 | 9.56 | -0.02 | 20.25 | -0.12 |
| | CodeGemma | 7B | 35.49 | -0.51 | 9.90 | -0.04 | 9.22 | -0.03 | 18.20 | -0.19 |
| | Avg. | - | 37.97 | -0.39 | 15.19 | 0.07 | 12.72 | 0.06 | 21.96 | -0.08 |
| Community Work | CoffeeEval | 6.7B | 35.84 | -0.49 | 10.92 | -0.02 | 8.87 | -0.04 | 18.54 | -0.18 |
| | ReflectionCoder | 6.7B | 41.64 | -0.20 | 11.26 | -0.01 | 10.92 | 0.02 | 21.27 | -0.07 |
| | Qwen-Reffix(Ours) | 7B | 38.57 | -0.36 | 24.57 | 0.27 | 19.45 | 0.24 | 27.53 | 0.05 |
| | DeepSeek-Reffix(Ours) | 6.7B | 40.96 | -0.24 | 23.21 | 0.24 | 17.75 | 0.19 | 27.31 | <u>0.07</u> |
| | Avg. | - | 39.25 | -0.32 | 17.49 | 0.12 | 14.25 | 0.10 | 23.66 | -0.03 |

4 Evaluation

4.1 Experimental Setup

Evaluated Models. We select 14 closed-source or open-source LLMs², covering various scales and domains, including Gemini-3-Flash, GPT-5, Claude-Sonnet-4.5, GPT-4o, DeepSeek-V3.2 (Liu et al., 2025), Qwen2.5-14B (Qwen et al., 2025), Llama3-8B (Grattafiori et al., 2024), Mistral-7B-v0.3, Qwen2.5-Coder (14B/7B) (Hui et al., 2024), CodeGemma-7B (Team et al., 2024), DeepSeek-Coder-6.7B (Guo et al., 2024), ReflectionCoder(6.7B) (Ren et al., 2025), CoffeeEval(6.7B) (Chae et al., 2024).

In addition, to verify the optimization potential of reflection, we train Qwen-Reffix (based on Qwen2.5-Coder-7B) and DeepSeek-Reffix (based on DeepSeek-Coder-6.7B) utilizing the training dataset of Code Reffix and evaluate them together.

Inference Settings. We access closed-source models and DeepSeek-V3.2 via API; deploy other models using vLLM framework (Kwon et al., 2023) and an NVIDIA RTX 6000 Ada Generation GPU. All inferences use greedy decoding.

Training Settings. We fine-tune Qwen-Reffix and DeepSeek-Reffix using LoRA (Hu et al., 2022) on an NVIDIA RTX 6000 Ada Generation GPU. More training setup details are in Appendix B.

²Unless otherwise specified, models are instruct(chat)-versions.

4.2 Bottleneck Identification of Reflection Repair (RQ1)

We evaluate the performance of 14 LLMs along with 2 fine-tuned models in three repair settings. Tab. 2 demonstrates that while incorporating reflection generally improves performance compared to direct repair, the effectiveness of this improvement depends heavily on the quality of the reflection. The marginal $\Delta P_{\text{self,fix}}$ reveals limited gains from self-reflections, which can even cause severe adverse effects in complex scenarios. For example, while Qwen2.5-14B achieves a 5.07% gain on easy codes, its performance drops sharply to -10.52% on hard codes. Furthermore, using oracle reflections results in significant performance improvements (with a large $\Delta P_{\text{guid,self}}$), particularly for small-scale models (with parameters $\leq 14\text{B}$).

Main bottlenecks in LLMs vary by scale. Large-scale closed-source models and DeepSeek-V3.2 perform well across P_{fix} , P_{self} , and P_{guid} , indicating limited room for reflection improvement and a primary bottleneck in repair capabilities. In contrast, for smaller-scale models, reflection capability becomes the critical bottleneck. When provided with the same oracle reflections, small-scale models achieve performance comparable to larger ones. For example, Qwen2.5-14B achieves P_{guid} of 61.09%, close to 65.87% for DeepSeek-V3.2. This suggests smaller models rely heavily on high-quality reflections due to limited internal analysis and understanding.

To analyze the bottlenecks in large-scale LLMs for repair, we take Gemini-3-Flash as a representa-

Table 4: Semantic quality of reflection (RQ2).

| Category | Model | Scale | BGE-M3 Similarity | | | | Multilingual-E5-Large Similarity | | | | Average Similarity | | | |
|------------------------|-------------------|-------|--------------------|--------------------|--------------------|-------------|----------------------------------|--------------------|--------------------|-------------|--------------------|--------------------|--------------------|-------------|
| | | | I_{trace} | I_{cause} | I_{guide} | Avg. | I_{trace} | I_{cause} | I_{guide} | Avg. | I_{trace} | I_{cause} | I_{guide} | Avg. |
| Closed-Source | Gemini-3-Flash | - | 0.81 | 0.73 | 0.76 | 0.77 | 0.93 | 0.93 | 0.93 | 0.93 | 0.87 | 0.83 | 0.85 | 0.85 |
| | GPT-5 | - | 0.79 | 0.71 | 0.74 | 0.75 | 0.93 | 0.92 | 0.91 | 0.92 | 0.86 | 0.81 | 0.83 | 0.83 |
| | Claude-4.5-Sonnet | - | 0.78 | 0.70 | 0.71 | 0.73 | 0.92 | 0.91 | 0.90 | 0.91 | 0.85 | 0.80 | 0.81 | 0.82 |
| | GPT-4o | - | 0.80 | 0.70 | 0.73 | 0.74 | 0.94 | 0.92 | 0.92 | 0.93 | 0.87 | 0.81 | 0.83 | 0.83 |
| | Avg. | - | 0.79 | 0.71 | 0.74 | 0.75 | 0.93 | 0.92 | 0.92 | 0.92 | 0.86 | 0.81 | 0.83 | 0.83 |
| Open-Source General | DeepSeek-V3.2 | 685B | 0.79 | 0.63 | 0.65 | 0.69 | 0.92 | 0.89 | 0.88 | 0.90 | 0.86 | 0.76 | 0.76 | 0.79 |
| | Qwen2.5 | 14B | 0.77 | 0.66 | 0.69 | <u>0.71</u> | 0.92 | 0.91 | 0.90 | <u>0.91</u> | 0.85 | <u>0.78</u> | 0.79 | <u>0.81</u> |
| | Llama3 | 8B | 0.76 | 0.64 | <u>0.70</u> | <u>0.70</u> | 0.92 | 0.91 | <u>0.91</u> | <u>0.91</u> | 0.84 | 0.77 | <u>0.80</u> | <u>0.81</u> |
| | Mistral-v0.3 | 7B | 0.75 | 0.64 | 0.68 | 0.69 | 0.92 | 0.91 | 0.90 | 0.90 | 0.84 | 0.77 | 0.79 | 0.80 |
| | Avg. | - | 0.77 | 0.64 | 0.68 | 0.69 | 0.92 | 0.90 | 0.90 | 0.91 | 0.84 | 0.77 | 0.79 | 0.80 |
| Open-Source Code | Qwen2.5-Coder | 14B | <u>0.75</u> | <u>0.65</u> | 0.66 | <u>0.69</u> | 0.91 | 0.90 | 0.89 | 0.90 | <u>0.83</u> | 0.77 | 0.78 | 0.79 |
| | Qwen2.5-Coder | 7B | 0.74 | <u>0.65</u> | <u>0.68</u> | <u>0.69</u> | <u>0.92</u> | <u>0.91</u> | 0.90 | <u>0.91</u> | <u>0.83</u> | <u>0.78</u> | <u>0.79</u> | <u>0.80</u> |
| | DeepSeek-Coder | 6.7B | 0.72 | 0.62 | 0.66 | 0.66 | 0.91 | 0.90 | 0.90 | 0.90 | 0.82 | 0.76 | 0.78 | 0.78 |
| | CodeGemma | 7B | 0.72 | 0.60 | 0.66 | 0.66 | 0.91 | 0.90 | 0.90 | 0.90 | 0.81 | 0.75 | 0.78 | 0.78 |
| | Avg. | - | 0.73 | 0.63 | 0.66 | 0.67 | 0.91 | 0.90 | 0.90 | 0.90 | 0.82 | 0.76 | 0.78 | 0.79 |
| Community Work | CoffeeEval | 6.7B | 0.66 | 0.61 | 0.62 | 0.63 | 0.90 | 0.90 | 0.89 | 0.89 | 0.78 | 0.76 | 0.75 | 0.76 |
| | ReflectionCoder | 6.7B | 0.72 | 0.61 | 0.65 | 0.66 | 0.91 | 0.90 | 0.89 | 0.90 | 0.82 | 0.75 | 0.77 | 0.78 |
| | Qwen-Reffix | 7B | 0.81 | <u>0.67</u> | 0.70 | <u>0.73</u> | <u>0.93</u> | 0.93 | <u>0.91</u> | 0.91 | 0.87 | 0.80 | 0.81 | <u>0.82</u> |
| | DeepSeek-Reffix | 6.7B | 0.80 | 0.65 | <u>0.72</u> | <u>0.72</u> | <u>0.93</u> | 0.91 | <u>0.91</u> | <u>0.92</u> | 0.87 | 0.78 | <u>0.82</u> | <u>0.82</u> |
| | Avg. | - | 0.75 | 0.64 | <u>0.67</u> | 0.69 | 0.92 | 0.91 | 0.90 | 0.91 | 0.83 | 0.77 | 0.79 | 0.80 |

tive case (see Fig. 4). While oracle reflections consistently improve repair performance—achieving nearly a 100% pass rate on ValueError—the improvements are limited in other categories. This indicates that, even with perfect reflection information, the model’s inherent implementation capabilities remain a significant bottleneck.

To verify the potential of reflection optimization for small-scale LLMs, we fine-tune 2 models based on the training dataset of Code Reffix. We observe that reflection optimization effectively mitigates the performance bottlenecks of small-scale models. For instance, both models significantly outperform the average level of open-source and community models under the self-reflection settings. This underscores the critical nature of reflection and demonstrates the importance of the reflection optimization.

The results highlight the high quality of our oracle reflections. Open-source code models achieve an average P_{guid} of 52.73% when using oracle reflections, which significantly outperforms their average P_{self} of 14.00% under the self-reflection setting. Remarkably, the 52.73% also exceeds the P_{self} of closed-source models (48.30%).

4.3 Measurement of Reflection Quality (RQ2)

While RQ1 analyzes performance across difficulty levels to identify the primary bottleneck, RQ2 seeks to establish a general benchmark for reflection quality. We aggregate the results across all difficulties to ensure sufficient sample size and maximize statistical significance for our metric validation. To

comprehensively evaluate reflection capability, we examine model-generated reflections from two perspectives.

Downstream Performance. We use reflection to guide 3 downstream models (DeepSeek-V3.2, Qwen2.5-Coder-7B, DeepSeek-Coder-6.7B) to generate repairs respectively, and use G to measure their effects. Tab. 3 shows that, in cross-model comparison, large-scale models can all produce reflections with good guidance effects, while the effects of small-scale models are generally poor.

Moreover, we find G can intuitively reflect the negative effect of reflection, where the reflection may instead hinder the repair process. This phenomenon is particularly significant on DeepSeek-V3.2. Specifically, low-quality reflection often suffers from hallucinations, such as nonexistent code snippets or wrong suggestions. These inaccuracies can mislead downstream models and hinder their understanding of code errors, ultimately leading to repair failures. We provide a case in Appendix D to show the detail.

In fact, reflections of small-scale models are predominantly negative ($\text{Avg.}G < 0$), highlighting a critical need for improvement. Notably, reflection optimization significantly boosted reflection quality, leading both fine-tuned models to achieve positive $\text{Avg.}G$ values, with DeepSeek-Reffix in particular successfully crossing the threshold from negative to positive.

Semantic Quality. We calculate the semantic similarity S between reflection and oracle reflection using BGE-M3 (Chen et al., 2023a) and

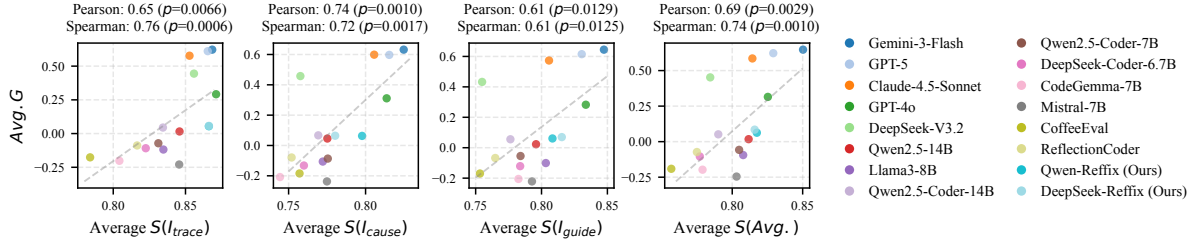


Figure 5: Correlation analysis between semantic quality (average S) and downstream performance (Avg. G).

Multilingual-E5-Large (Wang et al., 2024) respectively. Tab. 4 shows that, in terms of average similarity, the similarity of I_{trace} is significantly higher than that of I_{cause} and I_{guide} , indicating that models find it easier to describe the surface behavior of code but are deficient in deep cause diagnosis and repair suggestions. Consistent with their downstream performance, large-scale models also exhibit generally higher semantic similarity than small-scale models.

We further explored the correlation of metrics under the two perspectives. Fig. 5 shows that S and G have significant positive correlations of Pearson (0.69, $p < 0.01$) and Spearman (0.74, $p < 0.01$). Specifically for each component, the correlation strengths of I_{trace} and I_{cause} are particularly pronounced, reflecting the role of accurate error location and cause diagnosis in reflection.

Given that the calculation of G requires high inference and testing costs, this result validates the feasibility of S as a low-cost proxy and the value of oracle reflection in reflection quality evaluation. To complement this embedding-based measure, we further report an LLM-as-a-judge analysis in Appendix C.

4.4 Reflection Optimization Potential (RQ3)

To explore the feasibility and value of reflection optimization, we compared 3 SFT strategies based on oracle reflections in Code Reffix: optimizing repair code generation with error code (Direct), optimizing reflection generation with error code (Ref-Only), and joint optimization (Joint), and evaluated the fine-tuning effects of 2 models. All strategies follow the same training settings, where the results of Ref-Only correspond to Qwen-Reffix and DeepSeek-Reffix.

Fig. 6 illustrates that different strategies exhibit distinct advantages across various repair settings; specifically, Direct outperforms others in the direct repair setting, whereas Ref-Only excels in the self-

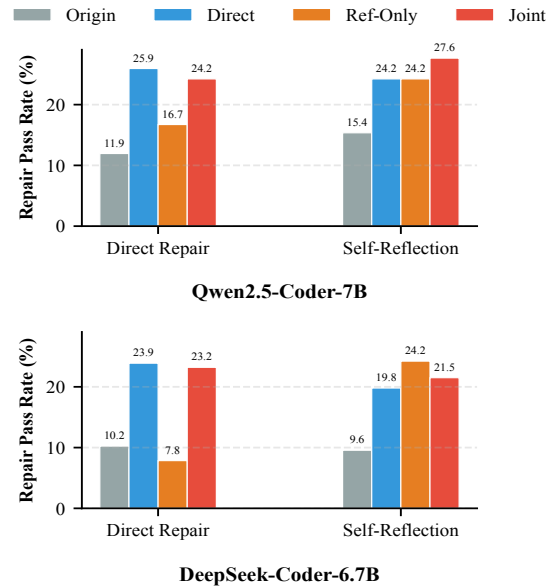


Figure 6: Performance enhancement of diverse SFT strategies on two LLMs using Code Reffix training dataset. The x-axis represents repair settings.

reflection context. Notably, for DeepSeek-Reffix, the Ref-Only strategy achieves the best repair performance across all settings. Given that the oracle-guided setting is impractical in real-world scenarios, we omit further comparisons for it. However, we also observe an impact of the Ref-Only strategy on the model’s direct repair capability. Under the direct repair setting, incorporating reflection process optimization (i.e., Ref-Only, Joint) proves less effective than focusing solely on repair generation. Particularly for DeepSeek-Reffix, its performance even falls significantly below that of the original model (Origin).

Meanwhile, the Joint strategy fails to produce a synergistic effect; it only achieves optimal performance on Qwen2.5-Coder-7B in the self-reflection setting. This suggests that a simple SFT objective may struggle to harmonize repair and reflection tasks, indicating a need for more sophisticated training strategy designs in future research. Neverthe-

less, the experimental results collectively validate the significant potential of reflection optimization for enhancing code repair tasks.

5 Conclusion

We propose Code Reffix, a benchmark designed for reflection repair paradigm. We evaluated 14 current LLMs and analyzed their reflection quality from two perspectives, verifying that reflection is the bottleneck restricting their repair performance. Moreover, we optimized two LLMs based on Code Reffix, demonstrating the feasibility and potential of reflection optimization. In this work, we deliberately focus on function-level repair to isolate additional engineering variables and enable a targeted study of reflection. Although our findings do not automatically generalize to repository-level repair, they still provide useful insights into the importance of reflection quality and the value of decoupling reflection from implementation. Extending the benchmark to repository-level settings remains an important direction for future work.

Limitations

Function-level benchmark scope. Code Reffix is intentionally scoped to function-level repair, which helps isolate the role of reflection without introducing additional confounding factors (e.g., cross-file retrieval, multi-file editing and build/test execution). However, this design also limits the benchmark’s ability to capture the full complexity of repository-level software maintenance scenarios. Therefore, extending the benchmark to repository-level settings is an important direction for future work.

Single oracle reflection per sample. We provide one unified oracle reflection for each sample, which offers a consistent reference for evaluation, bottleneck analysis, and supervised training. However, multiple reflections may lead to the same correct fix, and different models may benefit from different forms of guidance. Therefore, using a single oracle may underestimate the achievable upper bound of some models in the oracle-guided setting, and we leave incorporating multiple valid oracle reflections as future work.

Lack of manual analysis. Limited by cost, we only explored reflection quality analysis based on automated metrics. Although we introduced advanced embedding models and LLM-as-a-judge evaluation to assess reflection quality, this method

may still overlook some subtle but critical semantic deviations. Therefore, our automated metrics should be combined with human evaluation to comprehensively assess reflection quality.

Further exploration of reflection optimization. We verified the feasibility and potential of targeted optimization of reflection capability in RQ3, but only adopted the SFT training strategy. We look forward to the community exploring more reflection optimization methods based on Code Reffix.

Acknowledgments

This work was supported by the National Natural Science Foundation of China U23B2057.

References

- Islem Bouzenia, Yangruibo Ding, Kexin Pei, Baishakhi Ray, and Michael Pradel. 2023. Tracefixer: Execution trace-driven program repair. *arXiv preprint arXiv:2304.12743*.
- Ruven Brooks. 1999. Towards a theory of the cognitive processes in computer programming. *International Journal of Human-Computer Studies*, 51(2):197–211.
- Hyungjoo Chae, Taeyoon Kwon, Seungjun Moon, Yongho Song, Dongjin Kang, Kai Tzu-iunn Ong, Beong-woo Kwak, Seonghyeon Bae, Seung-won Hwang, and Jinyoung Yeo. 2024. Coffee-gym: An environment for evaluating and improving natural language feedback on erroneous code. *arXiv preprint arXiv:2409.19715*.
- Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2023a. [Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation](#). *Preprint*, arXiv:2309.07597.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.
- Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets

- programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695.
- Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.
- Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, and 1 others. 2023. Anpl: towards natural programming with interactive decomposition. *Advances in Neural Information Processing Systems*, 36:69404–69440.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Training llms to better self-debug and explain code. *arXiv preprint arXiv:2405.18649*.
- Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140.
- René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.
- Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When can llms actually correct their own mistakes? a critical survey of self-correction of llms. *Transactions of the Association for Computational Linguistics*, 12:1417–1440.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.
- Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. 2025. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348*.
- Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 workshop on instruction tuning and instruction following*.
- Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*.
- Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.
- Rishov Paul, Md Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna Santos. 2023. Enhancing automated program repair through fine-tuning and prompt engineering. *arXiv preprint arXiv:2304.07840*.

- Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186.
- Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, and 25 others. 2025. [Qwen2.5 technical report](#). *Preprint*, arXiv:2412.15115.
- Houxiang Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Juntao Pan, and Hongsheng Li. 2025. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9999–10020.
- Yuling Shi, Songsong Wang, Chengcheng Wan, Min Wang, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.
- CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, and 1 others. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.
- Iris Vessey. 2007. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(5):621–637.
- Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2024. Multilingual e5 text embeddings: A technical report. *arXiv preprint arXiv:2402.05672*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Mark Weiser. 1982. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452.
- Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *arXiv preprint arXiv:2404.12636*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.
- Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.
- Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.
- Li Zhong, Zilong Wang, and Jingbo Shang. 2024. [Debug like a human: A large language model debugger via verifying runtime execution step by step](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 851–870, Bangkok, Thailand. Association for Computational Linguistics.
- Changzhi Zhou, Xinyu Zhang, Dandan Song, Xiancai Chen, Wanli Gu, Huipeng Ma, Yuhang Tian, Mengdi Zhang, and Linmei Hu. 2025. Refinecoder: Iterative improving of large language models via adaptive critique refinement for code generation. *arXiv preprint arXiv:2502.09183*.
- Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Dataset Details

Statistics. For the scenario, we adopted BigCodeBench’s classification of task scenarios and categorized the samples in LiveCodeBench as Contest; for the error types, we used the error types captured during code execution, and for ease of demonstration, we categorized the less frequently occurring types as Others.

Construction Detail. We constructed our dataset based on BigCodeBench (1140 tasks) and LiveCodeBench (1055 tasks) data up to April 2025. By re-executing the public submissions of the official Top-K leaderboard models, we obtained consistent test feedback and filtered out samples containing

```

## User
{Task Description}
## Assistant
{Error Code}
## User
The code failed during testing. Repair it based on the error
information.
**Error information** (only one error shown even if multiple
exist):
- error type: {Error Type}
- error message: {Error Message}
- failed test case: {Failed Case}

Return only the repaired code, no code block markers or
explanations.
## Assistant

```

Figure 7: Prompt for Direct Repair

incomplete submissions, invalid feedback, or missing correct code. For dataset partitioning, we split the dataset based on task ID; specifically, for LiveCodeBench, we adopted a temporal partitioning strategy, placing later-released tasks into the evaluation set to reduce the risk of data leakage.

We utilize DeepSeek-V3.2 to synthesize oracle reflections, and provide the prompt of oracle reflection generation in Appendix E. Training dataset samples are generated by DeepSeek-V3.2 (Chat) and the integrity of the reflection structure is verified. Evaluation dataset samples are generated by DeepSeek-V3.2 (Thinking) and checked for leaked reference code. Specifically, we filter potentially leaky samples by detecting suggestive keywords, unique identifiers, and overlapping code snippets. Flagged samples are rewritten using DeepSeek-V3.2 (Chat) and manually reviewed to ensure that the oracle reflections can be inferred solely based on buggy code and test feedback.

As an additional quality assurance step, we manually audited a subset of the Code Reffix evaluation set using a predefined rubric. A reflection is considered correct if it identifies at least one key cause consistent with the observed failure and provides actionable repair guidance without revealing the reference implementation. Among the 30 audited evaluation samples, we found 1 clear misdiagnosis. The remaining reflections are generally correct and useful.

Figure 10 presents a representative misdiagnosed reflection. In this case, the oracle reflection focuses on the handling of equal values in the monotonic stack (highlighted in red in ‘Reflection’ section), but that part of the logic is actually correct. In contrast, it overlooks the critical boundary error

in the `count_subarrays` helper function, where the variable `M` is not clipped to the upper bound `A+1` (highlighted in red in ‘Error Code’ section). Therefore, applying the modifications suggested by the oracle reflection cannot fix this defect.

B Training Setup Details

To account for model-specific architectures, distinct training hyperparameters were selected. For Qwen-Reffix, we used a LoRA rank of 32, a LoRA dropout of 0.05, a learning rate of $5e-6$, a batch size of 16, and a training epoch of 2. For DeepSeek-Reffix, we used a LoRA rank of 32, a LoRA dropout of 0.1, a learning rate of $3e-5$, a batch size of 16, and a training epoch of 3. Both models use the AdamW optimizer. The Code Reffix training set was split by task ID, with 1% reserved for validation. In this paper, we report the test performance achieved at the checkpoint with the minimum validation loss.

C Additional Experiments

C.1 LLM-as-a-judge Evaluation.

To complement the semantic-similarity-based evaluation used in RQ2, we further introduce LLM-as-a-judge to assess the semantic quality of reflections. Specifically, we use DeepSeek-V3.2 (Chat) and Qwen3.5-flash to compare generated reflections with oracle reflections, and evaluate each component independently on a 1–5 Likert scale (see Appendix E for the prompt). Figure 8 shows a significant positive correlation between the judge scores and the downstream performance G . In particular, the Pearson correlations range from 0.94 to 0.97, while the Spearman correlations range from 0.91 to 0.92, with all p -values below 10^{-5} . These results provide strong evidence that high-quality reflections, as judged by LLM evaluators, are closely associated with downstream repair effectiveness.

Importantly, the LLM-based analysis leads to conclusions largely consistent with those from semantic similarity. As shown in Tab. 5, across all models, the semantic quality of I_{trace} is consistently higher than that of I_{guid} , suggesting that analyzing surface-level symptoms is easier than generating actionable guidance. In addition, large-scale models consistently achieve higher scores than small-scale models, indicating better overall semantic quality in generated reflections. Taken together, these results suggest that the main findings of RQ2 are robust under both embedding-based and LLM-based

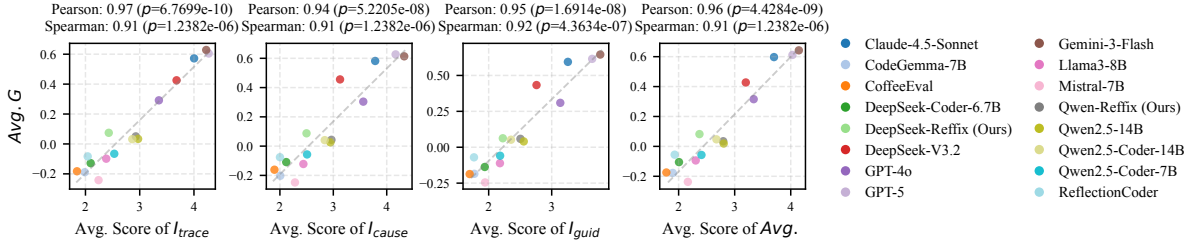


Figure 8: Correlation analysis between semantic quality (based on LLM-as-a-judge) and downstream performance ($Avg.G$).

Table 5: LLM-as-a-judge evaluation for reflection quality (RQ2 supplement). Scores are on a 1–5 Likert scale.

| Category | Reflector | Scale | DeepSeek-V3.2 | | | | Qwen3.5-flash | | | | Avg. Score | | | |
|---------------------|-------------------|-------|---------------|-------------|-------------|-------------|---------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| | | | I_{trace} | I_{cause} | I_{guid} | Avg. | I_{trace} | I_{cause} | I_{guid} | Avg. | I_{trace} | I_{cause} | I_{guid} | Avg. |
| Closed-Source | Gemini-3-Flash | - | 4.13 | 4.20 | 3.64 | 3.99 | 4.32 | 4.50 | 3.96 | 4.26 | 4.22 | 4.35 | 3.80 | 4.12 |
| | GPT-5 | - | 4.25 | 4.11 | 3.61 | 3.99 | 4.25 | 4.23 | 3.72 | 4.06 | 4.25 | 4.17 | 3.66 | 4.03 |
| | Claude-4.5-sonnet | - | 4.03 | 3.79 | 3.29 | 3.71 | 3.99 | 3.83 | 3.24 | 3.69 | 4.01 | 3.81 | 3.27 | 3.70 |
| | GPT-4o | - | 3.29 | 3.33 | 2.98 | 3.20 | 3.48 | 3.76 | 3.27 | 3.50 | 3.38 | 3.55 | 3.12 | 3.35 |
| | Avg. | - | 3.92 | 3.86 | 3.38 | 3.72 | 4.01 | 4.08 | 3.55 | 3.88 | 3.97 | 3.97 | 3.46 | 3.80 |
| Open-Source General | DeepSeek-V3.2 | 685B | 3.78 | <u>3.17</u> | <u>2.74</u> | <u>3.23</u> | <u>3.62</u> | 3.13 | <u>2.73</u> | 3.16 | <u>3.70</u> | 3.15 | <u>2.73</u> | 3.19 |
| | Qwen2.5 | 14B | 2.87 | 2.73 | 2.41 | 2.67 | 3.04 | <u>3.19</u> | 2.67 | 2.97 | 2.95 | 2.96 | 2.54 | 2.82 |
| | Llama3 | 8B | 2.15 | 2.15 | 1.96 | 2.09 | 2.58 | 2.73 | 2.37 | 2.56 | 2.37 | 2.44 | 2.16 | 2.32 |
| | Mistral-v0.3 | 7B | 2.00 | 2.00 | 1.77 | 1.92 | 2.54 | 2.62 | 2.10 | 2.42 | 2.27 | 2.31 | 1.93 | 2.17 |
| | Avg. | - | 2.70 | 2.51 | 2.22 | 2.48 | 2.95 | 2.92 | 2.47 | 2.78 | 2.82 | 2.72 | 2.34 | 2.63 |
| Open-Source Code | Qwen2.5-Coder | 14B | 2.74 | 2.61 | 2.30 | 2.55 | <u>3.02</u> | 3.04 | 2.45 | 2.84 | 2.88 | 2.83 | <u>2.38</u> | 2.70 |
| | Qwen2.5-Coder | 7B | 2.34 | 2.29 | 2.09 | 2.24 | 2.77 | 2.75 | 2.30 | 2.61 | 2.55 | 2.52 | 2.19 | 2.42 |
| | DeepSeek-Coder | 6.7B | 1.79 | 1.78 | 1.68 | 1.75 | 2.39 | 2.42 | 2.16 | 2.32 | 2.09 | 2.10 | 1.92 | 2.03 |
| | CodeGemma | 7B | 1.66 | 1.65 | 1.57 | 1.63 | 2.27 | 2.31 | 1.96 | 2.18 | 1.96 | 1.98 | 1.77 | 1.90 |
| | Avg. | - | 2.13 | 2.08 | 1.91 | 2.04 | 2.61 | 2.63 | 2.22 | 2.49 | 2.37 | 2.36 | 2.06 | 2.26 |
| Community Work | Coffee-Gym | 6.7B | 1.54 | 1.55 | 1.48 | 1.52 | 2.12 | 2.19 | 1.97 | 2.09 | 1.83 | 1.87 | 1.72 | 1.81 |
| | ReflectionCoder | 6.7B | 1.75 | 1.73 | 1.59 | 1.69 | 2.37 | 2.27 | 1.98 | 2.21 | 2.06 | 2.00 | 1.79 | 1.95 |
| | Qwen-Reffix | 7B | 2.86 | 2.84 | 2.43 | 2.71 | 2.94 | 3.04 | 2.56 | 2.84 | 2.90 | 2.94 | 2.49 | 2.78 |
| | DeepSeek-Reffix | 6.7B | 2.40 | 2.37 | 2.16 | 2.31 | 2.50 | 2.65 | 2.35 | 2.50 | 2.45 | 2.51 | 2.25 | 2.40 |
| | Avg. | - | 2.14 | 2.12 | 1.92 | 2.06 | 2.48 | 2.54 | 2.21 | 2.41 | 2.31 | 2.33 | 2.07 | 2.23 |

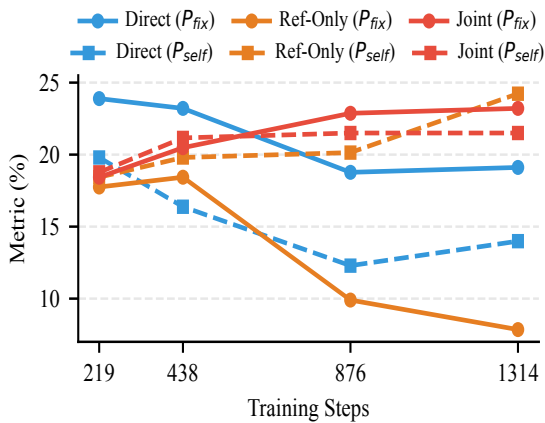


Figure 9: Training performance variation curves of DeepSeek-Coder-6.7B under different SFT strategies.

evaluation.

C.2 Training-dynamics Analysis.

In Section 4.4, we observed that joint optimization does not produce a consistent synergistic ef-

fect. Figure 9 provides one possible explanation by showing that the repair and reflection objectives follow different optimization dynamics. Specifically, Direct performs best at early checkpoints on the direct-repair setting but then declines, while Ref-Only improves more steadily on the self-reflection setting and performs worse on the direct-repair setting as training proceeds. In contrast, Joint shows a more balanced trend: it continues to improve on the metrics at later checkpoints, but does not match the best reflection performance achieved by Ref-Only. This suggests that the current joint SFT objective tends to learn a compromise solution rather than simultaneously optimizing both objectives. One plausible reason is that the two tasks differ in both data distribution and learning objective: repair is closer to a many-to-one mapping from diverse error implementations to a correct solution, whereas reflection requires error localization and causal analysis and is therefore inherently more diverse. Under the current joint setup with a simple combined SFT objective, the model may not automatically balance

Table 6: Self-reflection performance P_{self} (%) of several representative models under multiple reflection rounds.

| Model | Scale | $k = 1$ | $k = 2$ | $k = 3$ |
|---------------|-------|---------|---------|---------|
| Qwen2.5-Coder | 7B | 15.36 | 20.14 | 22.18 |
| Qwen2.5-Coder | 14B | 24.23 | 32.76 | 36.52 |
| Qwen2.5 | 14B | 26.62 | 34.81 | 37.88 |
| DeepSeek-V3.2 | 685B | 49.49 | 61.77 | 65.87 |

these two objectives and may therefore converge to a compromise solution.

C.3 Multi-round Reflection.

We further study multi-round reflection in the self-reflection setting. Starting from the initial (\tilde{C}, E) , we repeat the “reflect–repair–verify” loop for up to k rounds, updating \tilde{C} and E after each round.

Tab. 6 shows the self-reflection performance P_{self} of several representative models under different numbers of reflection rounds. Additional rounds consistently improve the pass rate, but the marginal gains gradually diminish. Importantly, these results do not change our main conclusion: for small-scale models, the main bottleneck remains reflection capability, and even with multiple rounds they still lag behind the oracle-reflection setting. Since the first round already brings most of the gains, we use $k = 1$ as the default setting to keep evaluation cost controllable and avoid introducing additional variables such as search policies.

D Case Study

In section 4.3, we observed that when the downstream performance G is low, it can actually hinder the repair process. We present a concrete example in Fig. 11: the original code failed to add the Z-score column to the returned variable ‘outliers’. The reflection model (Qwen2.5-Coder-7B) attributed the error to the code attempting to access the non-existent Z_score column during plotting (hallucination) and suggested modifying the actually correct plotting code. This caused the downstream model (DeepSeek-V3.2) to be distracted by false information, ignoring the real bug and ultimately failing to fix it.

E Prompts

E.1 Oracle Reflection Generation Prompts

Through multi-round interaction and continuous optimization with advanced LLM, we design the generation prompt for oracle reflection. We divide the prompt into two parts: system and user, as

shown in Fig. 12. The former describes the task scenario, critical rules, output format (consistent with the design of reflection in this paper), and three manually written examples to help the LLM understand the task; the latter includes the error code to be processed, its auxiliary context, and related correct code.

E.2 Direct Repair Prompt

We model the direct repair as a two-round dialogue. The first round is code generation, directly utilizing the error codes in the dataset. The second round is repair generation. The prompt is shown in Fig. 7.

E.3 Reflection Repair Prompts

We model the reflection repair as a 3-round dialogue. Round 1 is code generation, directly utilizing error codes from the dataset. Round 2 is reflection generation; in self-reflection setting, this content is generated by the model; in oracle-guided setting, it directly uses oracle reflections. Round 3 is repair generation. For ease of parsing, we require models to wrap their reflections with special tokens. For locally deployed offline models, considering their limited generation capabilities, we only require them to return reflections in Markdown format. The specific content of the prompts is shown in Fig. 13 and Fig. 14.

E.4 LLM-as-a-judge Prompt

Fig. 15 shows the prompt of LLM-as-a-judge evaluation.

Task Description

You are given an integer array `nums` and a positive integer `k`. Return the sum of the maximum and minimum elements of all subarrays with at most `k` elements.

Example 1

Input: `nums = [1, 2, 3]`, `k = 2`. Output: 20

Explanation: The subarrays of `nums` with at most 2 elements are:

Subarray Min Max Sum

[1] 1 1 2

[2] 2 2 4

[3] 3 3 6

[1, 2] 1 2 3

[2, 3] 2 3 5

Total 20

You should write self-contained code starting with:

```
""
class Solution:
    def minMaxSubarraySum(self, nums: List[int], k: int) -> int:
        ""
```

Error Code

from typing import List

```
class Solution:
    def minMaxSubarraySum(self, nums: List[int], k: int) -> int:
        n = len(nums)
```

Helper to compute sum of contributions using boundaries L and R and length limit k

def count_subarrays(L: int, R: int, k: int) -> int:

We count pairs (a, b) with $0 \leq a \leq L$, $0 \leq b \leq R$, $a + b + 1 \leq k$

Let $A = \min(L, k - 1)$. Only a up to A contribute.

$A = \min(L, k - 1)$

if $A < 0$:

return 0

If $R \geq k - 1$, then for all a in $[0..A]$:

$b_{\max} = \min(R, k - 1 - a) = k - 1 - a$

so count = $\sum_{a=0..A} (k - a)$

if $R \geq k - 1$:

$\sum_{a=0..A} (k - a) = (A + 1) * k - (A * (A + 1)) // 2$

return $(A + 1) * k - (A * (A + 1)) // 2$

Otherwise $R < k - 1$. Let M be first a where $k - 1 - a \leq R$:

Solve $k - 1 - a \leq R \Rightarrow a \geq k - 1 - R$

$M = \max(0, k - 1 - R)$

a in $[0..M - 1]$: $b_{\max} = R \Rightarrow$ contributes (R+1) each

$cnt1 = (R + 1) * M$

a in $[M..A]$: $b_{\max} = k - 1 - a \Rightarrow$ contributes (k - a) each

if $A \geq M$:

length = $A - M + 1$

$\sum_{a=M..A} (k - a) = \text{length} * k - \sum_{a=M..A} a$

$\text{sum}_a = (A * (A + 1) // 2) - ((M - 1) * M // 2)$

$cnt2 = \text{length} * k - \text{sum}_a$

else:

$cnt2 = 0$

return $cnt1 + cnt2$

Figure 10: Representative Misdiagnosed Reflection (Part I).

```

# Monotonic stack to get prev and next smaller (or larger) indices
def boundaries(is_min: bool):
    prev = [-1] * n
    next_ = [n] * n
    stack = []
    for i in range(n):
        # For min: we want prev strictly less, next less-or-equal
        # For max: prev strictly greater, next greater-or-equal
        while stack:
            j = stack[-1]
            if (is_min and nums[j] > nums[i]) or (not is_min and nums[j] < nums[i]):
                # j cannot dominate i for prev boundary
                stack.pop()
            else:
                break
        prev[i] = stack[-1] if stack else -1
        stack.append(i)
    # clear and do next
    stack.clear()
    for i in range(n-1, -1, -1):
        while stack:
            j = stack[-1]
            if (is_min and nums[j] >= nums[i]) or (not is_min and nums[j] <= nums[i]):
                stack.pop()
            else:
                break
        next_[i] = stack[-1] if stack else n
        stack.append(i)
    return prev, next_

total = 0
# sum of minimums
prev_min, next_min = boundaries(is_min=True)
for i in range(n):
    L = i - prev_min[i] - 1
    R = next_min[i] - i - 1
    cnt = count_subarrays(L, R, k)
    total += nums[i] * cnt

# sum of maximums
prev_max, next_max = boundaries(is_min=False)
for i in range(n):
    L = i - prev_max[i] - 1
    R = next_max[i] - i - 1
    cnt = count_subarrays(L, R, k)
    total += nums[i] * cnt

return total

```

Test Feedback
error_info: Expected 28959597955, but output: 141577768327...
error_type: WrongAnswer
failed_case: 'input': [37749,738200,842016,405637, ... (Too long. Truncated)]

Reflection
Analysis
The code attempts to compute the sum of min+max for all subarrays of length k using a monotonic stack to determine for each element the range where it is the minimum or maximum. The execution output is significantly larger than expected, indicating severe overcounting. **The primary issue lies in the boundary calculations: for equal elements, the stack conditions do not enforce the necessary strictness on one side to avoid double-counting.** Specifically, for minimums, the current code allows previous elements with equal value (\leq) and requires next elements to be strictly less, which incorrectly inflates the right boundary R when equal elements are present. Similarly for maximums, leading to overcounted subarrays where the element is not uniquely the min/max.

Root Cause
The monotonic stack conditions incorrectly handle equal elements, causing inflated left or right boundaries and overcounting contributions from duplicate values.

Fix Suggestion
Modify the boundaries function to enforce correct strictness: For $is_min=True$ (minimums), in the forward loop change the pop condition to `nums[stack[-1]] >= nums[i]` so prev is strictly less; in the reverse loop change to `nums[stack[-1]] > nums[i]` so next is less-or-equal. For $is_min=False$ (maximums), forward loop change to `nums[stack[-1]] <= nums[i]` for prev strictly greater; reverse loop change to `nums[stack[-1]] < nums[i]` for next greater-or-equal. This ensures unique assignment and correct L/R counts.

Figure 10: Representative Misdiagnosed Reflection (Part II).

Task Description

Identifies and plots outliers in the 'closing_price' column of a given DataFrame using the Z-Score method. Constants: - Z-Score threshold for identifying outliers is customizable via the 'z_threshold' parameter. >>> df2 = pd.DataFrame({ ... 'closing_price': [10, 20, 30, 40, 50, 100] ... }) >>> outliers2, plot2 = task_func(df2, z_threshold=1.5)

The function should output with:

tuple: A tuple containing the following elements:

pandas.DataFrame: A DataFrame containing the outliers in the 'closing_price' column.

matplotlib.axes._axes.Axes: The plot object displaying the outliers, if x-axis label 'Index', y-axis label 'Closing Price', and title 'Outliers in Closing Prices'.

You should write self-contained code starting with:

```
...
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import zscore
def task_func(df, z_threshold=2):
    ...
```

Error Code

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import zscore
def task_func(df, z_threshold=2):
    # Calculate Z-Scores for the 'closing_price' column
    z_scores = zscore(df['closing_price'])
    # Identify outliers based on the Z-Score threshold
    outliers = df[np.abs(z_scores) > z_threshold]
    # Plot the data and highlight outliers
    fig, ax = plt.subplots()
    ax.plot(df.index, df['closing_price'], label='Closing Price', marker='o')
    ax.scatter(outliers.index, outliers['closing_price'], color='red', label='Outliers', zorder=5)
    # Set plot labels and title
    ax.set_xlabel('Index')
    ax.set_ylabel('Closing Price')
    ax.set_title('Outliers in Closing Prices')
    ax.legend()
    # Show the plot
    plt.show()
    return outliers, ax
```

Test Feedback

error_info: 'Z_score'

error_type: KeyError

failed_case:

```
def test_case_2(self):
    df2 = pd.DataFrame({
        'closing_price': [10, 20, 30, 40, 50, 100]
    })
    outliers2, plot2 = task_func(df2, z_threshold=1.5)
    self.assertEqual(outliers2['closing_price'].tolist(), [100])
    self.assertEqual(outliers2['Z_score'].tolist(), [2.004094170098539])"""
```

Reflection

Analysis

The code calculates the Z-Scores for the 'closing_price' column and identifies outliers based on the Z-Score threshold. However, *when plotting the outliers, the code attempts to access a column named 'Z_score'*, which does not exist in the DataFrame. This leads to a KeyError.

Root Cause

The error is caused by the code trying to access a non-existent column 'Z_score' in the DataFrame.

Fix Suggestion

To fix the error, the code should remove the attempt to access the 'Z_score' column when plotting the outliers. Instead, *it should plot the Z-Scores directly*. Here is the suggested change:

```
...python
# Plot the data and highlight outliers
fig, ax = plt.subplots()
ax.plot(df.index, df['closing_price'], label='Closing Price', marker='o')
ax.scatter(outliers.index, z_scores[outliers.index], color='red', label='Outliers', zorder=5)
...

```

This change ensures that the plot correctly displays the Z-Scores of the outliers, avoiding the KeyError.

Figure 11: Case study: Low-quality reflection hinders the repair of the downstream model.

System

You are an expert coding agent and educator. Your task is to analyze a failed code submission and provide structured repair guidance.

CRITICAL RULES:

- **Blind Analysis****: You have access to a reference solution internally to verify the bug, but your response must be written as if no reference exists. ****Never mention or hint at the reference solution.****
- **Style Mimicry****: Adopt the user's coding style. Do not introduce variable names from the reference solution.
- **Minimal Invasive Fix****: Prioritize fixing the existing logic. Do not rewrite the algorithm unless it is fundamentally impossible (e.g., $O(N^2)$ logic on $O(N)$ constraints, or mathematically wrong).
 - If a pivot is required, explicitly state the unfixable point (e.g., complexity, constraint) in `<|reasoning|>`.
 - Provide specific architectural direction (e.g., data structures) in `<|suggestion|>` instead of full code.
- **Sandbox Noise****: Critically evaluate the execution feedback. If error messages seem inconsistent (e.g., claiming undefined functions that clearly exist), explicitly note it in `<|reasoning|>` and continue to provide a code-based fix.

Response Format:

`<|reasoning|>`Detailed analysis of failed code behavior, where its behavior diverges from the expected result, and why the error message arises.`<|endofblock|>`
`<|cause|>`The specific logical flaw or syntax error that directly leads to the failure (one sentence if possible).`<|endofblock|>`
`<|suggestion|>`Concrete repair suggestion describing what to change and why. Do not regenerate the full code solution.`<|endofblock|>`

Strictly format your response with above special tokens and follow these constraints.

- Keep block content complete and concise. Max 6 sentences per block.
- No full code rewrites. Use directive descriptions (e.g., "Change loop start from 0 to i+1").
- Output NOTHING outside the three blocks.

Examples:

Example 1 (Minimal Fix Rather Than Forced Align With Reference)

- Problem: Two Sum (Find indices of two numbers that add up to target).

- Incorrect Code:

class Solution:

```
def twoSum(self, nums: List[int], target: int) -> List[int]:
    for i in range(len(nums)):
        for j in range(len(nums)):
            if nums[i] + nums[j] == target:
                return [i, j]
```

- Reference Solution: Uses a Hash Map ($O(N)$).

- Execution Feedback:

error type: Wrong Answer

failed test case: {'nums': [3, 2, 4], 'target': 6}

error message: Expected [1, 2], but output: [0, 0]

- Response:

`<|reasoning|>`The code logically flawed because the inner loop starts from 0. This causes the code to check `nums[i] + nums[i]`, effectively using the same element twice (as seen in the failed test case returning `[0, 0]`), which violates the problem constraints. The approach uses a brute force $O(N^2)$. While not optimal, we can still focus on fixing this index logic within the current approach rather than suggesting a completely different one like Hash Map.`<|endofblock|>`

`<|cause|>`The inner loop allows using the same index twice (`j == i`), violating the "two distinct elements" requirement.`<|endofblock|>`

`<|suggestion|>`Modify the inner loop range to start from `i + 1` instead of `0`. This ensures only comparing distinct pairs of elements and avoids using the same element twice.`<|endofblock|>`

Example 2 (Sandbox Noise)

- Problem:

Given a string `s`, return the longest palindromic substring in `s`.

- Incorrect Code:

def helper(s, l, r): ... # Correctly defined

class Solution:

```
def longestPalindrome(self, s):
    res = ""
    for i in range(len(s)):
        tmp = helper(s, i, i) # Only odd length
        if len(tmp) > len(res): res = tmp
    return res
```

- Reference Solution:

def helper(s, l, r): ... # Same as above

class Solution:

```
def longestPalindrome(self, s):
```

Figure 12: Oracle Generation Prompt (Part I).

```

res = ""
for i in range(len(s)):
    # Check both odd (i,i) and even (i,i+1)
    res = max(res, helper(s, i, i), helper(s, i, i+1), key=len)
return res

```

- Execution Feedback:
error type: NameError
failed test case: test_even_palindrome ...
error message: name 'helper' is not defined

- Response:

<|reasoning|>The NameError likely comes from context/scope (helper not in execution context), not the core logic. Since helper is clearly defined, it may be a code-sandbox noise. ignoring this and analyzing the logic: The failed code only expands around `(i, i)`, which detects odd-length palindromes (like "aba"). It fails to expand around `(i, i+1)`, which is necessary to detect even-length palindromes (like "bb"). This omission causes it to miss the longest palindrome in the failed test case.<|endofblock|>

<|cause|>Missing edge case handling for even-length palindromes due to single-center expansion strategy.<|endofblock|>

<|suggestion|>Inside the main loop, call helper(s, i, i+1) in addition to helper(s, i, i). Compare both results against res and update if larger. Just to be on the safe side, define the helper inside the class like `def helper(self, s, l, r)` and call via self.helper(...).<|endofblock|>

****Example 3**** (Algorithmic pivot)

- Problem:
Return the minimum cost to paint n houses such that no two adjacent houses have the same color, and houses equidistant from the ends (i and n-1-i) have different colors.

- Incorrect Code:

```

# Sequential DP approach
for i in range(n):
    if i < n//2:
        dp[i] = cost[i] + min(prev) # Ignores equidistant constraint
    else:
        dp[i] = cost[i] + min(prev, dp[n-i-1]) # Tries to check back, but dp[n-i-1] is already fixed

```

- Reference Solution:

```

# Pairwise DP approach
for pair_idx in range(n//2):
    # Decide colors for both house[i] and house[n-1-i] simultaneously
    # Enforce color[i] != color[n-1-i] and adjacent constraints

```

- Execution Feedback:
error type: Wrong Answer
failed test case: {'input': 4...}
error message: Expected 9, but output: 6

- Response:

<|reasoning|>The failed code uses a linear DP iterating from `0` to `n-1`. It fails because `dp[0]` is finalized without knowing `dp[n-1]`'s color choice, but `dp[n-1]` is constrained by `dp[0]`. The code attempts to check `dp[n-i-1]` when processing the second half, but the values in `dp[n-i-1]` were computed without awareness of the future constraint. This sequential processing creates an unresolvable circular dependency for the bidirectional equidistant constraints.<|endofblock|>

<|cause|>Sequential processing creates a circular dependency for bidirectional constraints; the correct approach requires defining state based on pairs of houses.<|endofblock|>

<|suggestion|>Change the DP strategy to process houses in equidistant pairs simultaneously. Define the state as `dp[pair_idx][left_color][right_color]`. Iterate through pairs `(i, n-1-i)` and compute costs for valid `(left, right)` color combinations that satisfy both adjacent and equidistant constraints.<|endofblock|>

User

****Input**:**

- Problem:
\$context

- Incorrect Code:
\$code

- Execution Feedback:
- error type: \$error_type
- error message: \$feedback
- failed test case: \$test

- Reference Solution:
\$standard_code

Again :

- Never mention the existence of reference solution.
- Suggest switching to a reference-style approach only when the original architecture is impractical to fix—and never expose any of its code.

****Your Response**** (wrapped with special tokens, no redundant content outside; keep the content complete and concise):

Figure 12: Oracle Generation Prompt (Part II).

```

## User
{Task Description}
## Assistant
{Error Code}
## User
The code failed during testing. Analyze the error based on the error information and provide a repair suggestion.
**Error information** (only one error shown even if multiple exist):
- error type: {Error Type}
- error message: {Error Message}
- failed test case: {Failed Case}

**Your response** (strict format):
<|reasoning|>Step-by-step trace of how the code handles the given test, where its behavior diverges from the expected result,
and why the error message arises<|endofblock|>
<|cause|>The specific logical flaw or syntax error that directly leads to the failure (one sentence if possible)<|endofblock|>
<|suggestion|>Concrete repair suggestion describing what to change and why (Do not regenerate full solution)<|endofblock|>
## Assistant
{Reflection}
## User
Repair the code based on your suggestion. Return only the repaired code, no code block markers or explanations.
## Assistant

```

Figure 13: Reflection Repair Prompts for Online LLMs.

```

## User
{Task Description}
## Assistant
{Error Code}
## User
The code failed during testing. Analyze the error based on the error information and provide a repair suggestion.
**Error information** (only one error shown even if multiple exist):
- error type: {Error Type}
- error message: {Error Message}
- failed test case: {Failed Case}

Structure your response using the following three Markdown headers exactly:
## Analysis
[Step-by-step trace of how the code handles the given test, where its behavior diverges from the expected result, and why the
error message arises]

## Root Cause
[The specific logical flaw or syntax error that directly leads to the failure (one sentence if possible)]

## Fix Suggestion
[Concrete repair suggestion describing what to change and why (Do not regenerate full solution)]
## Assistant
{Reflection}
## User
Repair the code based on your suggestion. Return only the repaired code, no code block markers or explanations.
## Assistant

```

Figure 14: Reflection Repair Prompts for Offline LLMs.

Given a coding problem description, error code, error information, a reflection on the error, and an oracle reflection, your task is to compare the given reflection with the oracle reflection and use a 1-5 Likert scale to judge the quality of the reflection.

****Rules****

You need to score the three components of the reflection separately: reasoning, cause, and suggestion. The scoring criteria are:

- 1: Fails to meet the basic requirements of the component; the content is severely incomplete or cannot support an effective reflection. Specifically, `reasoning` should include an analysis of the code behavior and error localization; `cause` should summarize the root cause of the error; `suggestion` should provide a concrete repair suggestion.
- 2: The content is formally complete, but contains obvious errors, such as fabricated facts, misunderstanding the error information, or being significantly inconsistent with the oracle.
- 3: The main direction is correct and partially consistent with the oracle, but there are clear shortcomings, such as missing key errors, insufficient explanation, or suggestions that are not actionable.
- 4: Largely correct and broadly consistent with the oracle, with only minor shortcomings that do not affect the overall judgment or repair direction.
- 5: Accurately and completely covers the key content of the oracle; any additional points must also be reasonable, supported by the context, and genuinely insightful.

****Task context****

* problem description:
\$problem_description

* error code:
\$error_code

* error information (only one error shown even if multiple exist):
- error type: \$error_type
- error message: \$feedback
- failed test case: \$test

* oracle reflection:
\$oracle_content

* reflection to be judged:
\$model_content

****Your response****

Give the scores for the three components separately, only give an integer, no explanation needed. All content must be wrapped with the special tokens, with no additional text.

```
<reasoning>score1</end>  
<cause>score2</end>  
<suggestion>score3</end>
```

Here is an output example:

```
<reasoning>4</end>  
<cause>5</end>  
<suggestion>3</end>
```

Figure 15: LLM-as-a-judge Prompt.