CollabCoder: Plan-Code Co-Evolution via Collaborative Decision-Making for Efficient Code Generation

Anonymous ACL submission

Abstract

Automated code generation remains a persistent challenge in software engineering, as conventional multi-agent frameworks are often constrained by static planning, isolated execution, high computational overhead, and limited adaptability to complex tasks. This paper introduces CollabCoder[1], a novel Plan–Code Co-Evolution framework that improves code generation through dynamic multi-agent collaboration. The core idea is to design a collaborative decision-making process between the plan module and the code module to decide which module should be executed for the debugging process. Extensive experiments on widely used benchmarks demonstrate that CollabCoder consistently improves code quality and robustness across tasks. Importantly, CollabCoder achieves performance comparable to or exceeding current state-of-the-art methods while reducing computational overhead, with efficiency gains becoming more pronounced as benchmark difficulty increases. On the more challenging LiveCodeBench and xCodeEval benchmarks, our approach improves performance by 11-20% over strong baselines while reducing the number of API calls by an average of 4-10 per execution.

1 Introduction

Code generation (known as program synthesis), a long-standing challenge in computer science, involves automatically generating programs from natural language requirements. The rapid growth of large language models (LLMs) has enabled fully executable code generation without human intervention (Nijkamp et al., 2023; Lyu et al., 2025; Chen et al., 2021). However, generating correct code for complex requirements poses challenges, particularly in advanced programming tasks (Liu et al., 2023; Dong et al., 2024). Early approaches to

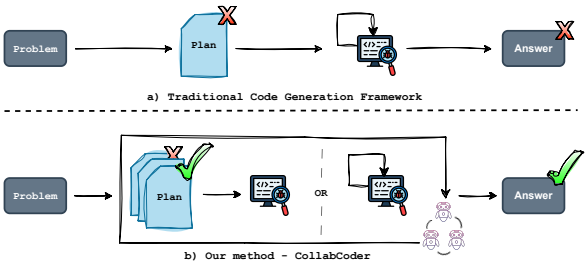

Figure 1: Overview of (a) a representative traditional code generation and (b) the proposed CollabCoder framework. Unlike conventional methods that rely on a fixed plan throughout code generation, CollabCoder allows the plan to be revised during execution. Multiple agents collaboratively assess intermediate outcomes and determine whether plan or code updates are required, enabling iterative refinement toward the final result.

LLM-based code generation primarily relied on direct prompting, where models such as Codex (Chen et al., 2021) generated code from natural language descriptions and input-output examples. More recent strategies have introduced structured prompting techniques like Chain-of-Thought reasoning (Wei et al., 2022), and retrieval-augmented generation (Parvez et al., 2021a), which guide the model using similar prior problems and solutions. Despite these improvements, performance on complex code generation tasks remains limited, as generated outputs frequently fail to pass test cases and lack integrated bug-fixing mechanisms. To address this, the *plan-before-code* paradigm (Jiang et al., 2024) has been proposed to separate high-level intent modeling from code synthesis. This idea has been incorporated into modern state-of-the-art systems, which typically adopt a dual-pass paradigm: In the first pass, a plan is generated and used to produce initial code with LLMs, while the second pass focuses on refining or debugging the resulting output (Figure 1(a)). However, refinement is often superficial, driven by simple score-based retries that do not address root causes of failure. To over-

[1]The source code is publicly available at https://anonymous.4open.science/r/CollabCoder.

come these limitations, recent works have proposed agent-based frameworks (Islam et al., 2024, 2025), which decompose the generation process into modular components for retrieval, planning, and debugging in an iterative manner for improving the code through iterations. While promising, these systems still face two fundamental challenges: (i) debugging remains largely reactive, lacking contextual learning and explicit error attribution, which results in repetitive, ineffective updates and an inability to learn from past mistakes; and (ii) the planning component is typically static throughout the debugging iterations, failing to evolve alongside the code and thereby preventing dynamic co-adaptation across stages of the generation pipeline.

Recognizing these limitations, we introduce CollabCoder, a novel Plan-Code Co-Evolution framework that addresses these challenges through multi-agent collaboration, adaptive feedback, and experience-driven learning, as illustrated in Figure 1(b). Specifically, the main contributions of this study to agent-based code generation are threefold as follows: (i) CollabCoder enables continuous improvement through a co-evolutionary process between planning and code generation. It leverages a multi-agent collaboration framework, termed Collaborative Decision-Making, in which planning, coding, and debugging agents work synergistically to decide whether to update the plan or refine the code at each iteration; (ii) To guide plan- or code-level updates, CollabCoder performs fine-grained analysis of multiple elements, including the plan, the generated code, and their alignment, rather than relying on superficial error-log inspection as in prior methods. These analyses are incorporated into a Reasoning Trajectory module, which integrates diagnostics from the current iteration with historical debugging strategies to iteratively improve both the plan and the code; and (iii) We conduct comprehensive evaluations of CollabCoder across a diverse set of benchmarks, ranging from simpler datasets (i.e., HumanEval and MBPP) to more challenging competitive programming benchmarks (i.e., xCodeEval and LiveCodeBench). The results demonstrate consistent improvements in terms of both accuracy and efficiency.

2 Related Work

2.1 Code Generation Tasks

Code generation has long been recognized as a fundamental challenge in software engineering (Niu et al., 2022). Traditional approaches primarily rely on training neural networks with task-specific annotated data methods, such as semantic parsing (Rabinovich et al., 2017) and retrieval-based techniques (Parvez et al., 2021b). More recently, several studies have leveraged pretrained language models (LMs) through continual training on both programming languages (PL) and natural languages (NL), enabling support for a variety of downstream NL-PL tasks, such as natural language code search and automatic code documentation generation (Feng et al., 2020; Wang et al., 2021). Nonetheless, the limited capacity of backbone language models restricts their utility in practical applications.

2.2 LLM for Code Generation

With their growing success, LLMs have demonstrated remarkable capabilities in code generation, driven by scaling model parameters to billions and training on extensive, diverse corpora with varied learning objectives. Recently, state-of-the-art closed-source models, such as GPT-4.1 and Claude 3.7 Sonnet, have emerged as powerful AI coding assistants. In parallel, open-source models like StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), DeepSeekCoder (Guo et al., 2024), and Qwen2.5-Coder (Hui et al., 2024) have achieved significant breakthroughs, surpassing traditional methods across numerous benchmarks. Despite these advances, modern foundational LLMs for code still lack execution awareness, struggle to distinguish reasoning errors from implementation bugs, operate in a largely stateless manner without learning from past failures. As a result, developers must handle all testing and debugging themselves.

2.3 LLM Agents for Code Generation

To overcome the limitations of standalone LLMs, recent work has proposed LLM agents that integrate planning, execution, and debugging into iterative workflows, such as MapCoder (Islam et al., 2024), CodeSIM (Islam et al., 2025), CodeAgent (Zhang et al., 2024), and ThinkCoder (Zhang et al., 2025). The core motivation behind these approaches is to enable LLMs to behave more like human developers within a digital environment by decomposing programming into multiple stages and iteratively refining solutions based on feedback. While agent-based frameworks significantly improve robustness over single-pass generation, most existing approaches still follow

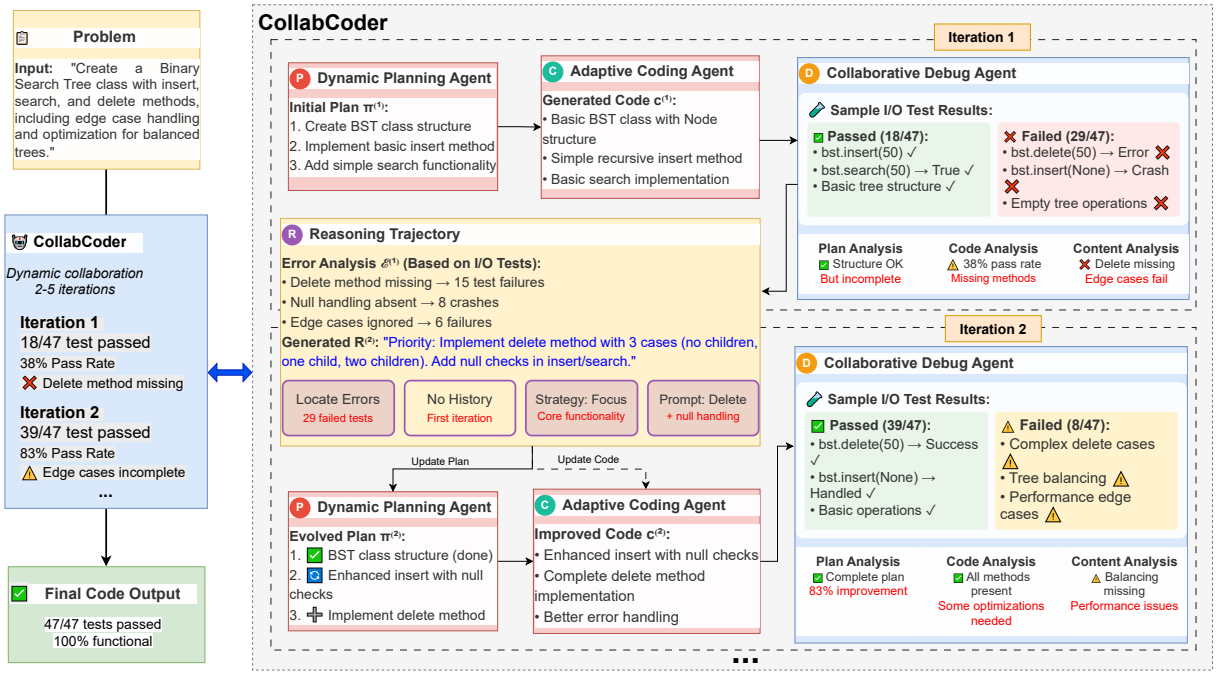

Figure 2: Architecture of the CollabCoder Framework. This diagram illustrates the co-evolutionary dynamics of the Dynamic Planning Agent and Adaptive Coding Agent, integrated with the Collaborative Debug Agent and Reasoning Trajectory Module, highlighting their continuous feedback loops and self-improving debugging mechanisms that enable adaptive plan evolution and code generation, overcoming the limitations of static planning in prior works.

a rigid trial-and-error paradigm: Planning, coding, and debugging are executed in fixed sequences with limited adaptability, and execution feedback is primarily used to repair code rather than revise high-level reasoning. As a result, these systems lack explicit mechanisms to decide whether failures should be addressed at the plan level or the implementation level, often requiring extensive exploration over multiple plans or repeated code refinements.

3 CollabCoder Framework

3.1 Overall Architecture

As illustrated in the Figure 2, CollabCoder operates in an iterative, multi-agent manner and consists of three interacting agents, following the standard agentic paradigm commonly adopted in prior code generation systems (Islam et al., 2024, 2025): A planning agent $\mathcal{A}_{\text{plan}}$, a coding agent $\mathcal{A}_{\text{code}}$, and a debugging agent $\mathcal{A}_{\text{debug}}$:

$$\text{CollabCoder} = \langle \mathcal{A}_{\text{plan}}, \mathcal{A}_{\text{code}}, \mathcal{A}_{\text{debug}} \rangle. \quad (1)$$

In our framework, the debug agent is designed as a **Collaborative Decision-Making (CDM)** module, to determine whether the error should be addressed by updating the plan or the code, followed by a **Reasoning Trajectory (RT)** module to produce an updated debugging strategy in a learn-from-mistakes, self-improving manner:

$$\mathcal{A}_{\text{debug}} = \langle \mathcal{A}_{\text{CDM}}, \mathcal{A}_{\text{RT}} \rangle. \quad (2)$$

This strategy guides plan or code refinement in the next iteration. The process repeats until the code satisfies all test cases or the iteration budget is exhausted. By combining CDM-driven adaptive decisions with RT-based self-improvement, CollabCoder enables dynamic co-evolution between planning and coding, avoiding the rigidity of static or fixed-planning approaches.

3.2 Methodology

Given a coding problem defined by a natural-language problem description P, a set of coding templates $\mathcal{T}$ that guide the LLM toward generating code compatible with an evaluation oracle $\mathcal{O}$, and a collection of Q test cases $\{(x_i, y_i)\}_{i=1}^{Q}$. At iteration t, CollabCoder maintains a solution plan $\pi^{(t)}$ and an executable program $c^{(t)}$. The program $c^{(t)}$ is executed on the test cases via $\mathcal{O}$ to produce observed outputs $\hat{y}_i$. All failing cases where $\hat{y}_i \neq y_i$ are aggregated into a test log $\mathcal{F}^{(t)}$.

3.2.1 Collaborative Decision-Making

Based on the information maintained at iteration t, the CDM agent $\mathcal{A}_{\text{CDM}}$ operates in two main phases: (i) an analysis phase, where the system examines the current state from multiple complementary perspectives; and (ii) a decision phase, where some analyses are aggregated to determine whether to update the high-level plan or refine the generated code. During the analysis phase, $\mathcal{A}_{\text{CDM}}$ performs three complementary analyses, namely, plan-level analysis $\mathcal{E}_{\pi}^{(t)}$, code-level analysis $\mathcal{E}_{c}^{(t)}$, and plan-code alignment analysis $\mathcal{E}_{align}^{(t)}$:

$$\mathcal{E}_{\pi}^{(t)}, \mathcal{E}_{c}^{(t)}, \mathcal{E}_{align}^{(t)} = \mathcal{A}_{\text{CDM}}\big(P, \mathcal{F}^{(t)}, \pi^{(t)}, c^{(t)}\big), \tag{3}$$

where the *plan-level analysis* $\mathcal{E}_{\pi}^{(t)}$ assesses whether the algorithmic reasoning encoded in $\pi^{(t)}$ is consistent with the observed failures, and identifies the underlying causes of plan-level errors. The *code-level analysis* $\mathcal{E}_{c}^{(t)}$ focuses on diagnosing implementation errors in $c^{(t)}$ under the assumption that the plan itself is correct. The *plan-code alignment analysis* $\mathcal{E}_{align}^{(t)}$ evaluates the semantic consistency between the plan and its realization in code, capturing cases where a correct plan fails due to incorrect or incomplete implementation. Sequentially, these analyses are jointly used to determine a decision $D^{(t)} \in \mathcal{D} = \{0, 1\}$, where $D^{(t)} = 0$ indicates updating the plan and $D^{(t)} = 1$ indicates updating the code.

More concretely, the refinement decision at iteration t is obtained by aggregating the three analysis signals $\mathcal{E}_{\pi}^{(t)}$, $\mathcal{E}_{c}^{(t)}$, and $\mathcal{E}_{\text{align}}^{(t)}$ through a consensus function $\mathcal{F}_{\text{cons}}$. This function is parameterized by a set of inter-module trust weights $\mathcal{W}_{\text{trust}} = \{w_{\pi}, w_{c}, w_{\text{align}}\}$, which are fixed hyperparameters shared across tasks to ensure stable and consistent decision behavior. Each weight $w_i \geq 0$ reflects the relative reliability of the corresponding analysis module $i \in \mathcal{H} = \{\pi, c, \text{align}\}$, with the normalization constraint $\sum_i w_i = 1$. The resulting collaborative decision $D^{(t)} \in \mathcal{D}$ is given by:

$$D^{(t)} = \mathcal{F}_{\text{cons}}\Big(\mathcal{E}_{\pi}^{(t)}, \mathcal{E}_{c}^{(t)}, \mathcal{E}_{align}^{(t)}, \mathcal{W}_{\text{trust}}\Big). \tag{4}$$

Specifically, the decision at iteration t is obtained by maximizing an aggregated score over the decision space $\mathcal{D}$, which jointly accounts for individual analysis confidence and cross-analysis consistency:

$$D^{(t)} = \arg\max_{d \in \mathcal{D}} \sum_{i \in \mathcal{H}} w_i \cdot \varphi_{i,d}^{(t)} \cdot \phi_{\mathcal{H}\setminus\{i\},d}^{(t)}. \tag{5}$$

Here, $\varphi_{i,d}^{(t)} \in [0, 1]$ denotes the confidence score indicating how strongly analysis i supports decision d, while $\phi_{\mathcal{H}\setminus\{i\},d}^{(t)} \in [0, 1]$ measures the consistency of decision d with the remaining analyses. All confidence and consistency scores $\{\varphi_{i,d}^{(t)}, \phi_{\mathcal{H}\setminus\{i\},d}^{(t)}\}$ are jointly produced in a single LLM invocation, conditioned on the set of analyses $\{\mathcal{E}_{\pi}^{(t)}, \mathcal{E}_{c}^{(t)}, \mathcal{E}_{\text{align}}^{(t)}\}$.

3.2.2 Reasoning Trajectory Module

The RT module enables iterative self-improvement by maintaining a persistent debugging strategy across iterations. Unlike prior approaches that treat each failure independently, RT explicitly accumulates historical diagnostic information and leverages it to guide subsequent refinements in a learn-from-mistakes manner. At iteration t, RT maintains a reasoning state $R^{(t)}$ that summarizes prior debugging insights and refinement patterns. This state is updated by jointly considering historical context and current diagnostic signals. Formally, the update rule is defined as:

$$R^{(t)} = \mathcal{A}_{\text{RT}}\Big(R^{(t-1)}, \mathcal{E}_{X}^{(t)}, P, X^{(t)}, \mathcal{F}^{(t)}\Big), \tag{6}$$

where we define $X^{(t)}$ as a unified refinement state to denote the solution component selected for refinement at iteration t and $\mathcal{E}_{X}^{(t)}$ denotes the diagnostic analysis corresponding to the current refinement target $X^{(t)}$. Specifically,

$$X^{(t)} \triangleq \mathbb{I}[D^{(t)} = 0] \cdot \pi^{(t)} + \mathbb{I}[D^{(t)} = 1] \cdot c^{(t)}. \tag{7}$$

The updated reasoning strategy $R^{(t)}$ is subsequently used to guide the refinement operator in the next iteration, influencing how the selected plan or code component is revised. Concretely, the next refinement state is obtained by applying the decision-conditioned refinement operator to the current state and the updated debugging strategy:

$$X^{(t+1)} = \mathcal{A}_{X}\Big(X^{(t)}, P, R^{(t)}, \mathcal{T}^{(t)}, \mathcal{F}^{(t)}\Big), \tag{8}$$

where $\mathcal{A}_{X} \in \{\mathcal{A}_{\text{plan}}, \mathcal{A}_{\text{code}}\}$ is selected according to the collaborative decision $D^{(t)}$. The coding template $\mathcal{T}^{(t)}$ is provided only when code refinement is selected, i.e., $\mathcal{T}^{(t)} \triangleq \mathbb{I}[D^{(t)} = 1] \cdot \mathcal{T}$, ensuring that structural constraints are enforced during code updates while plan refinement remains template-independent.

Technically, by jointly conditioning on multiple sources of information, namely, historical debugging strategies captured in $R^{(t-1)}$, localized error

diagnoses from the current iteration $\mathcal{E}_X^{(t)}$, the original problem specification P, the current plan or code $X^{(t)}$, and concrete failure evidence from $\mathcal{F}^{(t)}$, the RT module produces a structured reasoning strategy that highlights recurring error patterns, refines corrective heuristics, and avoids repeating ineffective fixes observed in previous iterations. Moreover, by explicitly modeling debugging as a stateful reasoning process, RT improves convergence stability and reduces redundant trial-and-error behaviors commonly observed in stateless debugging approaches.

4 Experiment

4.1 Datasets

We evaluate CollabCoder on two widely used benchmark datasets for code generation, HumanEval (HE) (Chen et al., 2021) and MBPP (Austin et al., 2021), along with their extended versions enriched with additional test cases, HumanEval-ET (HE-ET) and MBPP-ET, respectively (Dong et al., 2025). For evaluating performance on complex, contest-level problems, we utilize LiveCodeBench (Jain et al., 2025) and xCodeEval (Khan et al., 2024), recently established benchmarks for assessing LLM-based code generation.

4.2 Baselines and Experimental Setting

We compare CollabCoder with direct LLM-based baselines, including Chain-of-Thought (CoT) prompting (Yang et al., 2024) and Self-Planning (Jiang et al., 2024), as well as recent agent-based frameworks such as MapCoder (Islam et al., 2024), CodeSIM (Islam et al., 2025), and ThinkCoder (Zhang et al., 2025). All methods are evaluated under identical backbone LLM settings, covering both proprietary models (GPT-4o mini) and open-source models (Seed-Coder-8B and Qwen2.5-Coder-32B). We report effectiveness in terms of zero-shot Pass@1 accuracy on HumanEval, MBPP, and their extended variants (HumanEvalET and MBPP-ET), along with efficiency metrics including average token consumption and the number of API calls per problem. For agent-based baselines, including MapCoder, CodeSIM, and ThinkCoder, we use a fixed exploration budget of $k = 5$ planning iterations and a refinement budget of $n = 5$ debugging iterations for MapCoder and CodeSIM, while setting $k = 1$ and $n = 20$ for ThinkCoder, following the best configurations reported in their respective papers. For CollabCoder, we use $t = 5$ iterations, matching the same budget convention adopted by the baselines with $k = 1$ and $n = 5$. We fix the trust weights of CollabCoder's Collaborative Decision-Making module to $w_c = 0.4$, $w_\pi = 0.3$, and $w_{\text{align}} = 0.3$.

4.3 Main Results

4.3.1 Performance on basic code generation

The results in Table 1 highlight a clear trade-off between effectiveness and efficiency among different approaches. Lightweight methods such as Direct Prompting, CoT, Self-Planning, and ThinkCoder generally consume fewer LLM resources but yield limited performance gains. For instance, on Seed-Coder-8B, Direct Prompting achieves an average accuracy of only 33.30 despite minimal token usage (175.27 / 289.66), while ThinkCoder reaches 71.08 with moderate efficiency cost. More agentic frameworks, such as MapCoder and CodeSIM, substantially improve accuracy by incorporating multi-stage planning, simulation, and iterative debugging; however, these gains come at the cost of significantly higher computational overhead. For instance, on Seed-Coder-8B, MapCoder consumes 6323.28 / 3022.59 tokens with 9.84 API calls per problem, while CodeSIM requires 4154.30 / 3943.08 tokens and 6.69 API calls to achieve an average accuracy of 75.51. This overhead can be attributed to their effective inference complexity, which grows on the order of $O(nk)$. When an initial plan is misaligned, these methods may repeatedly expend API calls debugging code derived from fundamentally flawed plans, leading to substantial redundant computation.

To address this limitation, CollabCoder is designed to preserve the strengths of agentic reasoning while reducing unnecessary computation. By adopting a plan-code co-evolution strategy, CollabCoder jointly refines high-level plans and low-level code within a single evolving trajectory, thereby reducing the effective inference complexity to depend only on the number of refinement iterations. As a result, CollabCoder achieves comparable or superior accuracy with substantially lower resource consumption. Across Seed-Coder-8B, Qwen2.5-Coder-32B, and GPT-4o mini, it reduces total token usage by approximately 30-50% compared to MapCoder and by 10-25% compared to CodeSIM, while consistently attaining higher average accuracy. Overall, these results demonstrate that Collab-

	Accuracy ↑					Efficiency ↓			
Method	**HE**	**HE-ET**	**MBPP**	**MBPP-ET**	**Average**	**k**	**n**	**Token I/O**	**API calls**
Backbone LLM: Seed-Coder-8B									
Direct	18.90	17.07	59.19	38.03	33.30	–	–	175.27 / 289.66	1.00
CoT	82.32	75.00	75.06	50.13	70.63	–	–	1087.79 / 182.70	1.00
Self-Planning	82.32	71.34	74.06	51.13	69.71	–	–	2154.48 / 1018.15	2.00
MapCoder	79.88	70.12	73.55	49.12	68.78	5	5	6323.28 / 3022.59	9.84
CodeSIM	**90.24**	76.20	82.00	53.65	75.51	5	5	4154.30 / 3943.08	6.69
ThinkCoder	82.32	73.78	76.83	51.39	71.08	1	20	1613.02 / 1544.64	4.56
CollabCoder (Ours)	87.20	**78.05**	**83.37**	**56.42**	**76.26**	1	5	**4219.78 / 1964.03**	**5.06**
Backbone LLM: Qwen2.5-Coder-32B									
Direct	85.37	75.61	79.09	54.91	75.00	–	–	128.93 / 365.61	1.00
CoT	90.24	81.70	83.38	59.44	79.15	–	–	948.07 / 89.79	1.00
Self-Planning	87.80	76.83	77.33	53.15	74.70	–	–	1635.33 / 0640.73	2.00
MapCoder	90.24	79.00	86.80	**59.95**	79.84	5	5	5848.39 / 3309.55	9.05
CodeSIM	93.29	81.70	87.20	58.70	80.22	5	5	2191.03 / 2593.04	4.87
ThinkCoder	88.41	79.88	85.89	53.90	77.02	1	20	1128.67 / 1404.25	2.99
CollabCoder (Ours)	**95.73**	**84.15**	**90.17**	**59.95**	**82.50**	1	5	**2468.22 / 1606.88**	**4.12**
Backbone LLM: GPT-4o mini									
Direct	85.97	76.22	75.82	52.14	72.54	–	–	105.98 / 396.46	1.00
CoT	85.97	78.66	78.59	54.66	74.47	–	–	946.50 / 129.60	1.00
Self-Planning	82.32	74.39	78.84	53.65	72.30	–	–	1716.58 / 854.74	2.00
MapCoder	90.24	79.88	84.13	56.93	77.80	5	5	5767.54 / 2965.21	10.10
CodeSIM	94.51	81.70	89.92	59.95	81.52	5	5	2397.89 / 2688.32	5.16
ThinkCoder	90.85	81.70	81.61	55.92	77.52	1	20	1007.58 / 1172.93	2.20
CollabCoder (Ours)	**96.34**	**84.76**	**91.69**	**60.20**	**83.25**	1	5	**2993.33 / 1781.21**	**5.06**

Table 1: Accuracy and efficiency comparison across multiple benchmark datasets and backbone LLMs. Accuracy is measured by Pass@1, while efficiency is assessed using Token In/Out and the number of API calls. Token In/Out is averaged across all API calls and datasets, with its detailed formulation provided in Appendix B.3. In the *Accuracy* section, boldface indicates the best-performing method for each backbone, and underlined values denote the second-best. In the *Efficiency* section, boldface highlights the results of our method.

Method	**LiveCodeBench**	**xCodeEval**	**Average**	**Token I/O**	**API Calls**
MapCoder	34.82	40.57	37.70	28437.65 / 17692.18	22.41
CodeSIM	36.61	42.45	39.53	20907.82 / 13151.10	17.16
CollabCoder (Ours)	**41.96**	**47.16**	**44.56**	**15155.93 / 4491.37**	**12.27**

Table 2: Pass@1 accuracy and efficiency comparison on contest-level code generation benchmarks. Token I/O denotes the average number of input and output tokens per problem, and API Calls indicate the average number of model invocations. All methods are evaluated using GPT-4o mini.

Coder provides a more favorable balance between effectiveness and efficiency than existing baselines.

4.3.2 Complex Code Generation

We further evaluate the proposed method on complex, contest-level code generation tasks to assess its capability in solving programming problems that closely resemble real-world competitive coding scenarios. To this end, we adopt two widely used benchmarks, LiveCodeBench and xCodeEval, which are specifically designed to evaluate the robustness of code generation systems across different difficulty levels. In this setting, we compare CollabCoder against MapCoder and CodeSIM, two state-of-the-art agentic approaches for complex competitive programming.

For fairness and consistency, all methods are evaluated using GPT-4o mini. We exclude Seed-Coder-8B and Qwen2.5-Coder-32B from this evaluation, as preliminary experiments indicate that their limited model capacity leads to uniformly low performance on contest-level benchmarks, thereby obscuring meaningful performance differences. Table 2 summarizes the Pass@1 accuracy and efficiency of different methods on these competitive programming benchmarks. Accordingly, compared to basic code generation settings, the advantages of CollabCoder become more pronounced on complex, contest-level code generation tasks. As shown in Table 2, CollabCoder achieves a Pass@1 accuracy of 41.96% on LiveCodeBench and 47.16% on xCodeEval, outperforming MapCoder by approx-

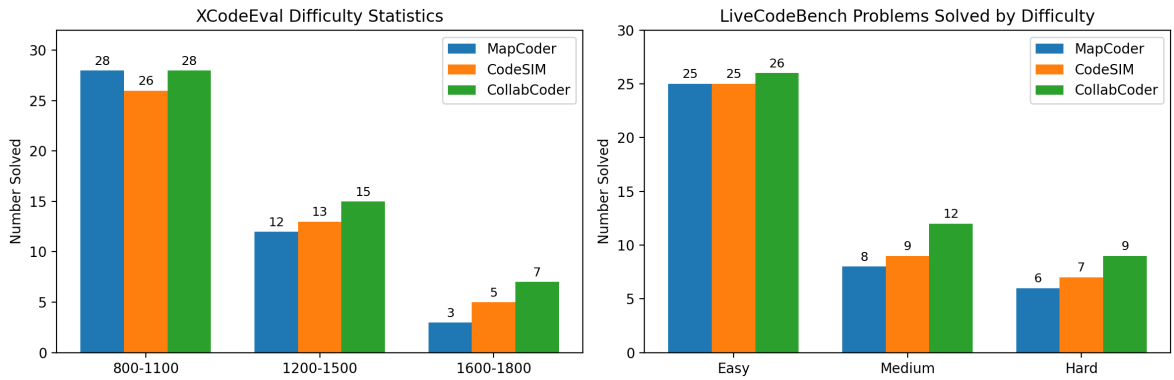

Figure 3: Distribution of solved competitive programming problems across different difficulty levels, illustrating the proportion of problems completed at each level.

imately 6.6-7.1 percentage points and CodeSIM by 4.7-5.3 points across the two benchmarks. In addition to accuracy gains, CollabCoder exhibits a clear advantage in computational efficiency, reducing total token consumption by approximately 57% compared to MapCoder and 42% compared to CodeSIM. These consistent improvements demonstrate CollabCoder's strong effectiveness and efficiency on especially challenging tasks.

Furthermore, to gain a finer-grained perspective on model performance, we further examine the distribution of solved problems across difficulty levels (Figure 3). On xCodeEval, CollabCoder achieves performance comparable to MapCoder in the easiest difficulty range (800–1100), with both methods solving 28 problems, while CodeSIM solves 26. As task difficulty increases, CollabCoder tends to maintain more stable performance. In the 1200–1500 range, CollabCoder solves 15 problems, compared to 12 for MapCoder and 13 for CodeSIM. In the hardest range (1600–1800), CollabCoder solves 7 problems, whereas MapCoder and CodeSIM solve 3 and 5 problems, respectively. These results suggest that CollabCoder experiences a milder performance degradation as problem difficulty increases, with the differences becoming more noticeable in the medium and hard difficulty ranges. A similar trend is observed on LiveCodeBench. For medium-difficulty problems, CollabCoder solves 12 tasks, compared to 8 for MapCoder and 9 for CodeSIM. On hard problems, CollabCoder remains competitive with 9 solved tasks, slightly higher than MapCoder (6) and CodeSIM (7). Overall, these results indicate that CollabCoder maintains relatively stable effectiveness across different difficulty levels and demonstrates better adaptability when handling more complex, contest-level programming tasks.

4.4 Self-improving Debugging Analysis

Figure 4 illustrates the accuracy-budget tradeoff of different debugging strategies on the LiveCodeBench benchmark. By examining this relationship on a competitive benchmark where iterative debugging plays a critical role, we can better understand how each method evolves code quality under constraints on the number of API calls. On LiveCodeBench, CollabCoder demonstrates a

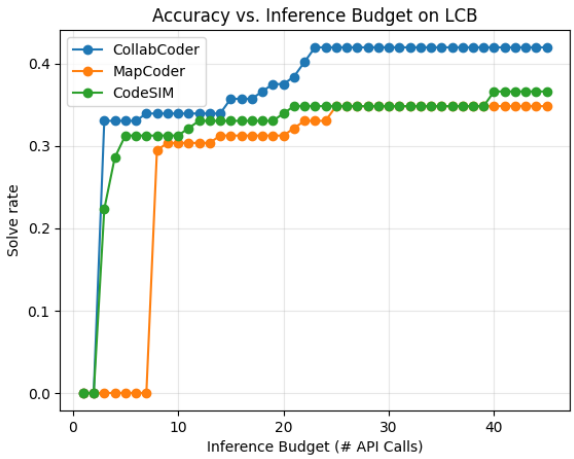

Figure 4: Accuracy vs. Inference budget on LiveCodeBench benchmark.

clear advantage in the low-budget regime. With an inference budget of only 10 API calls, CollabCoder achieves a solve rate of 33.93%, outperforming both MapCoder (30.36%) and CodeSIM (31.25%). This result indicates that, under identical budget constraints, CollabCoder more effectively integrates high-level reasoning signals through accumulated experience and structured analysis in its Reasoning Trajectory module. As a result, it is

able to translate limited feedback into targeted and meaningful improvements, rather than expending inference budget on largely trial-and-error debugging. In contrast, MapCoder and CodeSIM exhibit less efficient budget utilization, requiring additional API calls to achieve comparable accuracy gains.

4.5 Collaborative Decision-Making Analysis

To better understand the decision-making behavior of the CDM module, we analyze its behavior across different backbone models and datasets with varying levels of difficulty. An intuitive visualization is provided in Figure 5 and more detailed statistics are reported in Table 4 (Appendix B.6).

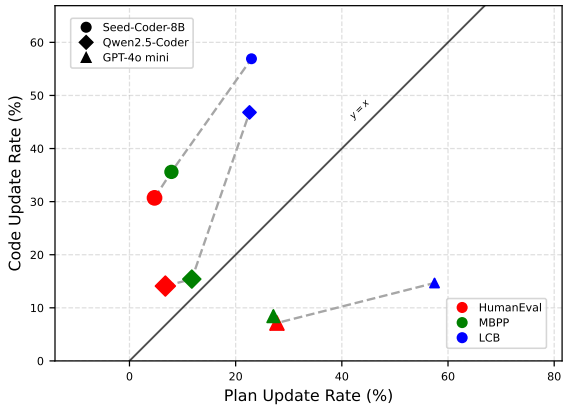

Figure 5: Relationship between *Plan Update Rate* and *Code Update Rate* across different backbone LLMs and datasets. Marker shapes denote backbone models, colors indicate datasets, and marker sizes are proportional to accuracy on the corresponding dataset. The update rate is a normalized metric that allows comparison across different datasets, defined as the ratio of the number of updates to the total number of iterations; further details are provided in Appendix B.3.

Effect of Dataset Difficulty. Across all backbones, the frequency of CDM-triggered updates consistently increases as dataset difficulty rises. As a result, a common trend across all three backbones is that the update rate increases (at both the code and plan levels) as the datasets become more challenging. This monotonic trend indicates that harder benchmarks introduce a higher incidence of failure cases, thereby requiring CDM to intervene more frequently. Such behavior suggests that CDM effectively adapts its intervention rate to task complexity, providing additional corrective signals when the problem space becomes harder to navigate.

Backbone-Specific Update Dynamics. Despite the shared trend described above, the allocation between plan-level and code-level updates differs markedly across backbones. Code-specialized models predominantly rely on code-level revisions, with plan update rates approximately two to three times lower than code update rates. In contrast, the general-purpose GPT-4o mini allocates a substantially larger fraction of its CDM interventions to plan-level revisions. As shown in Figure 5, the diagonal line $y = x$ clearly separates data points corresponding to these two classes of models.

Optimization Perspective. We interpret this divergence from an optimization perspective. Code-specialized models tend to operate within a narrow neighborhood of the initial plan, applying incremental code-level modifications. When the initial plan is suboptimal, such localized adjustments are prone to converging to local minima, where repeated implementation-level fixes fail to correct a flawed high-level strategy. In contrast, strong general-purpose models demonstrate a higher sensitivity to diagnosing failures that originate from inappropriate high-level strategies rather than isolated coding errors. In these cases, CDM more frequently selects plan-level updates, enabling the model to escape local minima by exploring alternative regions of the solution space.

5 Conclusion and Future Work

This study presented CollabCoder, a collaborative multi-agent framework that addresses fundamental limitations in existing code generation systems by introducing dynamic planning, adaptive coding, and self-improving debugging. Unlike conventional static pipelines, CollabCoder enables continuous co-evolution between plans and code through a collaborative analysis engine that synthesizes insights across planning, coding, and debugging. Experimental results across standard, extended, and contest-level benchmarks confirm that CollabCoder achieves higher accuracy, robustness, and adaptability, while maintaining computational efficiency. The framework establishes a new paradigm for automated programming, where coordinated agent collaboration surpasses isolated strategies. For future work, we plan to extend CollabCoder by integrating formal verification techniques, exploring multi-modal programming tasks, and enhancing semantic alignment between evolving plans and specifications. These directions position CollabCoder as a promising foundation for next-generation collaborative AI systems in software development.

Limitations

While CollabCoder demonstrates strong empirical performance across benchmarks, it is not without limitations. Firstly, our approach remains heavily dependent on the capability of the underlying LLM backbone. In particular, the analysis and collaborative decision-making stages require strong reasoning and code understanding, which may limit the effectiveness of CollabCoder when using weaker or resource-constrained models. Secondly, CollabCoder relies on a fixed and limited number of sample I/O pairs for debugging. While helpful for iterative refinement, their limited coverage may restrict robustness against diverse edge cases. Finally, future work will focus on improving automatic additional test case generation to enhance sample I/O quality, as well as adapting CollabCoder to lighter-weight backbone models to improve efficiency and accessibility.

References

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025. Codescore: Evaluating code generation by learning code execution. *ACM Trans. Softw. Eng. Methodol.*, 34(3):77:1–77:22.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7):189:1–189:38.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

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*, abs/2401.14196.

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

Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 4912–4944. Association for Computational Linguistics.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md. Rizwan Parvez. 2025. Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 5113–5139. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 33(7):182:1–182:30.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt

really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Michael R. Lyu, Baishakhi Ray, Abhik Roychoudhury, and Shin Hwei Tan. 2025. Automatic programming: Large language models and beyond. *ACM Trans. Softw. Eng. Methodol.*, 34(6):140:1–140:33.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 5546–5555. ijcai.org.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021b. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2719–2734. Association for Computational Linguistics.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1139–1149. Association for Computational Linguistics.

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, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. 2024. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Trans. Software Eng.*, 50(9):2437–2457.

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

Xiaoqing Zhang, Yuhan Liu, Flood Sung, Xiuying Chen, Shuo Shang, and Rui Yan. 2025. Thinking before running! efficient code generation with thorough exploration and optimal refinement. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 23268–23281. Association for Computational Linguistics.

A Error Analysis with Examples

To better understand the limitations of CollabCoder, we analyze cases where the framework fails or produces partially correct solutions. Our error analysis highlights recurring failure modes, including missing functionality, incomplete edge-case handling, semantic misalignment with the specification, and performance issues:

A.1 Case Study 1: Binary Search Tree (from framework example)

As illustrated in Figure 2, CollabCoder initially generated a Binary Search Tree class with insert and search methods but omitted the delete method and null handling. This led to 29 out of 47 test failures in the first iteration. The collaborative debug agent localized errors, prompting plan refinement to add delete functionality and edge-case handling. After two iterations, the solution improved significantly (83% pass rate) but still missed advanced balancing and performance optimization. This example illustrates how CollabCoder effectively reduces reliance

on superficial trial-and-error by progressively incorporating missing components. However, conceptual leaps (e.g., balanced tree optimization) remain difficult.

A.2 Case Study 2: MBPP Task 612 - Merging Sublists

"Write a function to merge the first and last elements separately in a list of lists"

Expected Behavior: The function should also generalize to cases where sublists contain more than two elements. For example:

$$[["x","y","z"],\ ["a","b","c"],\ ["m","n","o"]]$$
$$\longrightarrow$$
$$[["x","a","m"],\ ["y","b","n"],\ ["z","c","o"]]$$

Generated Output:

```
def merge(lst: List[List[Any]]) ->
    List[List[Any]]:
    if not lst:
        return [[], []]
    first_elements = []
    second_elements = []
    for sublist in lst:
        if len(sublist) != 2:
            raise ValueError("Each␣sublist␣must␣
    contain␣exactly␣two␣elements.")
        first_elements.append(sublist[0])
        second_elements.append(sublist[1])
    return [first_elements, second_elements]
```

Failure Analysis: The code worked for sublists of exactly two elements but failed on test cases where sublists contained three elements. Specifically:

- CollabCoder enforced a restrictive assumption (`len(sublist) != 2` → error), contradicting the broader specification.
- Subsequent debugging iterations failed to revise the plan for variable-length sublists.

Insight: This case reveals a key limitation: CollabCoder can overfit to an overly strict interpretation of the specification during plan formation. Moreover, its debugging mechanism struggles to generalize when the problem requires extending functionality beyond initial assumptions.

In general, while CollabCoder substantially improves over baselines in robustness and adaptability, systematic error analysis highlights the need for stronger semantic alignment between problem specification and evolving plans, particularly in tasks that require flexible generalization.

B Experimental Details

B.1 Baselines Description

Direct Prompting requires language models to generate code directly, without any additional instructions or intermediate reasoning steps, leveraging their intrinsic problem-solving abilities. CoT (Yang et al., 2024) encourages models to generate intermediate natural language reasoning to guide code generation, while Self-Planning (Jiang et al., 2024) separates the process into a planning phase and an implementation phase. MapCoder (Islam et al., 2024) adopts a multi-agent architecture covering example retrieval, planning, coding, and debugging, and CodeSIM (Islam et al., 2025) extends this framework by incorporating simulated input/output execution for iterative verification. ThinkCoder (Zhang et al., 2025) employs a streamlined two-agent design with an Exploration Agent for generating diverse solutions and a CodeVerifier Agent for independent evaluation using a test pool.

B.2 Dataset Description

For **HumanEval** and **MBPP**, we follow the standardized preprocessing protocol adopted in prior work (Islam et al., 2024) to ensure fair and reproducible evaluation. After preprocessing, the resulting benchmark consists of 164 HumanEval problems and 397 MBPP problems. Their extended counterparts, HumanEval-ET (HE-ET) and MBPP-ET, augment each original problem with additional test cases, enabling a more stringent assessment of functional correctness and robustness under out-of-distribution inputs (Dong et al., 2025). To evaluate performance on more challenging, competition-style programming tasks, we further employ **LiveCodeBench** and **xCodeEval**, which are designed to reflect real-world coding difficulty and temporal data distribution. In particular, we use the most recent release of LiveCodeBench (version 6), which contains 175 code generation problems collected between May 2023 and April 2025[2]. Following common practice for consistent evaluation, we exclude problems formulated in a functional input–output style and retain only 112 Stdin/Stdout-based tasks, which align with the execution-based evaluation protocol used in this work.

To enable execution-based evaluation on these datasets, the LLM is required to generate code that

[2] https://huggingface.co/datasets/livecodebench/code_generation_lite

strictly follows a predefined coding template compatible with the corresponding evaluation framework. This requirement ensures that the generated code can be executed automatically and evaluated consistently across different benchmarks.

HumanEval and MBPP adopt a functional coding paradigm, where the model is expected to generate a single Python function according to the function signature and problem description provided in the prompt, then the generated function is directly invoked by the evaluator.

Example coding template for HumanEval:

```
def truncate_number(number: float) -> float:
    """ Given a positive floating point number,
    ↪  it can be decomposed into
    and integer part (largest integer smaller
    ↪  than given number) and decimals
    (leftover part always smaller than 1).

    Return the decimal part of the number.
    >>> truncate_number(3.5)
    0.5
    """
```

Example coding template for MBPP:

```
from typing import List

def heap_queue_largest(nums: List[int], n:
↪  int) -> List[int]:
    """
    Write a function to find the n largest
    ↪  integers from a given list of numbers,
    returned in descending order.
    """
```

In contrast, xCodeEval and LiveCodeBench follow a stdin/stdout-based coding format. In this setting, the model is required to generate a complete program that reads inputs from standard input, processes them according to the problem specification, and writes the correct outputs to standard output. The entire program is executed as a script during evaluation, which more closely reflects real-world programming contest environments.

Example problem from xCodeEval:

```
Title: Make All Numbers Odd

Problem Description:
There are n positive integers a_1, a_2, ...,
a_n. In one move, you can choose
an even value c and divide by two all elements
equal to c. The goal is to find
the minimum number of moves required to make
all numbers odd.

Input:
The first line contains an integer t, the
number of test cases.
For each test case, the first line contains an
integer n.
The second line contains n positive integers
a_1, a_2, ..., a_n.

Output:
For each test case, output the minimum number
of moves required.

Constraints:
1 <= t <= 10^4,
1 <= n <= 2*10^5 (sum of n over all test
cases),
1 <= a_i <= 10^9.

Sample Input:
4
6
40 6 40 3 20 1
1
1024
4
2 4 8 16
3
3 1 7

Sample Output:
4
10
4
0
```

Example problem from LiveCodeBench:

```
Title: ABC400 Ceremony

Problem Description:
In the ceremony commemorating ABC400, we want
to arrange 400 people in a
rectangular formation with A rows and B
columns. Given A, determine a positive integer
B such that AB = 400. If no such B exists,
output -1.

Input:
A single integer A.

Output:
Output a single integer B, or -1 if no valid
arrangement exists.

Constraints:
1 <= A <= 400.

Sample Input:
10

Sample Output:
40
```

B.3 Metrics

Update Rate. Plan Update Rate (PU Rate) and Code Update Rate (CU Rate) measure the proportion of plan-level and code-level update decisions, respectively, made by the CDM module over the total number of executed iterations. Unlike raw

Model	w/ CDM	w/ RT	HumanEval	HumanEvalET	MBPP	MBPP-ET	Avg
Seed-Coder-8B	×	✓	85.37↓1.83	75.00↓3.05	76.07↓7.30	51.64↓4.78	72.02↓4.24
Seed-Coder-8B	✓	×	85.00↓2.20	75.61↓2.44	79.34↓4.03	51.64↓4.78	72.90↓3.36
Seed-Coder-8B	✓	✓	**87.20**	**78.05**	**83.37**	**56.42**	**76.26**
Qwen2.5-Coder-32B	×	✓	90.24↓5.49	81.10↓3.05	83.38↓6.79	56.93↓3.02	77.91↓4.59
Qwen2.5-Coder-32B	✓	×	92.68↓3.05	82.92↓1.23	88.41↓1.76	58.94↓1.01	80.74↓1.76
Qwen2.5-Coder-32B	✓	✓	**95.73**	**84.15**	**90.17**	**59.95**	**82.50**

Table 3: Ablation study on the impact of CDM and RT. We evaluate different variants of CollabCoder with and without CDM and RT on the HumanEval and MBPP benchmarks using the Pass@1 metric. *w/ CDM* and *w/ RT* indicate that the corresponding component is enabled in the pipeline.

update counts, which are often biased by the number of samples in a dataset, update rates provide a normalized measure that enables fair comparison across datasets of different sizes. Importantly, these metrics also account for iterations in which the CDM module decides not to perform any update, thereby reflecting the full decision space of the system rather than only corrective actions. The metrics on a dataset $\mathcal{S}$ are computed as follows:

$$\textbf{PU Rate} = \frac{\textbf{PU}}{\textbf{PU} + \textbf{CU} + |\mathcal{S}|}, \quad \textbf{CU Rate} = \frac{\textbf{CU}}{\textbf{PU} + \textbf{CU} + |\mathcal{S}|}. \tag{9}$$

Here, **PU** and **CU** denote the total numbers of plan-level and code-level updates, respectively, aggregated over all samples in the dataset. $|\mathcal{S}|$ represents the size of the dataset $\mathcal{S}$. The denominator thus corresponds to the total number of executed iterations across all samples in $\mathcal{S}$.

Token Input/Output. Token Input/Output (Token I/O) is defined as the combination of two complementary metrics: Token Input and Token Output. These metrics are computed as averages over the entire dataset and represent the average number of input and output tokens required to execute a single sample. The general formulation for computing Token Input and Token Output on a dataset $\mathcal{S}$ is as follows:

$$\textbf{Token Avg} = \frac{\sum_{i \in \mathcal{S}} \sum_{k=1}^{a_i} t_k}{|\mathcal{S}|}. \tag{10}$$

Here, a_i denotes the total number of API calls made during the execution of sample i, and t_k denotes the number of tokens (either input or output) consumed in the k-th API call. Overall, Token I/O serves as an objective measure of framework efficiency by capturing the average token consumption required to solve a sample.

B.4 Ablation Study

We conduct an ablation study to evaluate the contributions of the two core components of CollabCoder: The CDM module (Equation 5) and the RT module (Equation 6). To assess the impact of the CDM module, we replace it with a conventional debugging mechanism that updates the code following prior works (Islam et al., 2024, 2025). To examine the effectiveness of the RT module, we remove it and directly use the output of the CDM module to generate the next iteration. The results reported in Table 3 highlight the crucial role of both the CDM and RT modules, confirming their importance to the overall performance of CollabCoder. First, when the CDM module is replaced with a standard debugging approach, the performance drops significantly across all datasets for both base models. This suggests that CDM plays a crucial role in effectively coordinating decisions during iterative refinement, leading to more robust code updates compared to conventional debugging strategies. Second, removing the RT module also leads to a consistent performance decline, although the drop is generally less severe than removing CDM. This indicates that while CDM provides the foundation for collaborative improvement, RT further enhances solution quality by guiding the model to leverage structured reasoning paths rather than relying solely on raw outputs. Notably, the full version of CollabCoder achieves the best average performance on both base models, confirming that the two modules are complementary. For the larger Qwen2.5-Coder-32B, the benefits are more pronounced, with the full model outperforming both ablated versions by a noticeable margin, particularly on the HumanEval and MBPP benchmarks. This implies that CollabCoder scales well with stronger base models and can more effectively exploit their capacity when both CDM and RT are

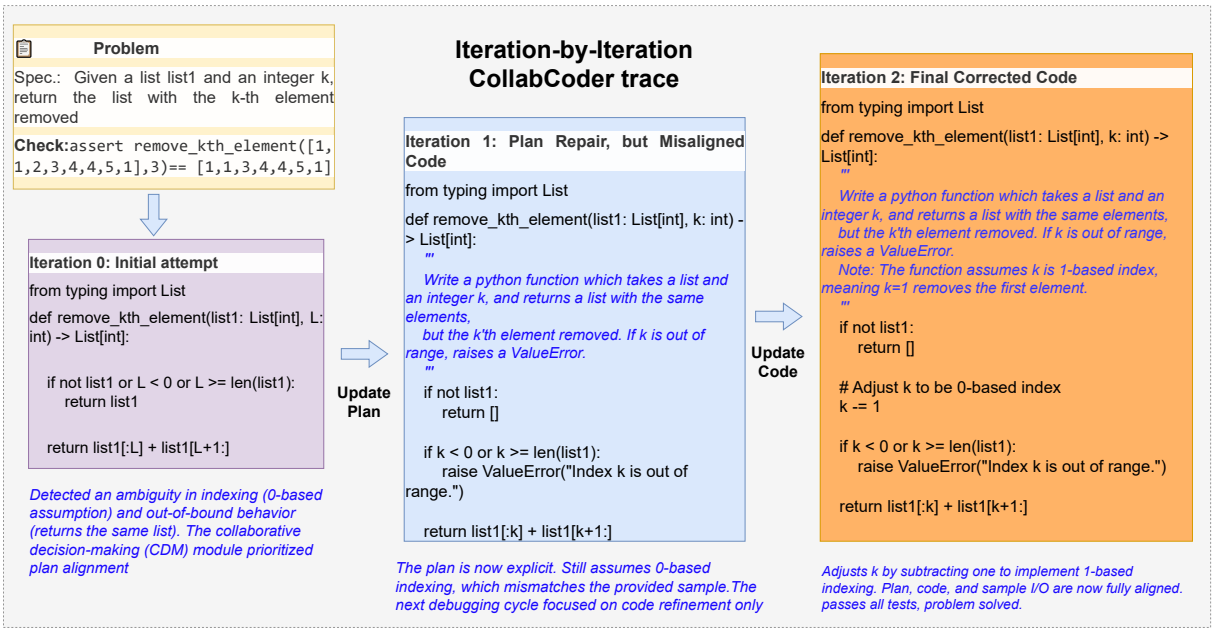

Figure 6: An example of self-improving debugging.

integrated.

B.5 Self-improving Debugging Analysis

An example of self-improving debugging is intuitively provided in Figure 6. The process shows how the CDM module first identifies a semantic mismatch (0-based indexing assumption) in the initial solution, then repairs the plan to make specification explicit (iteration 1), and finally refines the code by adjusting to 1-based indexing (iteration 2). The problem converges within two iterations, yielding a fully correct implementation aligned with both the specification and the sample I/O.

B.6 CDM Analysis

Table 4 reports detailed statistics of plan-level and code-level updates collected during the execution

Model	Dataset	Plan Update		Code Update	
		Quantity	Rate (%)	Quantity	Rate (%)
Seed-Coder-8B	HumanEval	12	04.72	78	30.71
	MBPP	55	07.90	244	35.06
	LiveCodeBench	128	22.98	317	56.91
Qwen2.5-Coder-32B	HumanEval	14	06.76	29	14.01
	MBPP	64	11.74	84	15.41
	LiveCodeBench	80	22.16	169	46.81
GPT-4o mini	HumanEval	70	27.78	18	07.14
	MBPP	167	27.11	52	08.44
	LiveCodeBench	231	57.46	59	14.68

Table 4: A detailed report of the quantity and update rate of plan and code updates for different backbone models on HumanEval, MBPP, and LiveCodeBench.

of our pipeline on three benchmarks: HumanEval, MBPP, and LiveCodeBench. To facilitate the computation of Plan Update Rate (PU Rate) and Code Update Rate (CU Rate), we further clarify the characteristics of these datasets, with particular emphasis on their sample sizes.

B.7 Additional Self-Improving Analysis on xCodeEval

Figure 7 presents an additional self-improving analysis of different debugging strategies on the xCodeEval benchmark, focusing on how solution accuracy evolves with increasing inference budgets. At a low inference budget of 10 API calls, Col-

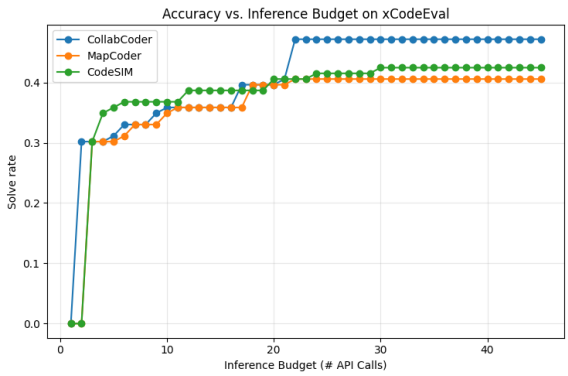

Figure 7: Accuracy vs. inference budget on xCodeEval benchmark.

labCoder attains a solve rate of 35.85%, which is comparable to MapCoder (34.91%) and CodeSIM (36.79%), indicating a relatively small early advantage on this benchmark. However, as the infer-

ence budget increases, CollabCoder exhibits a substantially steeper improvement trajectory. With approximately 20-25 API calls, CollabCoder rapidly converges to a higher solve rate of 47.16%, a performance level that MapCoder and CodeSIM do not reach even with significantly larger inference budgets.

C Implementation Details

C.1 Detailed Algorithm of CollabCoder

The detailed implementation of CollabCoder is provided as the pseudo-code in Algorithm 1.

Algorithm 1 CollabCoder pipeline

Require: Problem specification P, coding template $\mathcal{T}$, maximum number of iterations T, test set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{Q}$, trust weights $\mathcal{W} = \{w_\pi, w_c, w_{\text{align}}\}$, execution oracle $\mathcal{O}$
Ensure: Final program c

$R^{(0)} \leftarrow \emptyset$ ▷ Reasoning Trajectory
$\pi^{(0)} \leftarrow \mathcal{A}_{\text{plan}}(P)$
$c^{(0)} \leftarrow \mathcal{A}_{\text{code}}(\pi^{(0)}, \mathcal{T})$
for $t = 0$ **to** $T - 1$ **do**
  $\mathcal{F}^{(t)} \leftarrow \mathcal{O}(c^{(t)})$ ▷ Execute code
  if $\textsc{Satisfy}(\mathcal{F}^{(t)})$ **then**
    return $c^{(t)}$
  end if
  $(\mathcal{E}_\pi^{(t)}, \mathcal{E}_c^{(t)}, \mathcal{E}_{\text{align}}^{(t)}) \leftarrow \mathcal{A}_{\text{CDM}}(P, \pi^{(t)}, c^{(t)}, \mathcal{F}^{(t)})$
  $\Phi^{(t)} = \{\phi^{(t)}_{\{\pi,c,\text{align}\}\setminus\{i\},d} \mid i \in \{\pi, c, \text{align}\},\ d \in \{\text{plan}, \text{code}\}\},\ \Psi^{(t)} = \{\varphi^{(t)}_{i,d} \mid i \in \{\pi, c, \text{align}\},\ d \in \{\text{plan}, \text{code}\}\} \leftarrow \mathcal{A}_{\text{CDM}}(\mathcal{E}_\pi^{(t)}, \mathcal{E}_c^{(t)}, \mathcal{E}_{\text{align}}^{(t)})$
  $D^{(t)} \leftarrow \arg\max_{d \in \{\text{plan},\text{code}\}} \sum_{i \in \{\pi,c,\text{align}\}} w_i \cdot \phi^{(t)}_{\{\pi,c,\text{align}\}\setminus\{i\},d} \cdot \varphi^{(t)}_{i,d}$
  if $D^{(t)} = \text{plan}$ **then**
    $X^{(t)} \leftarrow \pi^{(t)}$
    $\mathcal{E}_X^{(t)} \leftarrow \mathcal{E}_\pi^{(t)}$
  else
    $X^{(t)} \leftarrow c^{(t)}$
    $\mathcal{E}_X^{(t)} \leftarrow \mathcal{E}_c^{(t)}$
  end if
  $R^{(t+1)} \leftarrow \mathcal{A}_{\text{RT}}(R^{(t)}, \mathcal{E}_X^{(t)}, P, X^{(t)}, \mathcal{F}^{(t)})$
  if $D^{(t)} = \text{plan}$ **then**
    $\pi^{(t+1)} \leftarrow \mathcal{A}_{\text{plan}}(P, \pi^{(t)}, \mathcal{F}^{(t)}, R^{(t+1)})$
    $c^{(t+1)} \leftarrow \mathcal{A}_{\text{code}}(\pi^{(t+1)}, \mathcal{T})$
  else
    $\pi^{(t+1)} \leftarrow \pi^{(t)}$
    $c^{(t+1)} \leftarrow \mathcal{A}_{\text{code}}(P, c^{(t)}, \mathcal{F}^{(t)}, R^{(t+1)}, \mathcal{T})$
  end if
end for
return $c^{(T)}$

C.2 Prompt Templates

For better reproducibility, we present all prompt templates shown in Figures 8–15 in the appendix.

INITIAL PLANNING ($\mathcal{A}_{\text{plan}}$)

Task:
Generate a detailed step-by-step plan to solve the given programming problem. The plan should describe the reasoning and algorithmic approach without generating any executable code.

- Recall a relevant but distinct example problem.
- Describe its solution approach and underlying algorithm.
- Based on this reasoning, produce a detailed plan for the original problem.
- Do **not** generate any code.

Problem:
{{problem}}
Sample Test Cases:
{{sample_io}}

Figure 8: Prompt for initial planning $\pi^{(0)} \leftarrow \mathcal{A}_{\text{plan}}(P)$ (Algorithm 1, Line 2).

INITIAL CODE GENERATION ($\mathcal{A}_{\text{code}}$)

Task:
Generate an executable program that solves the given problem by strictly following the provided plan. The code must conform to the specified programming language and input/output format.

- You are given a step-by-step plan $\pi^{(0)}$ describing how to solve the problem.
- Implement the solution strictly according to this plan.
- If available, follow the provided coding template $\mathcal{T}$ without modification.
- Follow the sample input/output format exactly.
- Do **not** add extra explanations, comments outside the code, or auxiliary text.
- Do **not** include assertion or testing statements.

IMPORTANT INSTRUCTIONS:

- The generated code must be written in {{language}}.
- The entire code must be enclosed within a triple backtick (“‘) block.
- Read input from standard input and write output to standard output.
- Do not include any extra print statements.

Problem:
{{problem}}
Plan ($\pi^{(0)}$):
{{plan}}
Sample Test Cases:
{{sample_io}}

Figure 9: Prompt for initial code generation $c^{(0)} \leftarrow \mathcal{A}_{\text{code}}(\pi^{(0)}, \mathcal{T})$ (Algorithm 1, Line 3).

MERGED DIAGNOSTIC ANALYSIS ($\mathcal{A}_{\text{CDM}}$)

Task:
Perform a comprehensive diagnostic analysis of the current plan, code, and their alignment with the problem, based on observed execution failures. The goal is to identify errors, inconsistencies, and misalignments without proposing new solutions.

Context Provided:

- The original problem description P.
- The current plan $\pi^{(t)}$.
- The current code implementation $c^{(t)}$.
- The failure log $\mathcal{F}^{(t)}$, which records incorrect behavior on test cases.

Response Structure (Strict):

- **Plan Analysis**
  - **Simulation:** Step-by-step simulation of the plan on the failing test cases.
  - **Insight:** Determine whether the plan is incorrect, or whether errors arise from plan-to-code translation, and explain how the plan should be corrected.
- **Code Analysis**
  - **Simulation:** Line-by-line execution of the code on the failing test cases.
  - **Insight:** Identify implementation bugs or logical errors and explain how they should be fixed.
- **Content Analysis**
  - Provide a *single concise insight* (4–5 sentences) evaluating the alignment between the problem, plan, and code.
  - Conclude which component(s) should be updated (plan, code, both).

IMPORTANT:

- The failure log is always correct and must not be questioned.
- Do **not** generate new plans or code.
- Do **not** introduce alternative solutions.
- Strictly follow the specified structure.

Problem (P):
{{problem}}
Current Plan ($\pi^{(t)}$):
{{plan}}
Current Code ($c^{(t)}$):
{{code}}
Failure Log ($\mathcal{F}^{(t)}$):
{{failure_log}}

Figure 10: Merged diagnostic analysis $(\mathcal{E}_{\pi}^{(t)}, \mathcal{E}_{c}^{(t)}, \mathcal{E}_{\text{align}}^{(t)}) \leftarrow \mathcal{A}_{\text{CDM}}(P, \pi^{(t)}, c^{(t)}, \mathcal{F}^{(t)})$ (Algorithm 1, line 9).

CDM SCORING ($\mathcal{A}_{\text{CDM}}$)

Task:
Evaluate each candidate decision using both *confidence* and *consistency* criteria, based on diagnostic insights produced by multiple analysis agents. The goal is to quantitatively assess which decision should be taken in the next iteration.

Context Provided:

- A set of candidate decisions (e.g., *update plan*, *update code only*).
- Diagnostic insights from multiple analysis types:
  - Plan analysis
  - Code analysis
  - Content (alignment) analysis
- Predefined analysis pairs for consistency evaluation.

Scoring Definitions:

- **Confidence ($\Phi^{(t)}$):** Measures how strongly a single analysis supports or refutes a given decision.
- **Consistency ($\Psi^{(t)}$):** Measures the degree of agreement between pairs of analyses regarding the same decision.

Scoring Rules:

- All scores must lie in the range $[0, 1]$.
- Higher confidence indicates stronger, more direct evidence.
- Higher consistency indicates stronger agreement across analyses.
- Contradictory insights must result in low scores.

Response Format (Strict JSON):

```
{
  "confidence_scores": {
    "<decision>": {
      "<analysis_type>": {
        "confidence": float,
        "reasoning": string
      }
    }
  },
  "consistency_scores": {
    "<decision>": {
      "<analysis1>-<analysis2>": {
        "consistency": float,
        "reasoning": string
      }
    }
  }
}
```

IMPORTANT:

- Output **JSON only**; do not include markdown or extra text.
- Use concise reasoning (1–3 sentences per score).
- If analyses contradict each other, assign low scores.

Decisions:
{{decisions}}
Diagnostic Insights:
{{analyses}}

Figure 11: CDM scoring of confidence and consistency $(\Phi^{(t)}, \Psi^{(t)}) \leftarrow \mathcal{A}_{\text{CDM}}(\mathcal{E}_{\pi}^{(t)}, \mathcal{E}_{c}^{(t)}, \mathcal{E}_{\text{align}}^{(t)})$ (Algorithm 1, line 10).

REASONING TRAJECTORY UPDATE ($\mathcal{A}_{\text{RT}}$)

Task:
Update the persistent debugging strategy based on newly observed diagnostic evidence, while maintaining continuity with the previous strategy.

Inputs:

- Previous strategy $R^{(t)}$
- Diagnostic evidence $\mathcal{E}_X^{(t)}$ for the selected target $X \in \{\pi, c\}$
- Problem description P
- Current target state $X^{(t)}$
- Failure log $\mathcal{F}^{(t)}$

Guidelines:

- Incorporate new evidence without repeating $R^{(t)}$ verbatim.
- State concrete next hypotheses or actions.
- Avoid ineffective or repeated fixes.
- Do **not** generate code or a new plan.

Output:
Return only the updated debugging strategy text $R^{(t+1)}$.

Figure 12: Reasoning trajectory update $R^{(t+1)} \leftarrow \mathcal{A}_{\text{RT}}(R^{(t)}, \mathcal{E}_X^{(t)}, P, X^{(t)}, \mathcal{F}^{(t)})$ (Algorithm 1, line 19).

PLAN REFINEMENT ($\mathcal{A}_{\text{plan}}$)

Task:
Refine the current plan based on observed failures and the updated reasoning trajectory, producing a corrected plan for the next iteration.

Inputs:

- Problem description P
- Current plan $\pi^{(t)}$
- Failure log $\mathcal{F}^{(t)}$
- Updated debugging strategy $R^{(t+1)}$

Guidelines:

- Modify the plan to address diagnosed errors.
- Ensure logical coherence and step-by-step correctness.
- Do **not** generate executable code.
- Output the plan only, without explanations.

Output:
Return the updated plan $\pi^{(t+1)}$.

Figure 13: Plan refinement $\pi^{(t+1)} \leftarrow \mathcal{A}_{\text{plan}}(P, \pi^{(t)}, \mathcal{F}^{(t)}, R^{(t+1)})$ (Algorithm 1, line 21).

CODE GENERATION AFTER PLAN UPDATE ($\mathcal{A}_{\text{code}}$)

Task:
Generate a new code implementation based on the refined plan, incorporating guidance from the updated reasoning trajectory.

Inputs:

- Refined plan $\pi^{(t+1)}$
- Coding template $\mathcal{T}$

Guidelines:

- Implement the solution strictly following $\pi^{(t+1)}$.
- Respect the coding template defined by $\mathcal{T}$.
- Generate a *new* implementation (do not reuse previous code).
- Do **not** include explanations or extra text.

Output:
Return the generated code $c^{(t+1)}$ only.

Figure 14: Code generation after plan update $c^{(t+1)} \leftarrow \mathcal{A}_{\text{code}}(\pi^{(t+1)}, \mathcal{T})$ (Algorithm 1, line 22).

CODE REFINEMENT / PATCHING ($\mathcal{A}_{\text{code}}$)

Task:
Refine the existing code implementation to correct observed failures, guided by diagnostic insights and the updated reasoning trajectory.

Inputs:

- Problem description P
- Current code $c^{(t)}$
- Coding template $\mathcal{T}$
- Failure log $\mathcal{F}^{(t)}$
- Updated debugging strategy $R^{(t+1)}$

Guidelines:

- Modify the code to address diagnosed errors.
- Incorporate guidance from $R^{(t+1)}$.
- Respect the coding template defined by $\mathcal{T}$.
- Do **not** reuse the same incorrect implementation.
- Do **not** add testing or assertion code.
- Output only the corrected code.

Output:
Return the refined code $c^{(t+1)}$ enclosed in a code block.

Figure 15: Code refinement / patching $c^{(t+1)} \leftarrow \mathcal{A}_{\text{code}}(P, c^{(t)}, \mathcal{F}^{(t)}, R^{(t+1)}, \mathcal{T})$ (Algorithm 1, line 25).