Self-Correcting Code Generation Using Small Language Models

Jeonghun Cho[1], **Deokhyung Kang**[1], **Hyounghun Kim**[1,2], **Gary Geunbae Lee**[1,2]
[1]Graduate School of Artificial Intelligence, POSTECH
[2]Department of Computer Science and Engineering, POSTECH
{jeonghuncho, deokhk, h.kim, gblee}@postech.ac.kr

arXiv:2505.23060v3 [cs.CL] 23 Aug 2025

Abstract

Self-correction has demonstrated potential in code generation by allowing language models to revise and improve their outputs through successive refinement. Recent studies have explored prompting-based strategies that incorporate verification or feedback loops using proprietary models, as well as training-based methods that leverage their strong reasoning capabilities. However, whether smaller models possess the capacity to effectively guide their outputs through self-reflection remains unexplored. Our findings reveal that smaller models struggle to exhibit reflective revision behavior across both self-correction paradigms. In response, we introduce **CoCoS**, an approach designed to enhance the ability of small language models for multi-turn code correction. Specifically, we propose an online reinforcement learning objective that trains the model to confidently maintain correct outputs while progressively correcting incorrect outputs as turns proceed. Our approach features an accumulated reward function that aggregates rewards across the entire trajectory and a fine-grained reward better suited to multi-turn correction scenarios. This facilitates the model in enhancing initial response quality while achieving substantial improvements through self-correction. With 1B-scale models, CoCoS achieves improvements of 35.8% on the MBPP and 27.7% on HumanEval compared to the baselines.[1]

1 Introduction

Despite the progress in large language models (LLMs), generating the correct code snippets in a single attempt remains a challenging task across many programming scenarios. In response, recent studies have explored the use of external supervision, typically provided by a more capable teacher model, to guide the correction process (Welleck et al., 2023; Yang et al., 2025). These approaches typically leverage teacher models, which either guide student models through iterative self-correction via feedback or perform both generation and correction in a self-directed manner without external models (Kamoi et al., 2024).

Given the dependence on powerful guidance, most of the successful results have been achieved using strong proprietary models such as Gemini (Team et al., 2023, 2024) and GPT-series (Brown et al., 2020; OpenAI, 2025), which are only available through external APIs (Chen et al., 2024, 2025). Considering the reliance on costly proprietary models in previous studies, it remains uncertain whether smaller, open-source language models (SLMs) can effectively guide their outputs through self-reflection (Huang et al., 2024; Han et al., 2024). To investigate this, we explore whether SLMs can achieve meaningful self-correction without proprietary systems. Our findings show that smaller models struggle to exhibit reflective revision behavior when they rely solely on prompting (§4.3). Building on these observations, we aim to explore training-based self-correction for code generation. However, we find that such methods are not directly applicable to SLMs, likely due to their assumption of sufficient self-correction capability (§5.2).

To tackle these limitations, we introduce **CoCoS** (Self-**Co**rrecting **Co**de generation using **S**mall LMs), a reinforcement learning (RL) framework designed to support intrinsic self-correction in small LMs. Our approach leverages an accumulated reward function that considers previous responses and outputs a scalar reward that captures the cumulative effect of multiple turns. This function encourages both successful self-correction and improved initial response quality by rewarding the entire response trajectory. Additionally, our method adopts a progressive reward that evaluates incremental improvements in code quality by tailoring the setting to a multi-turn scenario, which offers

[1]Our implementation can be accessed at https://github.com/jeonghun3572/CoCoS

a more fine-grained assessment than binary rewards based solely on code correctness. We validate our approach on 1B-scale models across various code generation tasks and observe robust generalization in unseen settings.

Our contributions are as follows: (1) We empirically demonstrate that previous prompting-based and training-based self-correction methods struggle to generalize to small language models. (2) We introduce a reinforcement learning reward function tailored for the multi-turn code generation setting, leveraging accumulated rewards with a discount factor and fine-grained assessments to encourage effective self-correction. (3) We validate our approach across diverse Python code generation datasets and show consistent improvements over baselines, including in unseen settings.

2 Related Work

Prompting-based correction guidance. Several recent studies on self-correction have leveraged the advanced reasoning capabilities of LLMs to improve the accuracy of their initial responses (Zelikman et al., 2022; Renze and Guven, 2024). In coding tasks that require complex reasoning, self-correction often involves reviewing generated code snippets to identify and fix errors with the model's own judgment. However, prior works suggest that naive prompting for self-correction may reduce performance (Huang et al., 2024; Qu et al., 2024). As a result, SETS (Chen et al., 2025) combines sampling, self-verification, and self-correction into a unified framework that facilitates LLMs to improve their own outputs. Self-Refine (Madaan et al., 2023) conducts iterative evaluation and correction of the initial responses using few-shot prompting without any additional fine-tuning. However, these methods rely on the assumption that the model is already capable of revising its outputs.

Training-based correction guidance. In addition to the above methods, fine-tuning has been explored to endow models with intrinsic self-correction. Self-Corrector (Welleck et al., 2023) employs a separate correction model that shares the same backbone as the initial response generator. Zhang et al. (2024) introduces a distinct verifier model to guide the revision process. Yang et al. (2025) propose a teacher-guided framework where a stronger model supervises both the intermediate reasoning steps and the final prediction of an SLM. However, all of these methods depend on training separate corrector or verifier models, which require additional computational resources.

Self-correction without external guidance. ReVISE (Lee et al., 2025) implements internal self-correction by generating special tokens that determine whether to revise or terminate the current output. SCoRe (Kumar et al., 2025) adopts a two-stage RL framework that provides rewards to both the previous and current responses to guide learning. While both methods have demonstrated improved code generation, ReVISE has limited generalization beyond the training turns due to its reliance on supervised fine-tuning (SFT) (Chu et al., 2025). SCoRe is the first work to alleviate this limitation through online learning with self-generated data. While both SCoRe and our approach leverage online-RL, SCoRe remains constrained to proprietary models, whereas CoCoS demonstrates effectiveness even for SLMs that struggle with self-correction.

3 Problem Description

In this work, we focus on code generation tasks and study a small-scale LLM to investigate its intrinsic ability to improve its outputs over subsequent trials. Suppose $\mathcal{D} = \{(x, y, u)\}$ consists of data points, each represented as a tuple (x, y, u), where x is a problem, y is the canonical code snippet, and u is the unit test cases. We define the SLM as π_θ and aim to generate correct $\hat{y}_t$ through the conditional distribution $\pi_\theta(\hat{y}_t|x, \hat{y}_{1:t-1}, p_{1:t-1})$, where $\hat{y}_{1:t-1}$ represents the history of SLM's previous trials, and $p_{1:t-1}$ denotes auxiliary instructions for revising the previous responses. Note that the subscripts indicate turn indices within the multi-turn setting. Since we do not consider external feedback (e.g., compiler outcomes, other LMs' feedback), $p_{1:t-1}$ is fixed for each turn.

4 Preliminary Study: Prompting-based Self-Correction

Prompting-based self-correction typically involves incorporating feedback into the prompt in the form of the auxiliary instruction sequence $p_{1:t-1}$ (Kamoi et al., 2024). This feedback can be either provided by a separate teacher model or generated by the model itself. In this section, we examine both settings to assess whether prompting-based self-correction is effective for small language models.

Method	Accuracy@t1	Accuracy@t2	$\Delta^{i\to c}$(t1, t2)	$\Delta^{c\to i}$(t1, t2)↓	Δ^{Acc}(t1, t2)
Simple-prompting	43.8%	44.6%	1.4%	0.6%	0.8%
Self-Refine	43.8%	38.2%	2.8%	8.4%	−5.6%
External-Refine	43.8%	41.8%	7.0%	9.0%	−2.0%
SETS	39.3%	38.8%	-	-	-
External-SETS	39.3%	**49.8%**	-	-	-

Table 1: Performance comparison of prompting-based methods.

4.1 Metrics

To evaluate self-correction, we follow the metrics used in Kumar et al. (2025). An output y_t is considered correct if it passes all test cases and is incorrect otherwise. **Accuracy@t1** and **Accuracy@t2** denote the accuracy at the first and second turns, respectively. $\Delta^{i\to c}$**(t1, t2)** denotes the fraction of problems that change from incorrect to correct between the first and second turns, and $\Delta^{c\to i}$**(t1, t2)** vice versa. Finally, Δ^{Acc}**(t1, t2)** measures the change in accuracy between the first and second turns.

4.2 Setup

We experiment with two prompting-based self-correction methods, **Self-Refine** (Madaan et al., 2023) and **SETS** (Chen et al., 2025)—both demonstrated exclusively on proprietary LLMs in their original work. To verify the effectiveness of SLMs, all experiments are conducted using the pre-trained Qwen2.5-1.5B (Yang et al., 2024). Self-Refine uses 3-shot prompting, whereas SETS relies on test-time scaling. Since SETS does not generate a fixed initial response, we estimate Accuracy@t1 by averaging the accuracy of 20 sampled initial outputs. Accuracy@t2 is similarly computed after all correction steps. However, because SETS verifies each sample and only applies correction to those deemed correct, Δ improvements between initial and final responses cannot be consistently measured. Instead, we report feedback accuracy in Appendix A.

We further investigate the influence of external feedback by employing strong teacher models,[2] which we refer to as **External-Refine** and **External-SETS**. For comparison, we also introduce a simple correction prompting method that first generates an initial response with a 3-shot prompt, then prompts for correction without disclosing the correctness of the initial output. Additional details including prompts and hyperparameters are provided in Appendix G.

[2]We use gpt-4.1-2025-04-14 (OpenAI, 2025) for teacher model.

4.3 Results

In conclusion, SLMs struggle to revise their responses using prompting alone, as shown in Table 1. Although simple prompting yields a marginal performance gain (Δ^{Acc} of 0.8), it fails to enable meaningful revisions. Both prior works, when used without an external LLM for self-verification, underperform compared to simple prompting and frequently modify responses that were initially correct. Effective self-correction becomes only possible when a strong external LLM provides feedback (External-SETS; 10.5% gain in Accuracy@t1). However, this setup assumes access to a powerful external model [2] and requires calling it alongside the SLM during inference, which is impractical in real-world scenarios.

5 Proposed Methodology

Based on the observations in §4, SLMs struggle to demonstrate reflective revision behavior when relying solely on prompting, without access to external LLMs. In this section, we introduce a training-based method designed to enable effective self-correction.

5.1 Multi-turn MDP

As outlined in §3, our goal is to enhance model outputs across multiple turns by leveraging the response history. Since the model's own generated history $\hat{y}_{1:t-1}$ naturally defines a sequential decision process, we formulate the process of self-correction as a multi-turn Markov decision process (Qu et al., 2024), where the policy is defined as

$$\pi_\theta(\hat{y}_t \mid x, \hat{y}_{1:t-1}, p_{1:t-1})$$

In this formulation, the state at each turn t is denoted as $s_t = (x, \hat{y}_{1:t-1}, p_{1:t-1})$, which consists

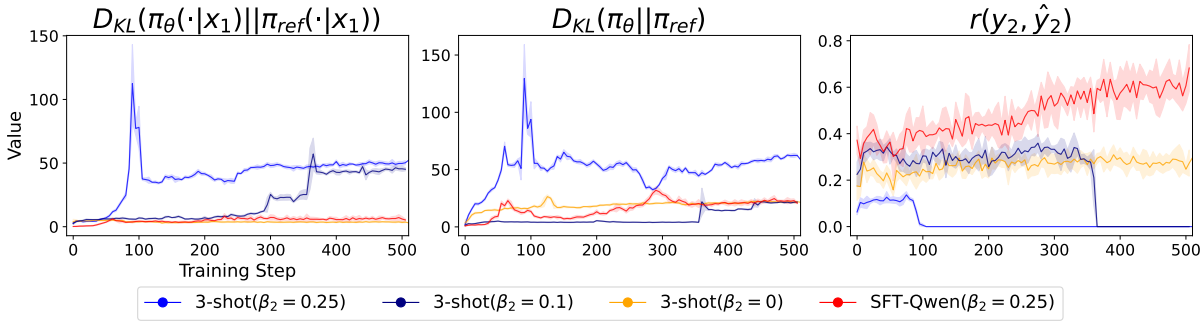

Figure 1: Learning curves for SCoRe based on experiments with Qwen2.5-1.5B. **(1)** KL-regularization for the first turn, **(2)** Default KL-divergence penalty for the policy gradient training, and **(3)** Reward for the second response. ***The KL-regularization for the first turn becomes excessively high as the training step increases so that the training collapses.*** We also observe collapse in another small model, which we analyze in Appendix E.

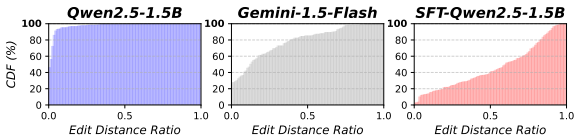

Figure 2: Cumulative distribution functions (CDFs) of edit distance ratios between the first and second responses, evaluated on Qwen2.5-1.5B and Gemini-1.5-Flash under the 3-shot setting.

of the input x, the model's prior outputs $\hat{y}_{1:t-1}$, and the prompt history $p_{1:t-1}$. The action a_t corresponds to generating a revised output $\hat{y}_t$. This action is then appended to form the next prompt p_t for the subsequent turn. Accordingly, the policy can be rewritten as $\pi_\theta(a_t \mid s_t)$. We evaluate the correctness of the generated code $\hat{y}_t$ using unit test execution $u(\hat{y}_t)$, which assesses the functional correctness of the code against the given unit tests. The test outcome is used as the reward:

$$r(y_t, \hat{y}_t) := u(\hat{y}_t)$$

where $r(y_t, \hat{y}_t)$ denotes the reward for $\hat{y}_t$.

Under this formulation, the learning objective is to optimize a policy π_θ that maximizes the expected cumulative reward over a refinement trajectory:

$$\max_\theta \mathbb{E}_{(x,y)\sim\mathcal{D},\, \hat{y}\sim\pi_\theta}\Big[\sum_{t=1}^{T} r(y_t, \hat{y}_t) - \beta\mathbb{D}_{KL}\big(\pi_\theta(\cdot \mid s_t) \,\|\, \pi_{ref}(\cdot \mid s_t)\big)\Big]$$

where $\beta > 0$ is the Kullback-Leibler (KL) coefficient that controls the strength of regularization against the reference policy π_{ref}.

5.2 Observed limitations

Previous training-based studies have used supervised learning to pair incorrect responses with corrections, enabling the model to revise incorrect outputs for multi-turn self-correction (Zelikman et al., 2022; Welleck et al., 2023). However, in an ideal scenario, LLMs also should aim to complete the given task on the first attempt, minimizing the need for correction. In other words, the goal of self-correction learning is to maximize not only Accuracy@t2 but also Accuracy@t1. To this end, SCoRe (Kumar et al., 2025) proposed a two-stage approach to optimize both turns. Specifically, in the first stage, training aims to maximize the reward of the second turn while applying strong KL-regularization to the first turn to preserve the distribution of the initial response. In the second stage, the model jointly optimizes both responses. This approach is the first to achieve positive intrinsic self-correction. The training formulation in the first stage is as follows:

$$r(y_2, \hat{y}_2) - \beta\sum_{t=1}^{2}\mathbb{D}_{KL}\big(\pi_\theta(\cdot \mid s_t) \,\|\, \pi_{ref}(\cdot \mid s_t)\big) - \beta_2\mathbb{D}_{KL}\big(\pi_\theta(\hat{y}_1 \mid s_1) \,\|\, \pi_{ref}(y_{ref} \mid s_1)\big) \quad (1)$$

where $r(y_2, \hat{y}_2)$ is a binary reward, taking a value of 1 for all-pass and 0 for fail. The second KL-divergence term represents the regularization, and $\beta \ll \beta_2$. However, we find that introducing the second KL-divergence term ($\beta_2 > 0$) leads to training collapse for SLMs, as shown in Figure 1 where the reward converges to zero.

We hypothesize that training collapse arises from the limited self-correction ability of SLMs. In the

simple-prompting baseline (Table 1), the low values of $\Delta^{i \to c}$ and $\Delta^{c \to i}$ suggest that the model rarely revises its initial response, often producing near-identical outputs across turns ($\hat{y}_1 \approx \hat{y}_2$). This is further supported by Figure 2, where Qwen2.5-1.5B shows minimal changes (edit distance ≤ 0.05) in 93% of cases when the second response is incorrect, compared to only 32% for Gemini.

This lack of revision becomes problematic when optimizing the reward objective in Equation 1. While the reward term encourages updates to y_2, the strong overlap with y_1 causes the KL regularization—applied to y_1—to inadvertently constrain y_2, thereby destabilizing training and limiting reward-driven updates.

To further investigate our hypothesis, we construct an SFT-Qwen model that is intentionally fine-tuned to produce outputs where $y_1 \neq y_2$. The edit distance between y_1 and y_2 can be observed in the right histogram in Figure 2. To this end, we create training data by mixing $\hat{y}_1$ and $\hat{y}_2$ generated from Pre-trained Qwen and Gemini, respectively, and deliberately overfit the model to MBPP to instill correction capability. Figure 1 illustrates the impact of KL-regularization on Pre-trained Qwen and SFT-Qwen. We observe that first-turn KL-regularization explodes during training. This leads to instability in the learning process, driving the reward toward zero and ultimately resulting in training collapse. In contrast, disabling this regularization (i.e., $\beta_2 = 0$) or using SFT-Qwen prevents such instability. Interestingly, SFT-Qwen does not exhibit training collapse, even when trained with a KL-regularization coefficient ($\beta_2 = 0.25$)—a large value that reflects SCoRe's intention—thus supporting our hypothesis about the conditions that give rise to instability. Building on this experiment, we propose a method to enhance intrinsic self-correction even in SLMs.

5.3 CoCoS

To enable self-correction, it is crucial to capture the difference between the initial and subsequent responses. A straightforward approach—maximizing only the difference of rewards at each turn by subtracting $r(y_1, \hat{y}_1)$ from $r(y_2, \hat{y}_2)$—may lead to reward hacking, where the model intentionally degrades $r(y_1, \hat{y}_1)$ to inflate the reward difference (Skalse et al., 2022). Therefore, SCoRE introduced the KL-regularization to prevent this collapse; however, SLMs struggle with this strategy (§5.2). To address this, we introduce CoCoS, an RL-based method that uses a discount factor γ to accumulate rewards from prior responses, preventing the collapse of the initial response while also providing fine-grained assessments through progressive reward that capture incremental improvements between turns.

Since our setting is also formulated as a multi-turn MDP, we adopt a REINFORCE-style policy gradient method (Ahmadian et al., 2024) to optimize the policy, which is widely used in single-turn RLHF ($\gamma = 0$). In order to adapt to a multi-turn setting, we introduce two key modifications to the reward function. First, we introduce an accumulated reward function that incorporates a discount factor γ to encourage the model to consider the assessment of the previous turn; in the case of SCoRe, $\gamma = 0$ since it computes the reward independently at each turn. Unlike conventional RL, where γ discounts future rewards, we adapt its usage to amplify the contribution of the most recent turn. The accumulated reward function $R(\hat{y}_{1:T})$ is defined as follows (with $T = 2$ in our setup):

$$R(\hat{y}_{1:T}) = \gamma^{T-1} r_1 + \sum_{t=2}^{T} \gamma^{T-t}(r_t - r_{t-1}) \quad (2)$$

where r_t denotes the reward at the t-th turn. When $\gamma < 1$, the function focuses more on recent changes in reward, whereas $\gamma > 1$, in principle, places greater emphasis on earlier responses. Based on empirical observations, we set $\gamma = 0.5$ in our experiments. The impact of varying γ is discussed in §8.1.

In multi-turn settings, capturing the model's progress over successive turns is important. However, prior work has typically used a binary reward that reflects whether all test cases pass, which makes it difficult to account for gradual improvements during the refinement process (Zheng et al., 2025). To address this, we introduce a progressive reward based on the pass ratio. We assume that each code is paired with K unit test cases to assess semantic correctness, and the progressive reward is computed as follows:

$$r_t = \frac{1}{K} \sum_{k=1}^{K} \mathbb{I}\{u_k(\hat{y}_t) \text{ passes}\} \quad (3)$$

The effectiveness of the progressive reward compared to the binary scheme is presented in §8.2. Accordingly, our training objective is defined as

Method	Accuracy@t1	Accuracy@t2
Pre-Trained (3-shot)		
Qwen2.5-1.5B	43.8%	44.6%
Llama-3.2-1B	27.6%	27.6%
deepseek-coder-1.3B	46.4%	46.0%
Boost model (0-shot)		
Qwen2.5-1.5B	9.0% (34.8%↓)	10.2% (34.4%↓)
Llama-3.2-1B	10.4% (17.2%↓)	11.0% (16.6%↓)
deepseek-coder-1.3B	16.6% (29.8%↓)	20.2% (25.8%↓)

Table 2: Performance comparison between pre-trained and boost models on the MBPP dataset. Pre-Trained models are evaluated using 3-shot prompting. ***The results suggest that using boost models as the backbone in our experiments has minimal impact.***

follows:

$$\max_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D},\, \hat{y}\sim\pi_\theta}\Big[R(\hat{y}_{1:T}) - \beta \sum_{t=1}^{T} \mathbb{D}_{KL}\big(\pi_\theta(\cdot \mid s_t) \parallel \pi_{ref}(\cdot \mid s_t)\big)\Big] \tag{4}$$

where β denotes the KL coefficient. Further experimental details can be found in Appendix C.

6 Experimental Settings

As prior studies are insufficient for enabling intrinsic self-correction (§4), we compare CoCoS against fine-tuning approaches. To support this comparison, we introduce the experimental settings, including descriptions of the models (§6.1), datasets (§6.2), and baselines (§6.3) used in our experiments.

6.1 Models

We use Qwen2.5-1.5B (Yang et al., 2024), Llama-3.2-1B (Grattafiori et al., 2024), and DeepSeek-Coder-1.3B-Base (Guo et al., 2024) as base models and fine-tune them for each method. During inference, we use greedy decoding to ensure reproducibility. Further implementation details are provided in Appendix C.

6.2 Datasets

All models are trained on MBPP (Austin et al., 2021) and evaluated on MBPP, HumanEval (Chen et al., 2021), and ODEX (Wang et al., 2023) to assess generalization to unseen data. For MBPP, we follow the data split provided in Austin et al. (2021), which defines train, validation, test, and few-shot sets. We do not use the validation set for model selection; instead, we select the checkpoint that achieves the highest reward during training. Also, few-shot data is excluded.

While supervised fine-tuning (SFT) allows models to learn proper code formatting through input-output pairs, online-RL lacks such structural guidance. Therefore, we perform SFT on the KodCode (Xu et al., 2025) dataset to produce parseable code so that RL training can be conducted with unit test execution. KodCode is curated to exclude data from MBPP, HumanEval, and ODEX. We measure the model's accuracy on MBPP at both the pre-SFT and post-SFT, which we denote as the **Pre-trained** and **Boost model**, respectively, and report the results in Table 2. As shown in Table 2, we emphasize that the purpose of SFT on KodCode is not to enhance performance on the target dataset. The **Boost model** is the backbone for both CoCoS and all baseline comparisons.

6.3 Baselines

We compare CoCoS with two training-based methods: **Self-Corrector**, which adopts a separate corrector model, and **ReVISE**, which follows an intrinsic self-correction approach. Since ReVISE is trained on a fixed two-turn dataset, it is constrained to generate a terminate token within two turns. This poses a limitation in terms of turn scalability. To enable multi-turn correction beyond two turns, we train our own SFT baseline, **Turn-SFT**, using fixed 1-turn and 2-turn examples. Turn-SFT can generalize beyond two turns by following the trained data format during inference, offering better turn scalability than ReVISE and making it suitable for analyzing self-correction beyond two turns.

However, such SFT-based methods still rely on the distribution of the training dataset, which highlights the need for RL–based approaches with better generalization. To compare with an RL-based approach, we also train **Turn-RL**, where the first response is fixed and the second turn is optimized according to Equation 4. In other words, Turn-RL constrains the optimization of the initial response in contrast to CoCoS. Further details on the baselines are provided in Appendix B.

7 Main Results

As shown in Table 3, CoCoS demonstrates the highest correction performance across all turns. Although the baselines are trained to actively revise prior responses, the $\Delta^{c\to i}$ metric indicates that they

Method	Accuracy@t1	Accuracy@t2	$\Delta^{i\to c}$(t1, t2)	$\Delta^{c\to i}$(t1, t2)↓	Δ^{Acc}(t1, t2)
Boost model	9.0%	10.2%	1.2%	0.0%	1.2%
Turn-SFT	44.2%	44.4%	1.4%	1.2%	0.2%
Self-Corrector	43.8%	48.8%	11.2%	6.2%	5.0%
ReVISE	34.6%	41.2%	11.3%	4.9%	6.4%
Turn-RL	**46.4%**	48.6%	4.0%	1.8%	2.2%
CoCoS	45.0%	**54.2%**	11.0%	1.8%	9.2%

Table 3: Main results of Qwen2.5-1.5B on the MBPP dataset. The highest accuracy values are highlighted in bold. ***CoCoS has the highest selective correction rate, correcting incorrect responses without changing correct ones. It also demonstrates strong generalization, achieving high correction success rates even on unseen data.***

Method	Accuracy@t1	Accuracy@t2	Δ^{Acc}(t1, t2)
MBPP			
Boost model	10.4 / 16.6	11.0 / 20.2	0.6 / 3.6
Turn-SFT	23.2 / 45.0	23.6 / 46.0	0.4 / 1.0
Self-Corrector	27.6 / 46.4	25.8 / 47.0	−1.8 / 0.6
ReVISE	29.0 / **48.6**	30.8 / 47.6	0.8 / −1.2
Turn-RL	16.6 / 36.4	16.4 / 55.4	−0.2 / 19.0
CoCoS	**57.2** / 48.2	**59.4** / **60.4**	2.2 / 12.2
HumanEval			
Boost model	15.2 / 21.3	15.2 / 21.3	0.0 / 0.0
Turn-SFT	14.6 / 19.5	12.8 / 22.0	−1.9 / 2.5
Self-Corrector	11.0 / 20.1	12.8 / 22.6	1.9 / 2.4
ReVISE	13.4 / **23.8**	11.9 / 19.0	−1.9 / −4.3
Turn-RL	8.5 / 23.2	8.5 / **25.6**	0.0 / 2.4
CoCoS	**34.1** / 22.6	**39.6** / 25.0	5.5 / 2.4
ODEX			
Boost model	13.2 / 27.3	12.8 / 27.6	−0.5 / −0.3
Turn-SFT	15.7 / **29.8**	15.3 / 28.7	−0.4 / −1.1
Self-Corrector	13.2 / 27.3	14.4 / 28.0	1.1 / 0.7
ReVISE	9.6 / 21.2	11.4 / 22.8	1.8 / 1.6
Turn-RL	19.8 / 23.6	21.9 / 30.5	2.1 / 6.9
CoCoS	**23.2** / 26.2	**25.1** / **31.4**	1.8 / 5.2

Table 4: Accuracy comparison across different models. Results are reported as Llama-3.2-1B / deepseek-coder-1.3B. Percentage units (%) are omitted from the table. The full results are provided in Appendix D.

frequently make unnecessary revisions to already correct outputs. CoCoS, on the other hand, corrects selectively: it achieves a low $\Delta^{c\to i}$ rate of 1.8%, avoiding unnecessary changes, while maintaining a high $\Delta^{i\to c}$ rate of 11%. This results in a Δ^{Acc} of 9.2%, the highest among all baselines.

7.1 Evaluation in broader scenarios

To evaluate the generalization of our approach, we report additional results on alternative models and unseen datasets in Table 4. CoCoS outperforms other baselines even when using SLMs other than Qwen as the backbone. For instance, using Llama

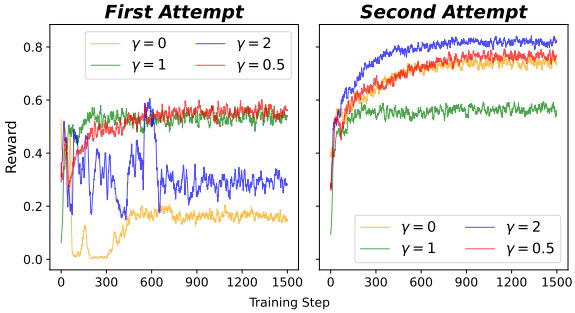

Figure 3: Learning curves under varying values of discount factor γ.

as the backbone, it achieves the highest correction rate on MBPP, with a Δ^{Acc} of 12.2%. Additionally, The SFT-based baselines (Turn-SFT, Self-Corrector, and ReVISE) often fail to generalize beyond their training distribution (Chu et al., 2025), as evidenced by the comparable performance to their backbones. On the MBPP, SFT-based baselines show up to a 2× improvement over the backbone in Accuracy@t2. However, in unseen settings, their correction performance is comparable to that of the Boost model or even degrades. In contrast, CoCoS consistently yields positive values on the Δ^{Acc} metric, demonstrating stable improvement across turns.

8 Analyses

In this section, we analyze the discount factor in the accumulated reward function (§8.1), compare progressive and binary reward schemes (§8.2), examine multi-turn correction beyond two turns (§8.3), evaluate the model's robustness to varied correction instructions during inference (§8.4), and conduct case studies on representative examples (§8.5).

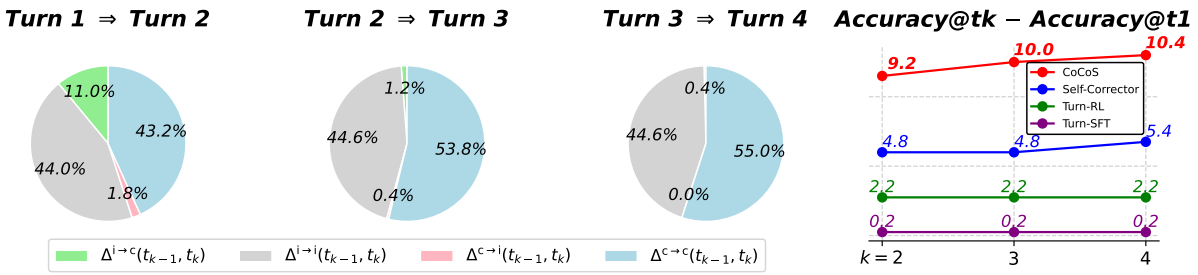

Figure 4: Turn-wise metric changes on MBPP using the Qwen model. The pie charts represent the distribution of transition types: $\Delta^{i\to c}$, $\Delta^{i\to i}$, $\Delta^{c\to i}$, and $\Delta^{c\to c}$ across successive turns. The line plot on the right shows the Accuracy@tk − Accuracy@t1 as the number of turns increases.

Dataset	Accuracy@t1	Accuracy@t2	$\Delta^{i\to c}_{\text{unit}}$	$\Delta^{c\to i}_{\text{unit}}\downarrow$
		Binary / Progressive		
MBPP	45.0 / 45.0	51.2 / **54.2**	10.5 / **11.9**	3.4 / **1.9**
HumanEval	24.4 / **29.3**	27.4 / **32.3**	**9.6** / 4.7	5.0 / **3.1**
ODEX	23.0 / 23.0	25.7 / **28.9**	9.8 / **34.2**	3.9 / **0.0**

Table 5: Comparison of progressive vs. binary reward schemes. Qwen2.5-1.5B models are trained under each reward setting and evaluated on MBPP.

8.1 Discount factor

As shown in Equation 2, when $T = 2$, the reward function is formulated as $R(\hat{y}_{1:2}) = (r_2 - r_1) + \gamma \cdot r_1$. We investigate the effect of varying the discount factor γ on training behavior, with learning curves shown in Figure 3. When $\gamma = 0$, training focuses on maximizing the reward difference $r_2 - r_1$, often by deliberately decreasing r_1. When $\gamma = 1$, the reward depends only on the second response, disregarding the first turn. As a result, responses across turns become tightly coupled, leading to poor coverage in subsequent iterations (Kumar et al., 2025). Under this setting, the model fails to develop the ability to self-correct.

Additionally, setting $\gamma = 2$ to jointly optimize both responses causes instability during training. In this case, it becomes unclear which of the two responses contributed positively to the total reward. This situation is called the credit assignment problem in MDPs (Pignatelli et al., 2024). This is beyond the scope of the current research and is not further addressed in this work. Consequently, we set $\gamma = 0.5$ to balance the quality of the initial response and the refinement achieved in the second.

8.2 Progressive reward

We compare the progressive reward (as defined in Equation 3) with a binary assessment scheme. To enable fine-grained evaluation, we assess the result of each unit test case individually. Specifically, $\Delta^{c\to i}_{\text{unit}}$ measures how many test cases drop from passing to failing between turns. Under the progressive reward, a transition from 2 passes to 1 is penalized for discouraging such behavior, whereas the binary reward assigns 0 in both cases, failing to capture the distinction.

Table 5 presents the results of models trained with the two reward schemes. The progressive reward demonstrates the strong preservation of correct unit test cases. For instance, on the ODEX, 34% of test cases show improvement, with a 100% preservation rate. In comparison, the binary reward causes around 40% of previously passed test cases to fail in subsequent turns. These results highlight the importance of fine-grained assessment, which has been overlooked in the multi-turn scenario.

8.3 Turn-wise changes

We analyze how CoCoS affects the distribution of correct and incorrect responses over multiple turns. Figure 4 presents the distribution and trend of changes in response correctness. First, the pie charts show that the proportion of $\Delta^{c\to c}$ gradually increases with each turn. This indicates that once a correct response is generated, the model maintains increasing consistency in subsequent turns.

Second, the proportion of $\Delta^{c\to i}$ remains consistently small, and it continues to decrease over successive turns. This suggests that CoCoS enables a model to refine its code iteratively, maintaining confidence in correct responses while effectively reducing errors over time. Finally, the line plot illustrates the cumulative net gain in Δ^{Acc}. CoCoS achieves steady gains over the baseline across turns, even though it starts from a high Accuracy@t1.

Model	Accuracy@t2	Δ^{Acc}(t1, t2)
Template / Fixed		
Qwen2.5-1.5B	52.8 / 54.2	7.2 / 9.2
Llama-3.2-1B	58.6 / 59.4	1.4 / 2.2
deepseek-coder-1.3B	60.2 / 60.4	12.0 / 12.2

Table 6: Evaluation results with varied instruction rephrasings on the MBPP dataset.

8.4 Auxiliary instruction

To evaluate the generalization ability of our method across different prompts, we intentionally use a different instruction prompt at test time than the one used during training (as defined in Appendix G.1). Instead of the fixed training prompt, we manually rephrase it into five alternative versions for the correction task. The results are shown in Table 6, and the rephrased instructions are provided in Appendix H. While the fixed prompt used during training yields slightly better overall performance, the model exhibits no substantial degradation under instruction variation. The average performance drop at the second attempt is only 0.8%, indicating robustness to diverse prompt variations.

8.5 Case Study

To analyze the types of mistakes that CoCoS makes and how it corrects them, we conduct three case studies. The observed error types include: (1) naming errors, (2) complex logic errors, and (3) incorrect usage of code libraries. In the main text, we focus on the first category, while detailed discussions of the latter two are provided in the Appendix F.

```
# Test case
assert test_duplicate(([1,2,3,4,5]))==False
assert test_duplicate(([1,2,3,4, 4]))==True
assert test_duplicate([1,1,2,2,3,3,4,4,5])==True

# Initial response
# wrong function name; won't match the test case
def contains_duplicate(nums):
    return len(nums)!= len(set(nums))

# Corrected response
def test_duplicate(arr):
    return len(arr)!= len(set(arr))
```

Code 1: Example of correcting a misnamed function

Code 1 highlights an issue related to function name generation. Given the test cases, the model is expected to generate a function name that allows the code to execute correctly. However, in some instances, the model fails to produce the appropriate function name. In such cases, even if the core logic is correctly implemented, the test cases cannot be executed, and the result is considered a generation failure. In the following turn, when prompted to correct the previous error, the model is able to preserve the correct logic and revise only the function name, thereby resolving the issue.

9 Conclusion

In this paper, we investigated self-correction methods for enabling SLMs to revise their own generated code. Our findings show that, despite their effectiveness in proprietary models, existing prompting- and training-based approaches fall short when applied to SLMs. To address these shortcomings, we propose CoCoS, which introduces an RL reward scheme tailored to the multi-turn code generation setting. By incorporating fine-grained assessment and an accumulated reward function, CoCoS demonstrates both successful self-correction and improved initial response quality through trajectory-level optimization across diverse SLMs and previously unseen scenarios.

Acknowledgments

This work was supported by the IITP(Institute of Information & Coummunications Technology Planning & Evaluation)-ITRC(Information Technology Research Center) grant funded by the Korea government(Ministry of Science and ICT)(IITP-2025-RS-2024-00437866, 47.5%). This work was supported by Smart HealthCare Program funded by the Korean National Police Agency(KNPA) (No. RS-2022-PT000186, 47.5%). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2019-II191906, Artificial Intelligence Graduate School Program(POSTECH), 5%).

Limitations

Online reinforcement learning incurs a cost at every action step, as the model learns by interacting with the environment in real-time. Due to infrastructural constraints, we limited CoCoS training to only two turns. In the turn-wise analysis section, CoCoS consistently demonstrated accuracy gains as the number of turns increased. These results suggest that increasing the number of training turns has the potential to yield further improvements. Therefore, future work should extend our approach to support multi-turn training beyond two turns, potentially

incorporating more cost-efficient strategies such as offline reinforcement learning.

For the same reason, our experiments were conducted only on a 1B-scale small language model. While we demonstrated that our methodologies are effective on small models, our approach is not limited to this setting. Moreover, evaluating its scalability to models larger than 1B parameters remains an interesting aspect to observe in future work.

Ethical Considerations

In our research, we use datasets such as KodCode (Xu et al., 2025), MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), and ODEX (Wang et al., 2023), which are licensed under CC BY-NC 4.0, CC BY 4.0, MIT, and CC BY-SA 4.0, respectively. The models GPT-4.1 (OpenAI, 2025), Qwen2.5-1.5B (Yang et al., 2024), Llama-3.2-1B (Grattafiori et al., 2024), and deepseek-coder-1.3B (Guo et al., 2024) are licensed under OpenAI, Apache-2.0, Llama 3.2 Community, and deepseek-license, respectively. All models were used strictly for research purposes, and no artifacts were utilized beyond the scope of the study.

References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267, Bangkok, Thailand. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Jiefeng Chen, Jie Ren, Xinyun Chen, Chengrun Yang, Ruoxi Sun, and Sercan Ö Arık. 2025. Sets: Leveraging self-verification and self-correction for improved test-time scaling. *arXiv preprint arXiv:2501.19306*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. 2025. Sft memorizes, rl generalizes: A comparative study of foundation model post-training. *arXiv preprint arXiv:2501.17161*.

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*.

Haixia Han, Jiaqing Liang, Jie Shi, Qianyu He, and Yanghua Xiao. 2024. Small language model can self-correct. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):18162–18170.

Shengchao Hu, Li Shen, Ya Zhang, Yixin Chen, and Dacheng Tao. 2024. On transforming reinforcement learning with transformers: The development trajectory. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):8580–8599.

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

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. 2025. Multi-turn code generation through single-step rewards. In *Forty-second International Conference on Machine Learning*.

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.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru,

George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. 2025. Training language models to self-correct via reinforcement learning. In *The Thirteenth International Conference on Learning Representations*.

Pawel Ladosz, Lilian Weng, Minwoo Kim, and Hyondong Oh. 2022. Exploration in deep reinforcement learning: A survey. *Information Fusion*, 85:1–22.

Hyunseok Lee, Seunghyuk Oh, Jaehyung Kim, Jinwoo Shin, and Jihoon Tack. 2025. Revise: Learning to refine at test-time via intrinsic self-verification. *arXiv preprint arXiv:2502.14565*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc.

Viktor Moskvoretskii, Chris Biemann, and Irina Nikishina. 2025. Self-taught self-correction for small language models. *Preprint*, arXiv:2503.08681.

OpenAI. 2025. Introducing gpt-4.1 in the api.

Eduardo Pignatelli, Johan Ferret, Matthieu Geist, Thomas Mesnard, Hado van Hasselt, and Laura Toni. 2024. A survey of temporal credit assignment in deep reinforcement learning. *Transactions on Machine Learning Research*. Survey Certification.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. 2024. Recursive introspection: Teaching language model agents how to self-improve. In *Advances in Neural Information Processing Systems*, volume 37, pages 55249–55285. Curran Associates, Inc.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. 2025. A self-improving coding agent. *Preprint*, arXiv:2504.15228.

Amrith Setlur, Katie Kang, Aviral Kumar, Feryal Behbahani, Roberta Raileanu, and Rishabh Agarwal. 2024. Self-improving foundation models without human supervision. In *ICLR 2025 Workshop Proposals*.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. Defining and characterizing reward hacking. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, and 1 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290, Singapore. Association for Computational Linguistics.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2023. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. 2025. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Ling Yang, Zhaochen Yu, Tianjun Zhang, Minkai Xu, Joseph E. Gonzalez, Bin CUI, and Shuicheng YAN. 2025. Supercorrect: Advancing small LLM reasoning with thought template distillation and self-correction. In *The Thirteenth International Conference on Learning Representations*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. STar: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*.

Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. 2024. Small language models need strong verifiers to self-correct reasoning. In

Findings of the Association for Computational Linguistics: ACL 2024, pages 15637–15653, Bangkok, Thailand. Association for Computational Linguistics.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, benjamin negrevergne, and Gabriel Synnaeve. 2025. What makes large language models reason in (multi-turn) code generation? In *The Thirteenth International Conference on Learning Representations*.

A Additional Experiment on SETS

Verifier	True Positive	False Positive	True Negative	False Negative
Qwen2.5-1.5B	35.1%	54.1%	7.2%	3.6%
GPT-4.1	27.6%	22.2%	43.1%	7.1%

Table 7: Verification accuracy based on different verifiers. ***The Qwen verifier shows near-random performance, while the teacher verifier reaches 70% accuracy.***

To assess the accuracy of the verifier module, we directly compare the verification results from SETS with the test case pass rate. Specifically, we use the test cases as the ground-truth labels and the verifier as the predictor and report the confusion matrix in Table 7. The Qwen verifier achieves an accuracy of 42%, which is close to random performance in distinguishing correct and incorrect responses. This indicates that small-scale Qwen fails to provide effective feedback, thereby harming rather than improving the initial response. In contrast, the teacher verifier reaches a higher verification accuracy of 70%. However, despite employing a costly verifier with relatively high accuracy, the results in Table 1 show only a marginal 5% gain over simple prompting.

B Baseline Details

In this section, we provide a detailed overview of the baseline used in our experimental evaluations. To implement the baseline, we constructed the training datasets, and to illustrate this process, we provide the 1-turn and 2-turn data samples. The examples are as follows:

Data Sample B.1: 1-Turn Example

```
You are an expert Python programmer, and here is your task: problem Your code should pass these tests:

{{test cases}}

[BEGIN]
{{correct code}}
[DONE]
```

Data Sample B.2: 2-Turn Example

```
You are an expert Python programmer, and here is your task: problem Your code should pass these tests:

test cases

[BEGIN]
{{incorrect code}}
[DONE]

There might be an error in the code above because of a lack of understanding of the question.
Please correct the error, if any, and rewrite the solution. Only output the final correct Python
program!

[CORRECT]
{{correct code}}
[DONE]
```

Turn-SFT. We generate training data using pre-trained models according to the format specified in §G.1. For each problem, we sample 10 candidate code solutions and determine their correctness based on the results of corresponding unit test cases. We then organize this data into two separate datasets: 1-turn and 2-turn. We combine the 1-turn and 2-turn datasets in a 1:1 ratio to match the data scale of other baseline methods, allowing the model to optimize initial code generation and correction capabilities.

Self-Corrector (Welleck et al., 2023). Self-Corrector acts as a specialized plug-in designed exclusively for the correction task. It trains as a separate component from the initial response model and generates only corrected outputs. For training data, we only use 2-turn data, which removes the initial code generation phase and focuses entirely on correction. We use the same LLM for both the response and correction LLM, but we freeze the response LLM during training while updating the correction LLM.

ReVISE (Lee et al., 2025). ReVISE uses a dedicated `[refine]` token to trigger the correction phase explicitly. The model generates either a `[refine]` token or an `[eos]` token. When the `[refine]` token is produced, the model enters the correction phase and continues generating outputs until it produces the `[eos]` token, which signals the end of the correction process. This design allows the model to complete self-correction in a single pass without intermediate user inputs, which eliminates the need for multiple generation stages. We generate data through sampling-based decoding and train the model by applying SFT loss and Direct Preference Optimization (Rafailov et al., 2023) loss during the backward pass.

Turn-RL. Turn-RL is trained to generate the `correct code` following the `[CORRECT]` token in 2-turn data, while the 1-turn data is pre-sampled and remains fixed during training. The training objective and reward function are shared with CoCoS.

C Experimental Details

C.1 Policy Optimization

We adopt the policy gradient method using the REINFORCE Leave-One-Out (RLOO) estimator (Ahmadian et al., 2024). RLOO provides a simple and efficient baseline, and can further reduce variance when multiple online samples are available by using each sample's reward as a baseline for others and averaging gradient estimates. We extend this approach to the multi-turn setting to train CoCoS. Our training objective, as defined in Equation 4, is as follows:

$$J(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D},\, \hat{y}\sim\pi_\theta}\Big[R(\hat{y}_{1:T}) - \beta \sum_{t=1}^{T} \mathbb{D}_{KL}\big(\pi_\theta(\cdot \mid s_t) \parallel \pi_{ref}(\cdot \mid s_t)\big)\Big]$$

We then optimize the expected return using policy gradients with the RLOO estimator:

$$\nabla_\theta J(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D},\, \hat{y}\sim\pi_\theta}\big[R'(\hat{y}_{1:T}) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\hat{y}_t \mid s_t)\big] - \beta \sum_{t=1}^{T} \nabla_\theta \mathbb{D}_{KL}\big(\pi_\theta(\cdot \mid s_t) \parallel \pi_{ref}(\cdot \mid s_t)\big)$$

Here, $R'(\hat{y}_{1:T})$ denotes the leave-one-out baseline-adjusted reward computed over the entire trajectory. For a sampled trajectory $\hat{y}^{(i)}_{1:T}$, the trajectory-level reward is defined as:

$$R(\hat{y}^{(i)}_{1:T}) = \gamma^{T-1} r^{(i)}_1 + \sum_{t=2}^{T} \gamma^{T-t}(r^{(i)}_t - r^{(i)}_{t-1})$$

where $r^{(i)}_t$ is the scalar reward associated with the model's response $\hat{y}^{(i)}_t$ at turn t. Given k such sampled trajectories, we define the leave-one-out adjusted reward as:

$$R'(\hat{y}^{(i)}_{1:T}) = R(\hat{y}^{(i)}_{1:T}) - \frac{1}{k-1} \sum_{j\neq i} R(\hat{y}^{(j)}_{1:T})$$

This estimator leverages the rewards of the other $k-1$ samples as a baseline, reducing variance while maintaining unbiasedness. Accordingly, the full policy gradient is computed as:

$$\therefore \nabla_\theta J(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\Bigg[\frac{1}{k}\sum_{i=1}^{k}\Big(R'(\hat{y}^{(i)}_{1:T}) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\hat{y}^{(i)}_t \mid s^{(i)}_t) - \beta \sum_{t=1}^{T} \nabla_\theta \mathbb{D}_{KL}\big(\pi_\theta(\cdot \mid s^{(i)}_t) \parallel \pi_{ref}(\cdot \mid s^{(i)}_t)\big)\Big)\Bigg]$$

C.2 Hyperparameter

We trained the model via the Transforming Reinforcement Learning (Hu et al., 2024). For both baselines and CoCoS training, we used 4 NVIDIA A100-SXM4-80GB. For inference, all experiments were conducted on a single NVIDIA RTX 6000 Ada Generation. The hyperparameters used in our experiments are listed in Table 8. Rather than conducting an extensive hyperparameter search, we trained all models with a unified set of hyperparameters for CoCoS, regardless of the backbone model.

Hyperparameters	**Boost model**	**CoCoS**
Dataset	KodCode	MBPP
Global batch size	256	128
Optmizer	AdamW	Adam
Weight decay	0.1	-
Learning rate	2e-5	1e-5
LR scheduler	cosine	cosine
Training steps	1000	1500
Sampling temperature	-	0.9
KL coefficient (β)	-	0.01
RLOO samples (k)	-	2

Table 8: Hyperparameters used in our experiments.

C.3 Dataset

KodCode		**MBPP**				**HumanEval**	**ODEX**
Train	Validation	Train	Validation	Test	Few-shot	Test	Test
144,068	36,018	374	90	500	10	164	439

Table 9: Dataset sizes used in our experiments.

We report all the datasets used in our experiments in Table 9. The KodCode (Xu et al., 2025) dataset was used for training the boost model, and we preprocessed the data to extract only samples that can be converted into our 2-turn format. As a result, out of 484k samples, only 180k were used for training. These samples were then randomly split into an 8:2 train/validation set. In MBPP (Austin et al., 2021), given that our boost models were already pre-trained, we excluded few-shot data. Instead of using separate validation data, we selected the final model checkpoint based on the highest reward achieved during training.

D Detailed Results

D.1 MBPP

Method	Accuracy@t1	Accuracy@t2	$\Delta^{i\to c}$(t1, t2)	$\Delta^{c\to i}$(t1, t2)↓	Δ^{Acc}(t1, t2)
Qwen2.5-1.5B					
Boost model	9.0%	10.2%	1.2%	0.0%	1.2%
Turn-SFT	44.2%	44.4%	1.4%	1.2%	0.2%
Self-Corrector	43.8%	48.8%	11.2%	6.2%	5.0%
ReVISE	34.6%	41.2%	16.1%	4.9%	6.4%
Turn-RL	46.4%	48.6%	4.0%	1.8%	2.2%
CoCoS	45.0%	54.2%	11.0%	1.8%	9.2%
Llama-3.2-1B					
Boost model	10.4%	11.0%	0.6%	0.0%	0.6%
Turn-SFT	23.2%	23.6%	1.2%	0.8%	0.4%
Self-Corrector	27.6%	25.8%	5.8%	7.6%	−1.8%
ReVISE	29.0%	30.8%	5.8%	5.0%	0.8%
Turn-RL	16.6%	16.4%	0.2%	0.4%	−0.2%
CoCoS	57.2%	59.4%	3.6%	1.4%	2.2%
deepseek-coder-1.3B					
Boost model	16.6%	20.2%	3.6%	0.0%	3.6%
Turn-SFT	45.0%	46.0%	4.0%	3.0%	1.0%
Self-Corrector	46.4%	47.0%	9.4%	8.8%	0.6%
ReVISE	48.6%	47.6%	3.2%	4.4%	−1.2%
Turn-RL	36.4%	55.4%	19.8%	0.8%	19.0%
CoCoS	48.2%	60.4%	13.0%	0.8%	12.2%

Table 10: Results on the MBPP dataset.

D.2 HumanEval

Method	Accuracy@t1	Accuracy@t2	$\Delta^{i\to c}$(t1, t2)	$\Delta^{c\to i}$(t1, t2)↓	Δ^{Acc}(t1, t2)
Qwen2.5-1.5B					
Boost model	24.4%	25.5%	1.1%	0.0%	1.1%
Turn-SFT	18.9%	20.1%	5.5%	4.3%	1.2%
Self-Corrector	23.2%	25.6%	6.1%	3.7%	2.4%
ReVISE	18.3%	15.9%	3.2%	4.5%	−1.3%
Turn-RL	27.4%	28.7%	1.8%	0.6%	1.2%
CoCoS	29.3%	32.3%	3.0%	0.0%	3.0%
Llama-3.2-1B					
Boost model	15.2%	15.2%	0.6%	0.6%	0.0%
Turn-SFT	14.6%	12.8%	1.8%	3.7%	−1.9%
Self-Corrector	11.0%	12.8%	4.9%	3.0%	1.9%
ReVISE	13.4%	11.9%	3.7%	5.6%	−1.9%
Turn-RL	8.5%	8.5%	0.0%	0.0%	0.0%
CoCoS	34.1%	39.6%	6.1%	0.6%	5.5%
deepseek-coder-1.3B					
Boost model	21.3%	21.3%	0.0%	0.0%	0.0%
Turn-SFT	19.5%	22.0%	5.5%	3.0%	2.5%
Self-Corrector	20.1%	22.6%	7.9%	5.5%	2.4%
ReVISE	23.8%	19.0%	3.1%	7.4%	−4.3%
Turn-RL	23.2%	25.6%	3.0%	0.6%	2.4%
CoCoS	22.6%	25.0%	3.0%	0.6%	2.4%

Table 11: Results on the HumanEval dataset.

D.3 ODEX

Method	Accuracy@t1	Accuracy@t2	$\Delta^{i\to c}$(t1, t2)	$\Delta^{c\to i}$(t1, t2)↓	Δ^{Acc}(t1, t2)
Qwen2.5-1.5B					
Boost model	21.2%	20.5%	0.0%	0.7%	−0.7%
Turn-SFT	25.5%	25.7%	0.5%	0.2%	0.3%
Self-Corrector	21.2%	27.1%	7.5%	1.6%	5.9%
ReVISE	18.5%	19.0%	8.7%	6.0%	0.5%
Turn-RL	10.9%	24.6%	14.1%	0.5%	13.6%
CoCoS	23.0%	28.9%	5.9%	0.0%	5.9%
Llama-3.2-1B					
Boost model	13.2%	12.8%	0.2%	0.7%	−0.5%
Turn-SFT	15.7%	15.3%	0.7%	1.1%	−0.4%
Self-Corrector	13.2%	14.4%	5.9%	4.8%	1.1%
ReVISE	9.6%	11.4%	4.8%	3.0%	1.8%
Turn-RL	19.8%	21.9%	2.1%	0.0%	2.1%
CoCoS	23.2%	25.1%	2.3%	0.5%	1.8%
deepseek-coder-1.3B					
Boost model	27.3%	27.6%	0.5%	0.2%	−0.3%
Turn-SFT	29.8%	28.7%	0.0%	1.1%	−1.1%
Self-Corrector	27.3%	28.0%	3.9%	3.2%	0.7%
ReVISE	21.2%	22.8%	8.1%	4.8%	1.6%
Turn-RL	23.6%	30.5%	6.9%	0.0%	6.9%
CoCoS	26.2%	31.4%	5.9%	0.7%	5.2%

Table 12: Results on the ODEX dataset.

E Additional SCoRe Experiment on Another Model

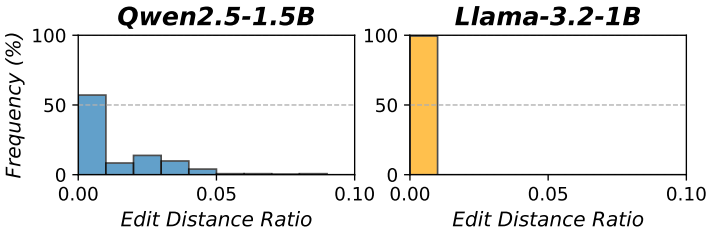

Figure 5: Edit distance ratio distributions between first and second responses for Qwen2.5-1.5B and Llama-3.2-1B in the 3-shot setting.

We further analyze the training trajectory of SCoRe using a small-scale Llama model in addition to the Qwen model. Prior to this, we measured the edit distance ratio. Following the same procedure as in §5.2, we compute the edit distance ratio for samples where the second response is incorrect. The results are shown in Figure 5. Remarkably, the Llama model exhibits an extreme tendency to make no changes to its responses, with a no-edit rate of 99%. We then compare the SCoRe training trajectories of Qwen and Llama in Figure 6. As Qwen has already been described in §5.2, we omit further discussion here. In the case of Llama, applying KL-regularization leads to a rapid convergence of the reward to zero, similar to what was observed with Qwen. Even without KL regularization, the Llama model eventually collapses. We speculate that this shows a limitation of training SLMs with binary rewards. When the policy is trained with binary rewards, the model receives only a pass or fail signal for $r(y_2, \hat{y}_2)$, which severely limits the space the policy can explore during training. As a result, in the case of SLMs that receive persistently low rewards, the policy ultimately collapses, failing to explore diverse solutions or improve over time. This is consistent with prior observations that sparse rewards make it challenging for RL to associate actions with rewards (Ladosz et al., 2022).

In contrast, CoCoS employs the same RLOO training algorithm (Ahmadian et al., 2024) as SCoRe but benefits from a progressive reward scheme. Under this setting, correctly solving even one more test case in a subsequent turn produces a positive reward, whereas making additional mistakes leads to a negative reward. This dynamic feedback encourages the expansion of the exploration space during training, enabling a more stable learning process. This progressive reward scheme encourages the expansion of the exploration space during training, enabling a more stable learning process.

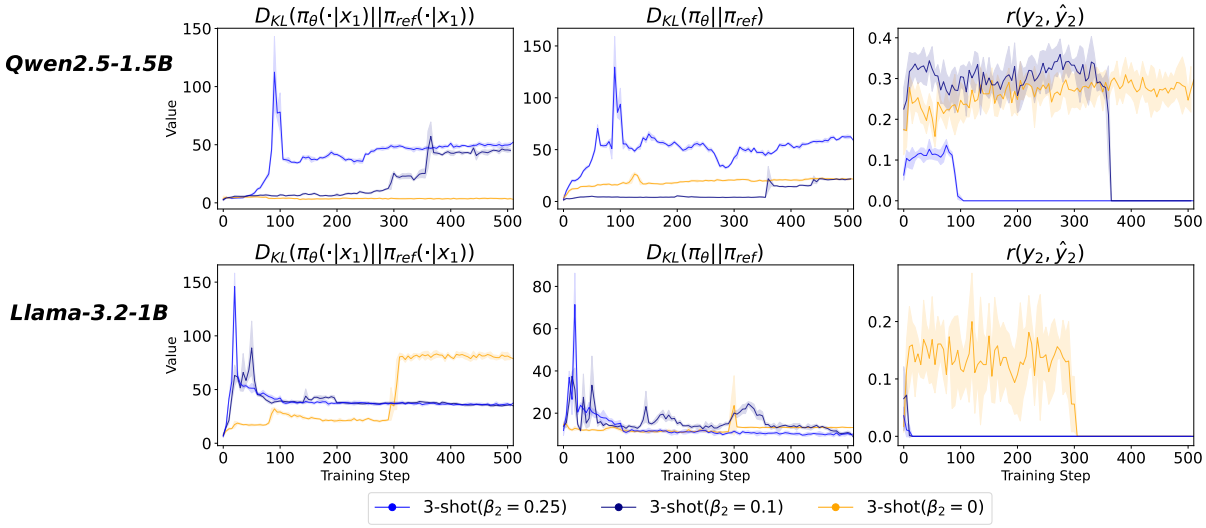

Figure 6: Learning curves for SCoRe, including additional results for the small-scale Llama model.

F Additional Case Studies

We present two additional case studies that illustrate distinct types of errors: (2) complex logic errors and (3) incorrect usage of code libraries. These additional cases serve to complement the discussion in the main text by illustrating error types that extend beyond naming issues. They underscore CoCoS's capacity to detect and remediate a diverse set of programming errors.

```
# Initial response
def amicable_numbers_sum(n):
    sum = 0
    for i in range(1, n+1):
        # just adds divisor sums
        sum += sum_of_divisors(i)
    return sum

def sum_of_divisors(num):
    divisors = [1]
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            divisors.append(i)
            if i!= num // i:
                divisors.append(num // i)
    return sum(divisors)
```

```
# Corrected response
def sum_of_divisors(n):
    divisors = [1]
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors.append(i)
            if i!= n // i:
                divisors.append(n // i)
    return sum(divisors)

def amicable_numbers_sum(n):
    sum = 0
    for i in range(1, n+1):
        div_sum = sum_of_divisors(i)
        # check if i and div_sum form an amicable pair
        if div_sum != i and sum_of_divisors(div_sum) == i:
            sum += i
    return sum
```

Code 2: Example of resolving a multi-function interaction error

Code 2 illustrates an error involving the coordination of multiple functions. The model initially fails to implement the correct interaction logic across functions, resulting in an incorrect solution. In the subsequent turn, it is able to revise the necessary parts while preserving the correct subcomponents, effectively resolving the issue.

```
# Initial response
def remove_dirty_chars(s1, s2):
    # Invalid in Python 3
    return s1.translate(None, s2)
```

```
# Corrected response
def remove_dirty_chars(s1, s2):
    result = ""
    for char in s1:
        if char not in s2:
            result += char
    return result
```

Code 3: Example of using a non-existent method

Lastly, Code 3 demonstrates a misuse of a Python method. The model generates an invalid call to `str.translate()` that is not supported in the current version of Python. In the subsequent turn, it corrects the implementation by replacing the unsupported method with a valid character filtering loop.

G Prompts

In this section, we present the prompts used for prompt-based approaches and for CoCoS training and evaluation. The prompts for implementing prompting-based approaches are described in §G.2 and §G.3. Finally, the prompts used for CoCoS training and evaluation are reported in §G.1.

G.1 Main experiments

We present the instructions for evaluating the MBPP, HumanEval, and datasets. Tokens such as [BEGIN], [DONE], and [CORRECT] were used to delimit model outputs, and the prompts were created from Kumar et al. (2025).

Prompt G.1: Prompt used for generating initial code responses in the first turn

```
You are an expert Python programmer, and here is your task: {{problem}} Your code should pass these
tests:

{{test cases}}

[BEGIN]
```

Prompt G.2: Prompt used for generating self-corrected responses

```
You are an expert Python programmer, and here is your task: {{problem}} Your code should pass these
tests:

{{test cases}}

[BEGIN]
{{initial code}}
[DONE]

There might be an error in the code above because of a lack of understanding of the question. Please
correct the error, if any, and rewrite the solution. Only output the final correct Python program!

[CORRECT]
```

G.2 Self-Refine

The prompts were designed and implemented based on the manuscript, with Self-Refine (Madaan et al., 2023) using a 3-shot prompting for both feedback generation and correction. Additionally, as the implementation was conducted without further fine-tuning, backticks such as ```` ```python ```` and ```` ``` ```` were used to delimit model outputs.

Prompt G.3: Prompt used for generating feedback in Self-Refine

````
I have some code. Can you give one suggestion to improve the solution to the problem. Don't fix the
code, just give one suggestion.

Problem:
{{problem}}

Code:
```python
{{initial code}}
```

Feedback:
````

Prompt G.4: Prompt used for self-correction in Self-Refine

````
I have some code. Can you give one suggestion to improve the solution to the problem. Don't fix the
code, just give one suggestion.

Problem:
{{problem}}

Code:
```python
{{initial code}}
```

Feedback:
{{feedback}}

Now fix the code.

Fixed Code:
```python
````
````

G.3 SETS

SETS (Chen et al., 2025) employs test-time scaling laws (Snell et al., 2025) for self-correction, which precludes direct evaluation of Accuracy@t1 in §4. Accordingly, we approximate it by reporting the average accuracy of the sampled initial responses. We use a sampling temperature of 0.7 and set the number of generation to 20. To distinguish Python code from instructions, we used backticks, while the remaining instructions were created based on the manuscript.

Prompt G.5: Prompt used for verifier in SETS

```
You are an expert in solving coding problems. You are given a PROBLEM and a PROPOSED CODE. Your job
is to:
1. Transform the PROPOSED CODE into a statement given the PROBLEM and identify all constraints in
the PROBLEM for verifying the statement.
2. Think step by step to verify if the statement satisfies each of the constraints.
3. Write a line of the form "The statement is correct" or "The statement is incorrect" at the end
of your response based on your analysis.

PROBLEM:
{{problem}}

PROPOSED CODE:
```python
{{initial code}}
```

ANALYSIS:
```

Prompt G.6: Prompt used for self-correction in SETS

```
You are an expert in solving coding problems. You are given a PROBLEM and a set of CODE-ANALYSIS
pairs. Your job is to generate a correct answer.

PROBLEM:
{{problem}}

CODE:
```python
{{initial code}}
```

ANALYSIS:
{{analysis}}

CORRECTED CODE:
```python
```
```

H Auxiliary instruction

Prompt H.1: Templates for the five rephrased instruction prompts.

```
Original:
There might be an error in the code above because of a lack of understanding of the question. Please
correct the error, if any, and rewrite the solution. Only output the final correct Python program!

Insturction 1:
There might be an error in the code above because of a lack of understanding of the question. Please
correct the error, if any, and rewrite the solution. Only output the final correct Python program!

Instruction 2:
If the solution above contains any mistakes due to a misunderstanding of the problem, fix them and
rewrite the code. Only return the corrected Python program.

Instruction 3:
Check the code for potential errors caused by misinterpreting the task.  Correct any issues and
output just the final Python solution.

Instruction 4:
Review the code for possible bugs or logic errors. If needed, fix them and provide only the updated
Python code as your answer.

Instruction 5:
Make sure the code fully matches the problem description. If any part is incorrect, fix it and return
just the corrected Python code.
```