

ReflexiCoder: Teaching Large Language Models to Self-Reflect on Generated Code and Self-Correct It via Reinforcement Learning

Anonymous ACL submission

Abstract

Large language models (LLMs) have demonstrated remarkable proficiency in code generation yet still frequently fail to produce correct solutions for complex programming tasks in a single attempt. While prior works attempt to mitigate this by incorporating external feedback such as execution results, these approaches suffer from heavy dependencies on environmental interaction and fail to cultivate the intrinsic debugging capabilities of the model. In this work, we propose ReflexiCoder-8B, a novel reinforcement learning (RL) framework that empowers models to autonomously self-reflect on their generated code and perform self-correction without reliance on external oracles. ReflexiCoder first produces an initial solution, then repeatedly reviews the previously generated code, performs bug and optimization aware reflection, and conditionally rewrites the program until no issues are found or a maximum number of rounds is reached. To strictly enforce this behavior, we formulate the process as a structured trajectory and optimize it with RL to align the model with effective self-reflection and self-correction trajectories, guided by a reward function specifically designed to value accurate error detection and successful repair. Extensive experiments on seven widely used benchmarks demonstrate that ReflexiCoder-8B achieves state-of-the-art results among open-source code models, reaching **94.51%** and **87.20%** on HumanEval and HumanEval Plus, **81.80%** and **78.57%** on MBPP and MBPP Plus, **35.00%** on BigCodeBench, **52.21%** on LiveCodeBench, and **37.34%** on CodeForces. Furthermore, our ReflexiCoder-8B is competitive with strong proprietary GPT-5.1 model on the first five benchmarks and surpasses it on complex reasoning benchmarks including LiveCodeBench and CodeForces. To facilitate future research, we release our source code at <https://anonymous.4open.science/r/ReflexiCoder-EF3C>.

1 Introduction

Large Language Models (LLMs) have revolutionized software engineering, demonstrating exceptional proficiency in translating natural language specifications into executable code (Chen et al., 2021; Dakhel et al., 2023; Jiang et al., 2024). Despite these advancements, standard “System 1” approaches which generate solutions in a single forward pass face an inherent ceiling when tackling complex, multi-step algorithmic problems (Li et al., 2022; Chen et al., 2024; Bairi et al., 2023; Zhong et al., 2024). In intricate scenarios typical of competitive programming or enterprise-level development, even state-of-the-art models frequently produce plausible-looking but functionally incorrect code on their first attempt.

To mitigate this limitation, recent studies have largely pivoted towards iterative refinement strategies. These can be broadly categorized into three paradigms: (1) Re-ranking, which samples multiple candidates to select the best one (Shi et al., 2022; Li et al., 2022; Chen et al., 2022; Zhang et al., 2023b; Ni et al., 2023); (2) External Repairers, utilizing separate models to patch errors (Gupta et al., 2020; Jiang et al., 2023a; Zhang et al., 2023a); and (3) Feedback-Guided Refinement, which relies on signals from execution environments (*e.g.*, compilers, unit tests) or human annotators to iteratively improve code (Chen et al., 2024; Jiang et al., 2023b; Zhong et al., 2024; Shinn et al., 2024). While effective, these methods suffer from a critical bottleneck: *dependency on external oracles and environmental interaction*. In real-world software development, comprehensive unit tests are often absent, and high-quality human feedback is costly and unscalable. Furthermore, relying heavily on external feedback prevents models from cultivating intrinsic debugging capabilities, the ability to “read”, scrutinize, and correct one’s own logic before execution.

Inspired by the recent success of reasoning-

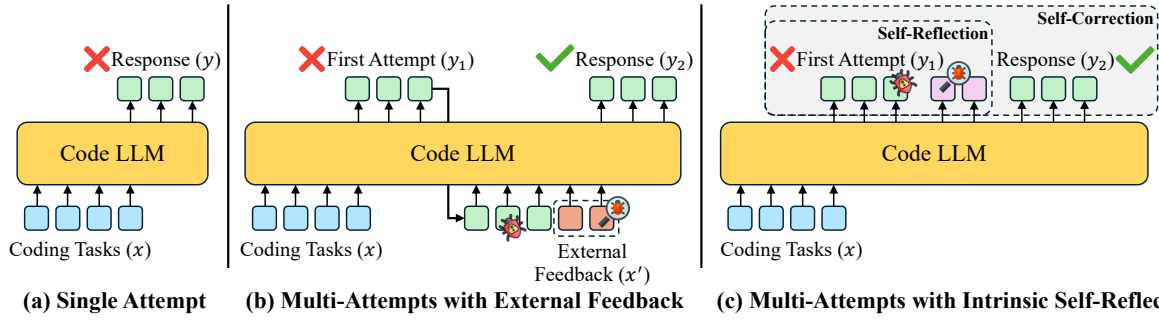


Figure 1: A comparative overview of code refinement workflows. (a) Existing code LLMs often struggle to generate correct solutions for complex programming tasks on a single attempt. (b) Prior practices mitigate this by relying on external feedback (e.g., compilers or human oracles). (c) Our proposed ReflexiCoder fosters an intrinsic capability to self-reflect and self-correct via a structured reasoning trajectory, eliminating the need for external supervision.

intensive models like OpenAI o1 (Jaech et al., 2024; Qin et al., 2024) and DeepSeek-R1 (Guo et al., 2025), which utilize extended inference time to facilitate deeper reasoning, we propose that code generation models should similarly possess an autonomous “inner monologue” for debugging. We introduce ReflexiCoder, a novel Reinforcement Learning (RL) framework designed to teach LLMs to self-reflect on generated code and perform self-correction without reliance on ground-truth feedback or external execution engines. Figure 1 provides a comparative overview of this paradigm shift from prior practices.

Unlike previous methods that treat generation and correction as separate supervised tasks, ReflexiCoder formulates the entire debugging process, generating an initial solution, reviewing the code for bugs/optimization, and rewriting, as a structured decision-making trajectory. We optimize this trajectory directly using RL. This approach aligns the model with effective self-reflection and self-correction paths guided by a specialized reward function that values both accurate error detection and successful repair. Crucially, we adopt an RL-zero training paradigm, bypassing the traditional supervised fine-tuning (SFT) warm-up stage (Guo et al., 2025; Li et al., 2025). We hypothesize that supervising weaker models with trajectories distilled from stronger teacher models (e.g., GPT-5 (OpenAI, 2025)) may inadvertently suppress the student’s exploration of correction strategies that best suit its own capabilities (Wang et al., 2022; Wei et al., 2024; Yuan et al., 2024; Kumar et al., 2024). By optimizing via RL directly, ReflexiCoder autonomously discovers efficient reflection patterns tailored to its parameter space.

Our extensive experiments on seven diverse benchmarks demonstrate the efficacy of this

paradigm. ReflexiCoder-8B establishes new state-of-the-art performance among leading open-source models, achieving **94.51%** on HumanEval (Chen et al., 2021), **35.00%** on BigCodeBench (Zhuo et al., 2024), **52.21%** on LiveCodeBench (Naman Jain et al., 2024), and **37.34%** on CodeForces (Quan et al., 2025). Remarkably, despite having significantly fewer parameters, our model remains competitive with the proprietary GPT-5.1 on standard benchmarks and surpasses it on complex reasoning tasks including LiveCodeBench (**52.21%** vs. **48.03%**) and CodeForces (**37.34%** vs. **34.70%**). These results suggest that cultivating intrinsic self-reflection and self-correction is a scalable and potent direction for the next generation of code LLMs. In summary, our main contributions are as follows:

- We propose ReflexiCoder, an RL-based framework that empowers LLMs to autonomously detect and correct coding errors through intrinsic self-reflection, eliminating the dependency on external feedback.
- We formulate the self-reflection and self-correction loop as a multi-step trajectory and optimize it via RL. By designing a granular reward function that targets both reflection quality and repair success, we enable the model to incentivize intrinsic debugging strategies.
- Our ReflexiCoder-8B significantly outperforms leading open-source models across seven benchmarks. It surpasses proprietary models like GPT-5.1 on challenging competitive programming tasks including LiveCodeBench and CodeForces.
- We release our model, code, and data to facilitate reproducibility and foster future research into the self-improving capabilities of LLMs.

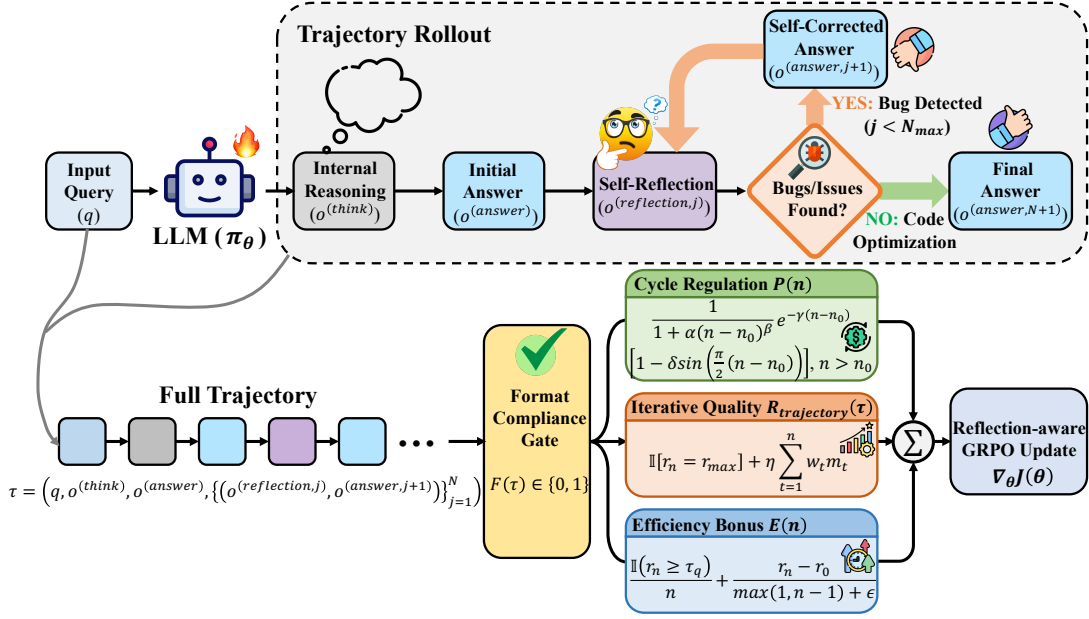


Figure 2: The architecture of ReflexiCoder formulates code generation as an RL-optimized intrinsic self-debugging trajectory. A carefully designed composite reward jointly incentivizes reflection quality and repair success.

2 Methodology

In this section we formalize the proposed ReflexiCoder training pipeline, which integrates a structured self-reflection and self-correction mechanism into large language models (LLMs) and optimizes their behavior via reinforcement learning.

2.1 Structured Reasoning-Reflection Process

Let $q \in \mathcal{Q}$ denote a programming-related query, and let an LLM parameterized by θ produce a sequence of textual outputs in structured segments

$$o = (\underbrace{o^{(\text{think})}}_{\text{reasoning}}, \underbrace{o^{(\text{answer})}}_{\text{answer}}), \quad (1)$$

$$\underbrace{\{(o^{(\text{reflection},j)}, o^{(\text{answer},j+1)})\}_{j=1}^n}_{n \text{ reflection cycles}}$$

where $n \in \mathbb{N}$ denotes the number of reflection iterations. The internal reasoning segment $o^{(\text{think})}$ may be unobservable to the end-user yet available to the training environment for evaluation, ensuring the policy incorporates invisible planning before producing visible answers. Each reflection-answer pair is constrained to be contiguous and well-formed according to a global format specification \mathcal{F} .

We model the full trajectory corresponding to one prompt-response interaction as

$$\tau \equiv \left(q, o^{(\text{think})}, o^{(\text{answer})}, \{(o^{(\text{reflection},j)}, o^{(\text{answer},j+1)})\}_{j=1}^n \right) \quad (2)$$

and define the set of all format-compliant trajectories as $\mathcal{T}_{\text{valid}} = \{\tau \in \mathcal{T} \mid \Phi(\tau) = \mathcal{F}^*\}$ with $\Phi(\cdot)$ denotes syntax extractor and \mathcal{F}^* the target global format specification, which will be strictly enforced in reward computation.

2.2 Iterative Reflection Rewards

Format Compliance Constraints. A fundamental prerequisite for our reinforcement learning setup is that the model’s outputs conform exactly to a pre-determined structural format. Each generated response must consist of a distinct internal reasoning segment, a user-facing answer, and a reflection-answer pair for every revision, with the reflection and its subsequent revised answer always appearing together. *Additional revision pairs are permitted only when prior reflection identifies new issues, and the total number of reflections must not exceed the specified global limit.*

This structure is not a superficial constraint that our reward mechanisms rely on being able to unambiguously identify each reasoning step, every answer, and the corresponding reflection. Deviations, such as missing segments, incorrect ordering, or unmatched reflection-answer pairs, break the parsing pipeline and undermine the core iterative improvement process. To enforce strict adherence, we introduce a format compliance reward $F(\tau)$:

$$F(\tau) = \mathbb{I}[\tau \in \mathcal{T}_{\text{valid}}] \Rightarrow F : \mathcal{T} \rightarrow \{0, 1\}. \quad (3)$$

This binary reward $F(\cdot)$ acts as a gating factor that if $F(\tau) = 0$, the total reward for the trajectory is

zero, irrespective of content quality. Only trajectories that satisfy the format constraint are eligible for further quality-related reward shaping.

Once format compliance is guaranteed, the reward model incorporates three complementary components, including a smoothly decaying penalty for excessive reflection cycles, a trajectory improvement term that emphasizes progressive quality gains, and an efficiency bonus that rewards significant improvement with minimal iteration.

Cycle Count Regulation. Reflection cycles inherently present a trade-off between depth and efficiency. Empirically, one to three cycles often yield substantial benefits, such as clarity enhancement, logical coherence, and factual accuracy. Beyond that, gains diminish, and the LLM may waste computational effort or even regress. Let $n \in \mathbb{N}$ denote the total number of reflection cycles, and n_0 is the no-penalty depth. When $1 \leq n \leq n_0$, we apply no penalty ($P(n) = 1$), preserving the freedom to engage in “reasonable depth” revision. For $n > n_0$, rewards ($P(n) \in (0, 1]$) are multiplicatively attenuated by a composite decay term:

$$P(n) = \begin{cases} 1, & 1 \leq n \leq n_0, \\ \frac{1}{1 + \alpha(n - n_0)^\beta} \cdot e^{-\gamma(n - n_0)} & n > n_0 \\ \cdot \left[1 - \delta \sin\left(\frac{\pi}{2}(n - n_0)\right) \right], & \end{cases} \quad (4)$$

where $\alpha > 0$, $\beta > 1$ control polynomial decay strength, $\gamma > 0$ governs exponential attenuation, and $\delta \in (0, 0.3)$ introduces a mild oscillatory perturbation to encourage exploration of slightly different iteration depths.

Iterative Quality Improvement. Beyond regulating cycle count, the learning objective needs to explicitly encourage sustained improvement in the quality of generated answers. We denote the trajectory of quality scores as $\mathbf{r} = (r_0, r_1, \dots, r_n) \in \mathbb{R}^n$, where r_t represents the quality score of the t -th solution obtained through automated execution and validation. Ideally, the optimal trajectory should satisfy $r_0 \leq r_1 \leq \dots \leq r_n$, reflecting a progressive improvement in code quality. To emphasize the importance of later improvement stages within a trajectory, we apply exponential time-weighting

$$w_t = \frac{e^{\lambda t}}{\sum_{k=1}^n e^{\lambda k}}, \quad \lambda > 0 \quad (5)$$

which yields a normalized vector $\mathbf{w} = (w_1, w_2, \dots, w_n) \in \Delta^n$ over the probability sim-

plex in \mathbb{R}^n , with the parameter λ controlling the degree to which later iterations are prioritized. The resulting weights satisfy $w_1 < w_2 < \dots < w_n$, thereby favoring improvements occurring in later stages.

A central challenge in iterative refinement lies in designing a reward signal that captures not only the absolute quality of each answer but also the trajectory’s progression. Let $\Delta r_t = r_t - r_{t-1}$ for $t \geq 1$ denote the gains in quality between successive answers. We define the improvement signal m_t using a piecewise formulation:

$$m_t = \begin{cases} +f\left(\frac{\Delta r_t}{s}\right) & \Delta r_t > 0, \\ +h_{pos}|\Delta r_t| < \varepsilon \text{ and } |r_{t-1} - r_{max}| < \varepsilon, & \\ -g\left(\frac{|\Delta r_t|}{s}\right) & \Delta r_t < 0, \\ -h_{neg} & |\Delta r_t| < \varepsilon \text{ and } r_{t-1} < r_{max} \end{cases} \quad (6)$$

where $s > 0$ controls the sensitivity to quality changes, ε is a small tolerance for numerical stability, r_{max} denotes the maximum achievable score, and $h_{pos} > 0$ and $h_{neg} > 0$ are constants used to handle stagnation: h_{pos} provides a bonus when the score has effectively converged near r_{max} , while h_{neg} imposes a penalty when the answer stagnates below r_{max} . We adopt $\tanh(\cdot)$ for $f(\cdot)$ and $g(\cdot)$ as it provides a smooth mapping from raw score differences to bounded rewards, which facilitates stable policy optimization.

The trajectory-level reward is then defined as

$$R_{\text{trajectory}}(\tau) = \underbrace{\mathbb{I}[r_n = r_{max}]}_{\text{final solution}} + \eta \underbrace{\sum_{t=1}^n w_t m_t}_{\text{quality improv.}} \quad (7)$$

where $r_{max} = 1$ means code passes all tests, $\eta > 0$ adjusts the contribution of the improvement signal relative to the absolute quality score.

Notably, this reward design provides positive reinforcement for quality gains, penalizes declines, suppresses stagnation when improvement is still possible, and avoids penalizing the absence of change when the quality is already optimal. Detailed principles motivating this design are provided in Appendix C.

Efficiency Reward. However, solely combining $P(n)$ and $R_{\text{trajectory}}$ may lead to undesirable behaviors that the model might overfit to a fixed n , ignore task difficulty, or become hypersensitive to noise in r_t . Strong penalties could discourage beneficial exploration, and credit assignment over long horizons

remains problematic. To counter these problems, we introduce an efficiency term:

$$E(n) = \underbrace{\frac{\mathbb{I}[r_n \geq \tau_q]}{n}}_{\text{absolute}} + \underbrace{\frac{r_n - r_0}{\max(1, n - 1) + \epsilon}}_{\text{relative}}, \epsilon > 0 \quad (8)$$

where $\mathbb{I}[\cdot]$ is the indicator function, τ_q denotes the required quality threshold, and ϵ prevents singularities. This term rewards average quality gain per reflection, encouraging policy to achieve maximal improvement with minimal steps.

Finally, the overall reward model is:

$$R_{\text{overall}}(\tau) = \mathbb{I}[F(\tau) = 1]P(n)(\varphi R_{\text{trajectory}}(\tau) + \psi E(n)) + \xi F(\tau), \quad (9)$$

where φ , ψ and ξ control trajectory quality, efficiency bonus, and formatting constraints, respectively. The reward surface R_{overall} therefore enforces $\tau \in \mathcal{T}_{\text{valid}}$ and balances *progressive refinement* $R_{\text{trajectory}}$ and *economy of iterations* $E(n)$, formalizing the self-reflection and self-correction objectives in a mathematically explicit manner.

In practice, this integrated reward landscape allows the learning process to internalize both how to think more effectively across iterations and when to stop, achieving a disciplined reflection mechanism aligned with the overarching objectives of human.

2.3 Reflection-aware GRPO

We adopt GRPO objective (Guo et al., 2025) for policy π_θ updates, which replaces the value function $V^\pi(s)$ with a group-normalized advantage estimate $\hat{A}(s, a)$, enhancing stability and reducing variance in large action spaces \mathcal{A} . The detailed formulation is provided in Appendix D.

3 Experiments

3.1 Experimental Settings

Models and Benchmarks We instantiate our model by fine-tuning a recent open-source backbone, Qwen3-8B (Yang et al., 2025), to obtain our **ReflexiCoder-8B**. We evaluate performance across a diverse set of seven code generation benchmarks, ranging from foundational programming challenges such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and EvalPlus (Liu et al., 2023), to significantly more complex and competitive programming problems found in BigCodeBench (Zhuo et al., 2024), LiveCodeBench (Jain et al., 2024), and CodeForces (Quan et al.,

2025). The entire evaluation pipeline is managed by the EvalChem framework (Raouf et al., 2025).

Implementation Details We implement our RL pipeline using the Open-R1 framework (Hugging Face, 2025)¹. We train the model for two epochs using our curated open-source dataset of programming problems, derived from (Luo et al., 2025). The detailed dataset curation can be found in Appendix E.2. The no-penalty reflection depth n_0 was set to 5. For the cycle count penalty $P(n)$, we used $\alpha = 0.1$, $\beta = 2.0$, $\gamma = 0.05$, and $\delta = 0.1$. The exponential weighting for trajectory improvement $R_{\text{trajectory}}(\tau)$ used $\lambda = 0.2$, with the improvement signal weight $\eta = 0.5$. The main reward component weights are $\varphi = 0.5$ for trajectory quality $R_{\text{trajectory}}(\tau)$, $\psi = 1.0$ for the efficiency bonus $E(n)$, and $\xi = 1.0$ for formatting constraints $F(\tau)$. All experiments are conducted on a cluster of $64 \times$ NVIDIA H200 GPUs with a per-device batch size of 1. More details are provided in Appendix E.

3.2 Main Results

We evaluate our ReflexiCoder-8B across seven representative benchmarks to assess its code generation and self-correction capabilities. Table 1 summarizes the pass@1 performance compared to state-of-the-art (SOTA) proprietary and open-source models. ReflexiCoder-8B establishes a new SOTA performance for open-source models in the 7B-8B parameter range. Compared to its backbone Qwen3-8B (Yang et al., 2025), our ReflexiCoder achieves substantial absolute gains by +5.49% on HumanEval, +14.46% on LiveCodeBench, and +13.64% on CodeForces. Notably, it outperforms specialized code models like Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) and Seed-Coder-8B-Instruct (Seed et al., 2025) by significant margins, particularly on reasoning-intensive benchmarks (LCB and CF). These gains demonstrate that our proposed RL paradigm, which optimizes the entire reflection-correction trajectory, is effective for cultivating deep algorithmic reasoning. A pivotal finding is that our ReflexiCoder-8B, despite its compact parameter scale, competes directly with and occasionally surpasses leading proprietary models. While maintaining parity with GPT-5.1 on HumanEval+ and MBPP+, ReflexiCoder-8B demonstrates superior performance on the most challenging benchmarks: it outperforms GPT-5.1 on LiveCodeBench (52.21% vs. 48.03%) and CodeForces

¹<https://github.com/huggingface/open-r1>

Table 1: Main results on seven code generation benchmarks, reporting pass@1 accuracy (%). We compare our ReflexiCoder against leading proprietary and open-source models. Our models establish a new state-of-the-art for open-source models and demonstrate competitive performance against much larger proprietary models. HE(+) denotes HumanEval(+). BCB is BigCodeBench, LCB is LiveCodeBench, and CF is CodeForces

Institution	Model	HE	HE+	MBPP	MBPP+	BCB	LCB	CF
OpenAI	GPT-5.1	95.12	87.20	84.00	79.10	39.56	48.03	34.70
	GPT-4.1	96.34	78.88	85.20	79.10	41.32	42.77	31.37
Anthropic	Claude-Sonnet-4.5	98.17	77.44	76.80	75.40	45.00	50.78	53.79
Google	Gemini-2.5-Pro	97.56	92.07	94.20	84.39	41.32	62.01	52.40
Alibaba	Qwen3-8B	89.02	80.49	70.20	70.37	32.63	37.75	23.70
	Qwen2.5-Coder-7B-Instruct	86.59	79.88	75.80	69.84	33.33	13.86	6.39
ByteDance	Seed-Coder-8B-Instruct	85.98	80.49	68.40	72.49	36.05	21.03	4.09
DeepSeek	DeepSeek-Coder-7B-Instruct	73.17	67.07	65.00	63.76	27.02	8.24	2.56
Google	CodeGemma-7B-IT	54.88	41.46	53.20	54.76	25.44	8.12	1.88
Meta	CodeLlama-7b-Instruct	30.49	26.22	2.40	3.97	16.58	1.19	0.68
Ours	ReflexiCoder-8B	94.51 ^{+5.49}	87.20 ^{+6.71}	81.80 ^{+11.6}	78.57 ^{+8.20}	35.00 ^{+2.37}	52.21 ^{+14.46}	37.34 ^{+13.64}

Table 2: Ablation study of reward components. Performance is reported as pass@1 (%) on four benchmarks. The results confirm that each component is vital for achieving optimal performance and behaviors.

Method	HumanEval	BigCodeBench	LiveCodeBench	CodeForces
ReflexiCoder-8B (Full)	94.51	35.00	52.21	37.34
w/o Format Gating $F(\tau)$	84.75	32.02	39.07	24.81
w/o Cycle Regulation $P(n)$	92.68	33.68	52.09	35.84
w/o Efficiency Reward $E(n)$	91.46	33.42	42.41	29.92
w/o Progressive Imp. m_t	93.29	34.74	39.19	34.10

(37.34% vs. 34.70%). These results are particularly salient as LCB and CF consist of high-complexity problems that typically require multiple logical iterations. This performance leap stems from our granular reward function, which incentivizes the model to detect its own errors and refine outputs autonomously. By eliminating the bottleneck of external execution feedback, ReflexiCoder proves that intrinsic self-reflection and self-correction is not only a scalable path toward reliable code generation but also the key to mastering complex algorithmic reasoning.

3.3 In-depth Analysis and Insights

In this section, we conduct a series of deeper analyses to understand the emergent behaviors and scalability of our ReflexiCoder.

Ablation Study We conduct an ablation study to deconstruct the influence of each component in our composite reward function R_{overall} . We train four variants of ReflexiCoder where each variant removes one term from the full reward formulation in Equation 9: (1) Remove Format Gating $F(\tau)$ to quantify the importance of enforcing a strict reasoning-reflection-answer format for the learning process. (2) Remove Cycle Regulation $P(n)$

to test our hypothesis that without regulation, the model may indulge in computationally wasteful or even counter-productive deep reflection, failing to learn when to terminate the process. (3) Remove Efficiency Reward $E(n)$ to investigate its role in encouraging the model to make more substantial corrections in fewer steps. (4) Remove Progressive Improvement m_t from $R_{\text{trajectory}}(\tau)$, relying solely on the absolute quality scores r_t , to verify that explicitly rewarding the positive delta in quality is crucial for guiding the model towards a monotonically improving trajectory. As shown in Table 2, the results demonstrates that each component of our reward designs plays a distinct and indispensable role in achieving the optimal performance and behaviors.

Impact of Reasoning & Reflection To investigate the source of ReflexiCoder’s performance gains, we conduct a comparative analysis against three critical baselines: (1) Baseline, the backbone model without any additional fine-tuning; (2) No Reasoning Pattern, where the backbone model with non-thinking mode; and (3) Vanilla Outcome-RL, which optimizes the model using a binary pass/fail reward signal without incentivizing the intermediate reflection trajectory. The results are

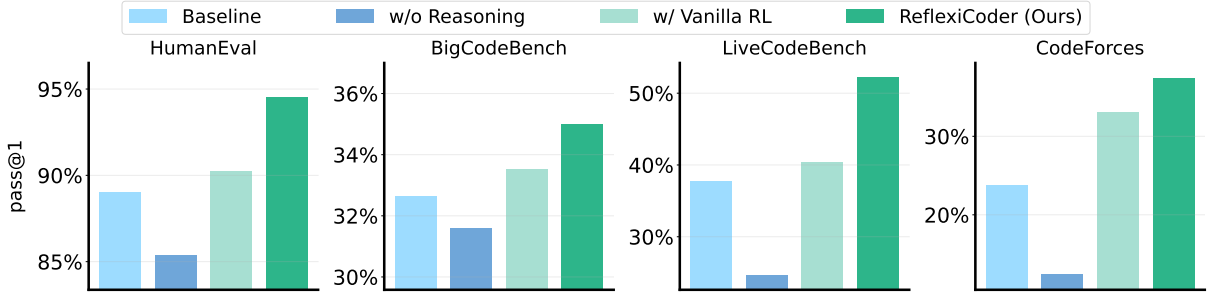


Figure 3: Analysis of the impact of reasoning and reflection. We compare ReflexiCoder against baselines that lack a structured reasoning and reflection step. Performance is pass@1 (%). The significant performance gap highlights that the structured reasoning-reflection cycle is the key driver of improvement.

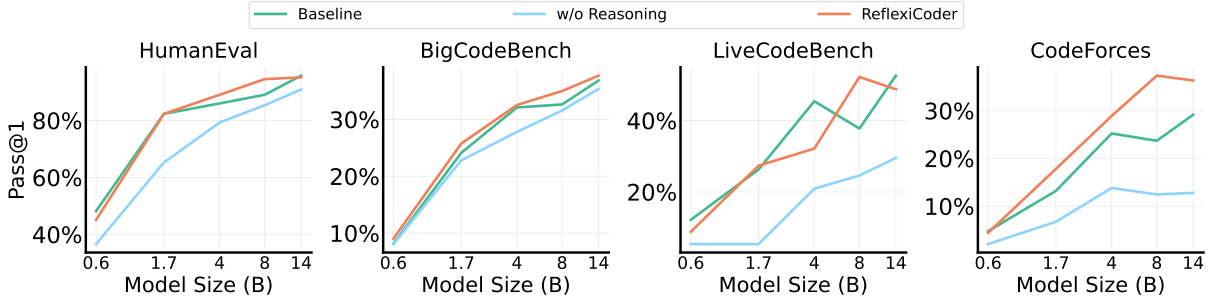


Figure 4: Scaling analysis of ReflexiCoder with model size. Performance is pass@1 (%) across four representative benchmarks. The performance grow with model scale, indicating a super-linear benefit.

summarized in Figure 3. The substantial performance gains of ReflexiCoder over both the *Non-Reasoning* and *Vanilla RL* baselines with absolute average improvements of 18.64% and 5.22%, respectively, across three reasoning-intensive benchmarks demonstrate that the model’s success is not merely a byproduct of reasoning and RL, but rather stems from the structured self-reflection and self-correction process. This suggests that our ReflexiCoder has developed an intrinsic debugging capability that mimics human-like cognitive oversight.

Scalability with Model Size We evaluate scalability by training ReflexiCoder on Qwen3 backbones from 0.6B to 14B. Figure 4 reports average pass@1 over HumanEval, BigCodeBench, LiveCodeBench, and CodeForces for the base model, a *no-reasoning* variant, and our RL-trained ReflexiCoder. Across all sizes, *no-reasoning* consistently underperforms, with the largest drops on LiveCodeBench and CodeForces, highlighting that intermediate reasoning is critical for algorithmic planning and bug detection. In contrast, our ReflexiCoder yields larger gains as model size increases, with especially strong improvements on reasoning-intensive benchmarks. This supports our key claim that optimizing the full “generate, reflect, correct” trajectory with RL teaches intrinsic self-correction, and larger models can internalize this policy more

effectively. Training curves in Appendix Figure 7 further show faster optimization of $R_{\text{trajectory}}(\tau)$ and $E(n)$ for larger models, indicating more effective and efficient reflection.

Hyperparameter Analysis We evaluate the sensitivity of ReflexiCoder to the reward weights in Eq. 9: trajectory quality φ , efficiency bonus ψ , and format constraint ξ . Figure 5 shows that performance is robust across a wide range of settings, indicating a stable RL objective. The best overall configuration is $\langle \xi, \varphi, \psi \rangle = \langle 1.0, 0.5, 1.0 \rangle$, which yields the strongest and most consistent results across all benchmarks, with particularly large gains on reasoning-intensive tasks (LiveCodeBench and CodeForces). Notably, overly large φ degrades performance, suggesting that the improvement comes from reward-aligned, effective self-reflection that enables repair, rather than longer or more elaborate reflection. Increasing ψ generally helps, supporting the need to explicitly encourage efficient multi-step self-correction.

3.4 Case Study

To provide qualitative insight into the self-correction process, we conduct a case study on a challenging problem from the TACO benchmark. We sample a trajectory generated by ReflexiCoder-8B and manually annotate the errors in the ini-

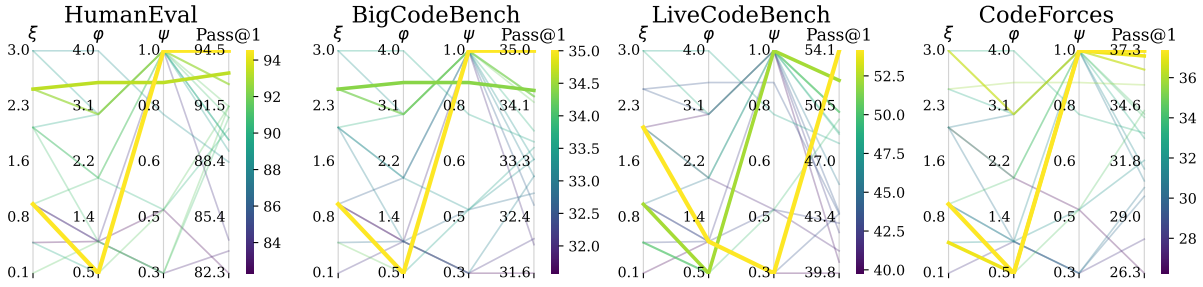


Figure 5: Pass@1 sensitivity to reward weights Format (ξ), Iter. Quality (φ), Efficiency (ψ) on four benchmarks.

Case Study: Self-Reflection and Correction on a TACO Problem

Problem: Given an array a of length n and an integer k , find the number of indices $1 \leq i \leq n - k$ such that the subarray $[a_i, \dots, a_{i+k}]$ with length $k + 1$ (not with length k) has the following property: ... More formally, count the number of indices $1 \leq i \leq n - k$ such that $2^0 \cdot a_i < 2^1 \cdot a_{i+1} < 2^2 \cdot a_{i+2} < \dots < 2^k \cdot a_{i+k}$.

Cycle 0: Reasoning and Generation

```
<think>Okay, I need to solve this programming problem. Let's read the question carefully...</think>
<answer>
```

```
...
        if val < (a[i + j - 1] * (1 << (j - 1))):
...
</answer>
```

Cycle 1: Reflection and Correction (BUG DETECTED)

```
<reflection>STATUS: BUG_DETECTED
The logic inside the inner loop currently uses the condition if val < (a[i + j - 1] * (1 << (j - 1))) to invalidate a
sequence. This means that if val is equal to the previous value, valid remains True, effectively checking for a
non-decreasing sequence. However, the problem logic typically requires a strictly increasing sequence, meaning the current
value must be strictly greater than the previous one. Therefore, the condition should be if val <= ... to ensure
valid = False when values are equal or smaller.
</reflection>
```

```
<answer>
...
        # Corrected: Strictly increasing check
        if val <= (a[i + j - 1] * (1 << (j - 1))):
...
</answer>
```

Cycle 2: Reflection and Correction (CODE OPTIMIZATION)

```
<reflection>STATUS: OPTIMIZATION_ONLY
The code is functionally correct, but it can be optimized for better readability and efficiency. The nested loops result in a
time complexity of  $O(t * n * k)$ , which may be too slow for large values of  $n$  and  $k$ . To optimize, we can precompute the power
of two values and avoid redundant calculations. Additionally, using a more efficient approach for checking the conditions can
reduce the time complexity. These changes will not affect the correctness of the algorithm.
</reflection>
```

```
<answer>
...
        # Precompute powers of two up to k
        powers_of_two = [1 << j for j in range(k + 1)]
...
</answer>
```

Figure 6: A qualitative example of ReflexiCoder’s iterative self-reflection and self-correction on a TACO task.

495 tial solution and the corrections made in subse-
496 quent reflection cycles. As illustrated in Figure
497 6, in Cycle 0, the model produces an initial brute-
498 force implementation to count valid subarrays under
499 the 2^j scaling constraint. In Cycle 1, self-
500 reflection detects a correctness bug: the check
501 mistakenly allows equal consecutive scaled val-
502 ues (non-decreasing), and is corrected to enforce
503 strict increase by changing $<$ to $<=$. In Cycle 2,
504 the model performs an optimization-only revision
505 by precomputing powers of two and reducing red-
506 undant computations, improving efficiency and
507 readability while preserving correctness.

4 Conclusion

508
509 In this work, we propose ReflexiCoder, a frame-
510 work that leverages reinforcement learning to teach
511 LLMs intrinsic self-reflection and self-correction
512 without relying on external oracles. By formulating
513 debugging as a trainable decision-making trajec-
514 tory, our ReflexiCoder-8B achieves state-of-the-art
515 performance among open-source models and sur-
516 passes proprietary models like GPT-5.1 on complex
517 reasoning benchmarks. These results demonstrate
518 that optimizing internal debugging capabilities via
519 RL is a scalable and effective direction for the next
520 generation of reliable code LLMs.

5 Limitations

Our proposed ReflexiCoder improves reliability by allocating multiple reflection and correction cycles. Even with cycle regulation and efficiency bonuses, the method increases token usage and latency compared to single pass generation. This trade-off can limit applicability in tight latency settings, and the optimal reflection budget may vary by task difficulty in ways that are hard to predict a priori. The proposed intrinsic debugging primarily targets algorithmic correctness and local code issues within a single file setting. It does not explicitly address repository level development, long horizon refactoring, dependency management, or interactive debugging with evolving specifications. Extending the trajectory formulation to multi-file contexts and richer tool interfaces remains future work. We instantiate ReflexiCoder on Qwen3 family models and evaluate on common Python centric benchmarks. While scaling trends are promising, it is unclear how well the same trajectory format and reward shaping transfer to other backbones, other programming languages, or domains where correctness cannot be captured by unit tests alone.

References

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*.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, and 1 others. 2023. Codeplan: Repository-level coding using llms and planning. *arXiv preprint arXiv:2309.12499*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*.

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 *Proceedings of the Twelfth International Conference on Learning Representations: ICLR*.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shiron Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695.

Hugging Face. 2025. [Open r1: A fully open reproduction of deepseek-r1](#).

Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023a. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE.

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023b. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, and

627	1 others. 2024. Training language models to self-correct via reinforcement learning. <i>arXiv preprint arXiv:2409.12917</i> .	
628		
629		
630	Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coder1: Mastering code generation through pretrained models and deep reinforcement learning. <i>Advances in Neural Information Processing Systems</i> , 35:21314–21328.	
631		
632		
633		
634		
635	Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. <i>Science</i> , 378(6624):1092–1097.	
636		
637		
638		
639		
640	Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, and 1 others. 2025. From system 1 to system 2: A survey of reasoning large language models. <i>arXiv preprint arXiv:2502.17419</i> .	
641		
642		
643		
644		
645		
646	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. <i>Advances in Neural Information Processing Systems</i> , 36:21558–21572.	
647		
648		
649		
650		
651	Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deepcoder: A fully open-source 14b coder at o3-mini level. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-o3-mini-Level-1cf81902c14680b3bee5eb349a512a51 . Notion Blog.	
652		
653		
654		
655		
656		
657		
658		
659		
660	King Han Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. <i>arXiv preprint arXiv:2403.07974</i> .	
661		
662		
663		
664		
665		
666	Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In <i>International Conference on Machine Learning</i> , pages 26106–26128. PMLR.	
667		
668		
669		
670		
671		
672	OpenAI. 2025. Gpt-5 is here .	
673	Yiwei Qin, Xuefeng Li, Haoyang Zou, Yixiu Liu, Shijie Xia, Zhen Huang, Yixin Ye, Weizhe Yuan, Hector Liu, Yuanzhi Li, and 1 others. 2024. O1 replication journey: A strategic progress report–part 1. <i>arXiv preprint arXiv:2410.18982</i> .	
674		
675		
676		
677		
678	Shanghaoran Quan, Jiayi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. <i>arXiv preprint arXiv:2501.01257</i> .	
679		
680		
681		
682		
683		
	Negin Raoof, Etash Kumar Guha, Ryan Marten, Jean Mercat, Eric Frankel, Sedrick Keh, Hritik Bansal, Georgios Smyrnis, Marianna Nezhurina, Trung Vu, Zayne Rea Sprague, Mike A Merrill, Liangyu Chen, Caroline Choi, Zaid Khan, Sachin Grover, Benjamin Feuer, Ashima Suvarna, Shiye Su, and 27 others. 2025. Evalchemy: Automatic evals for LLMs.	684 685 686 687 688 689 690
	Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. <i>arXiv preprint arXiv:2308.12950</i> .	691 692 693 694 695
	ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, and 1 others. 2025. Seed-coder: Let the code model curate data for itself. <i>arXiv preprint arXiv:2506.03524</i> .	696 697 698 699 700
	Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. <i>arXiv preprint arXiv:2204.11454</i> .	701 702 703 704
	Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. <i>Advances in Neural Information Processing Systems</i> , 36.	705 706 707 708 709
	Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. <i>Transactions on Machine Learning Research</i> .	710 711 712 713
	CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, and 1 others. 2024. Codegemma: Open code models based on gemma. <i>arXiv preprint arXiv:2406.11409</i> .	714 715 716 717 718 719
	Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Han-naneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. <i>arXiv preprint arXiv:2212.10560</i> .	720 721 722 723 724
	Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. <i>arXiv preprint arXiv:2410.24198</i> .	725 726 727 728 729
	An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. <i>arXiv preprint arXiv:2505.09388</i> .	730 731 732 733 734
	Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. <i>arXiv preprint arXiv:2401.10020</i> .	735 736 737 738

739 Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a.
740 Self-edit: Fault-aware code editor for code genera-
741 tion. *arXiv preprint arXiv:2305.04087*.

742 Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike
743 Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang.
744 2023b. Coder reviewer reranking for code generation.
745 In *International Conference on Machine Learning*,
746 pages 41832–41846. PMLR.

747 Li Zhong, Zilong Wang, and Jingbo Shang. 2024.
748 Ldb: A large language model debugger via verify-
749 ing runtime execution step-by-step. *arXiv preprint*
750 *arXiv:2402.16906*.

751 Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman,
752 Haohan Wang, and Yu-Xiong Wang. 2023. Lan-
753 guage agent tree search unifies reasoning acting
754 and planning in language models. *arXiv preprint*
755 *arXiv:2310.04406*.

756 Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu,
757 Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani
758 Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and
759 1 others. 2024. Bigcodebench: Benchmarking code
760 generation with diverse function calls and complex
761 instructions. *arXiv preprint arXiv:2406.15877*.

A Related Work

Iterative Refinement with External Feedback.

Recent advancements suggest that code generation is fundamentally an iterative process rather than a single-turn translation task. A prevalent paradigm involves refining initial outputs using feedback signals. Approaches such as Self-Debugging (Chen et al., 2024) and LDB (Zhong et al., 2024) prompt LLMs to revise code based on execution traces or unit test results. Similarly, SelfEvolve (Jiang et al., 2023b) and Reflexion (Shinn et al., 2024) utilize iterative loops where models generate feedback or consult external evaluators to guide rectification. While promising, these approaches predominantly rely on *extrinsic* signals, such as compilers, test suites, or separate critic models, to identify errors. LATS (Zhou et al., 2023) extends this by integrating Monte Carlo Tree Search (MCTS) with external evaluation values. A significant limitation of this paradigm is the dependency on high-quality external oracles, which are frequently unavailable or computationally expensive in real-world deployment. Unlike these methods, our ReflexiCoder is designed to internalize the debugging capability, enabling the model to self-correct based on intrinsic reflection without accessing external execution environments.

RL for Code and Reasoning. Reinforcement Learning (RL) has been widely adopted to align LLMs with functional correctness. CodeRL (Le et al., 2022) and PPOCoder (Shojaee et al., 2023) leverage actor-critic architectures (such as PPO) to optimize models using compiler feedback or unit test pass rates as reward signals. DeepCoder (Luo et al., 2025) further explores RL to enhance code generation. However, these methods typically treat the generation process as a black box, optimizing for the final output correctness via sparse binary rewards without explicitly incentivizing the intermediate reasoning or error-detection process. Recently, reasoning-oriented models like OpenAI o1 (Jaech et al., 2024; Qin et al., 2024) and DeepSeek-R1 (Guo et al., 2025) have demonstrated that extending inference time with intrinsic chain-of-thought (CoT) can significantly boost performance. While these models illustrate the potential of test-time scaling, supervising or incentivizing the specific structure of “self-reflection and self-correction” for code remains underexplored. ReflexiCoder addresses this gap by formulating the debugging loop as a structured trajectory and

optimizing it via RL, allowing the model to autonomously discover effective strategies for error localization and repair.

B Algorithm for ReflexiCoder

Algorithm 1 REFLEXICODER TRAINING VIA REFLECTION-AWARE GRPO

Require: Prompt distribution \mathcal{D} , old policy $\pi_{\theta_{\text{old}}}$, reflection limit n_{max} , group size G , reward parameters $(\alpha, \beta, \gamma, \delta, n_0, \lambda, \eta, \tau_q, \epsilon, \varphi, \psi, \xi)$

- 1: Initialize policy parameters $\theta \leftarrow \theta_{\text{old}}$
- 2: **while** not converged **do**
- 3: Sample prompt $q \sim \mathcal{D}$
- 4: Rollout G structured trajectories $\{\tau_i\}_{i=1}^G$ from $\pi_{\theta_{\text{old}}}(\cdot | q)$
- 5: **for** $i = 1$ **to** G **do**
- 6: Parse segments
 $(o^{(\text{think})}, o^{(\text{answer})}, \{(o^{(\text{reflection},j)}, o^{(\text{answer},j+1)})\}_{j=1}^{n_i})$
- 7: Compute format gate $F(\tau_i) = \mathbb{I}[\tau_i \in \mathcal{T}_{\text{valid}}]$ \triangleright Eq. (3)
- 8: **if** $F(\tau_i) = 0$ **then**
- 9: Set $R_{\text{overall}}(\tau_i) \leftarrow 0$
- 10: **else**
- 11: Evaluate quality scores $\mathbf{r}_i = (r_{i,0}, r_{i,1}, \dots, r_{i,n_i}) \triangleright r_{i,0}$ for $o^{(\text{answer})}$, $r_{i,t}$ for $o^{(\text{answer},t+1)}$
- 12: Compute cycle penalty $P(n_i)$ \triangleright Eq. (4)
- 13: Compute trajectory quality reward $R_{\text{trajectory}}(\tau_i)$ \triangleright Eq. (7)
- 14: Compute efficiency term $E(n_i)$ \triangleright Eq. (8)
- 15: Compute overall reward:
 $R_{\text{overall}}(\tau_i) \leftarrow P(n_i)(\varphi R_{\text{trajectory}}(\tau_i) + \psi E(n_i)) + \xi$ \triangleright Eq. (9) since $F(\tau_i) = 1$
- 17: **end if**
- 18: **end for**
- 19: Compute $\mu_R \leftarrow \frac{1}{G} \sum_{j=1}^G R_{\text{overall}}(\tau_j)$
- 20: Compute $\sigma_R \leftarrow (\frac{1}{G} \sum_{j=1}^G (R_{\text{overall}}(\tau_j) - \mu_R)^2)^{\frac{1}{2}}$
- 21: **for** $i = 1$ **to** G **do**
- 22: $\hat{A}_i \leftarrow \frac{R_{\text{overall}}(\tau_i) - \mu_R}{\sigma_R + \epsilon}$ \triangleright Group-relative advantage
- 23: **end for**
- 24: Update π_{θ} with Reflection-aware GRPO using $\{\tau_i, \hat{A}_i\}_{i=1}^G$ (see Sec. D)
- 25: $\theta \leftarrow \theta + \eta_{\text{optimization}} \nabla_{\theta} \mathcal{J}_{\text{GRPO}}(\theta)$
- 26: **end while**

813

814

815

816

C Reward Design Principles

This trajectory-level quality reward is motivated by the intrinsic requirements of iterative refinement tasks in reinforcement learning. The reward structure serves three interrelated objectives as follows:

- First, it promotes continuous improvement of answers by providing positive reinforcement for any quality gain and proportional scaling with the magnitude of improvement.
- Second, it penalizes decline in quality and also discourages stagnation when the performance is below the maximum achievable level, thereby maintaining the incentive to search for better solutions.
- Third, it preserves stability once the quality has reached its maximum by avoiding penalties for lack of improvement at that point.

This combination of principles balances the drive for progress with the preservation of optimal states, preventing policies from sacrificing existing high-quality answers in pursuit of transient improvement signals or prematurely halting refinement before reaching optimal performance.

D Reflection-aware GRPO

For a given prompt q under the old policy $\pi_{\theta_{\text{old}}}$, we sample a group of G outputs $\{o_i\}_{i=1}^G$, each evaluated with the proposed reward $R_{\text{overall}}(\tau_i)$. The group-relative normalized advantage for the i -th trajectory is

$$\hat{A}_i = \frac{R_{\text{overall}}(\tau_i) - \mu_R}{\sigma_R} \quad (10)$$

where $\mu_R = \frac{1}{G} \sum_{j=1}^G R_{\text{overall}}(\tau_j)$, $\sigma_R = \left(\frac{1}{G} \sum_{j=1}^G (R_{\text{overall}}(\tau_j) - \mu_R)^2 \right)^{-\frac{1}{2}}$. Policy optimization then follows the clipped surrogate objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q \sim \mathcal{D}, \{o_i\} \sim \pi_{\theta_{\text{old}}}} \left[\frac{1}{G} \sum_{i=1}^G \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \left(\min(r_{i,t}(\theta) \hat{A}_i, \text{clip}(r_{i,t}(\theta), 1 - \varepsilon, 1 + \varepsilon) \hat{A}_i) - \beta_{\text{KL}} D_{\text{KL}}(\pi_{\theta} \parallel \pi_{\text{ref}})) \right) \right], \quad (11)$$

$$r_{i,t}(\theta) = \frac{\pi_{\theta}(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}.$$

where $r_{i,t}(\theta)$ denotes per-token likelihood ratio. This formulation preserves GRPO’s stability advantages while embedding ReflexiCoder’s reflection-aware reward into advantage computation, aligning gradient updates with both code correctness and self-reflection efficiency.

E Implementation Details

In this section, we further illustrate the implementation details as follows.

E.1 Baselines

We compare against five representative open-source code models: Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), Seed-Coder-8B-Instruct (Seed et al., 2025), DeepSeek-Coder-7B-Instruct (Guo et al., 2024), CodeGemma-7B-IT (Team et al., 2024), and CodeLlama-7B-Instruct (Roziere et al., 2023). For each baseline, we use the official chat template and generation settings recommended by the authors whenever available.

E.2 Dataset Curation

Our training dataset is derived from the open-source DeepCoder training corpus (Luo et al., 2025). We directly use four subsets released by DeepCoder: TACO-Verified (7,436 problems), LiveCodeBench (599), CodeForces (6,128), and LeetCode (2,641). We follow DeepCoder’s released preprocessing, including quality filtering and decontamination, to ensure reliable problem statements and to reduce overlap with standard evaluation benchmarks. The LiveCodeBench portion of our training data only contains problems submitted between May 1, 2023 and July 31, 2024. We additionally verify that it does not overlap with the LiveCodeBench v5 test split used by the EvalChemistry² evaluation framework, whose problems fall in the time window August 1, 2024 to February 1, 2025.

E.3 Hyperparameters Settings

Our ReflexiCoder optimizes a multi-step self-reflection and self-correction trajectory with a composite reward. For clarity and reproducibility, we list all reward-related coefficients and thresholds in Table 3. Unless otherwise specified, we keep these hyperparameters fixed across all tasks and benchmarks. In the sensitivity analysis in Section 3.3, we observe that final performance is stable over a

²<https://github.com/mlfoundations/evalchemistry>

Table 3: Reward-related hyperparameters. Unless otherwise stated, these values are kept fixed across all tasks and benchmarks for reproducibility.

Source	Symbol	Value	Meaning
Eq. 4	α	0.1	polynomial decay strength
	β	2.0	polynomial decay curvature
	γ	0.05	exponential decay strength
	δ	0.1	oscillation magnitude
	n_0	5	reflection depth with no penalty
Eq. 5	λ	0.2	weight concentration on later iterations
Eq. 6	s	0.1	sensitivity to quality changes
	ε	1e-4	tolerance for numerical stability
	h_{pos}	0.05	stagnation penalty
	h_{neg}	1.0	stagnation penalty
	r_{max}	1.0	maximum achievable pass rate
Eq. 7	η	0.5	weight of improvement term
Eq. 8	τ_q	1.0	required quality threshold
	ϵ	1e-6	prevents division singularity
Eq. 9	φ	1.0	weight of trajectory quality term
	ψ	0.5	weight of efficiency term
	ξ	1.0	weight of formatting constraints term

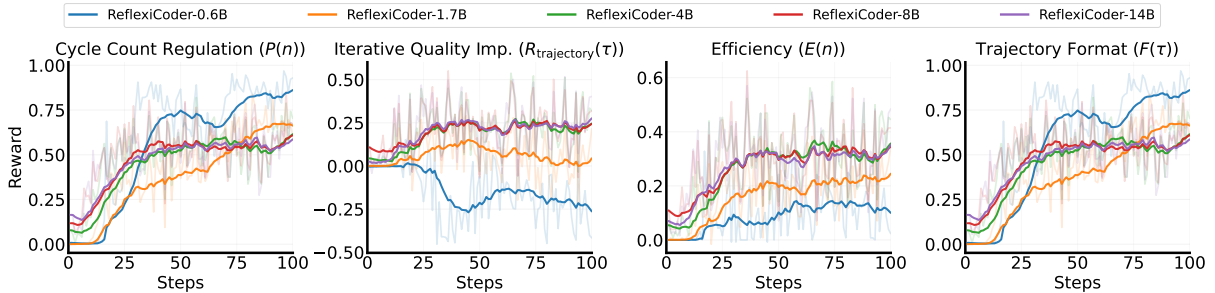


Figure 7: Training dynamics of key reward components and behaviors for different model sizes. Larger models learn to achieve higher progressive improvement and efficiency rewards more quickly. They also converge to a more optimal number of reflection cycles, whereas smaller models struggle to fully optimize the complex reward landscape.

broad range of values, suggesting that the proposed RL objective does not rely on brittle tuning.

F Training Dynamics and Reward Scaling

In this section, we provide a detailed analysis of the reward-learning dynamics across different model scales, as referenced in the main text. To investigate how model capacity influences the optimization of the reflection policy, we track the progression of the key reward components during the RL fine-tuning process.

As illustrated in Figure 7, we observe that different reward components exhibit distinct scaling behaviors, rather than a uniform “larger-is-always-

better” trend:

- **Cycle Count Regulation ($P(n)$):** Larger backbones improve cycle-count control more substantially. In particular, the 0.6B model continues to climb and reaches the highest $P(n)$ by the end of training, while 4B/8B/14B rise quickly but plateau earlier at a lower level. This suggests that smaller models may rely more on increasing/refining the number of cycles to gain reward, whereas larger models learn an adequate stopping behavior earlier.
- **Progressive Improvement ($R_{\text{trajectory}}(\tau)$):** Scaling is crucial for learning genuine iterative quality gains. The 0.6B model’s

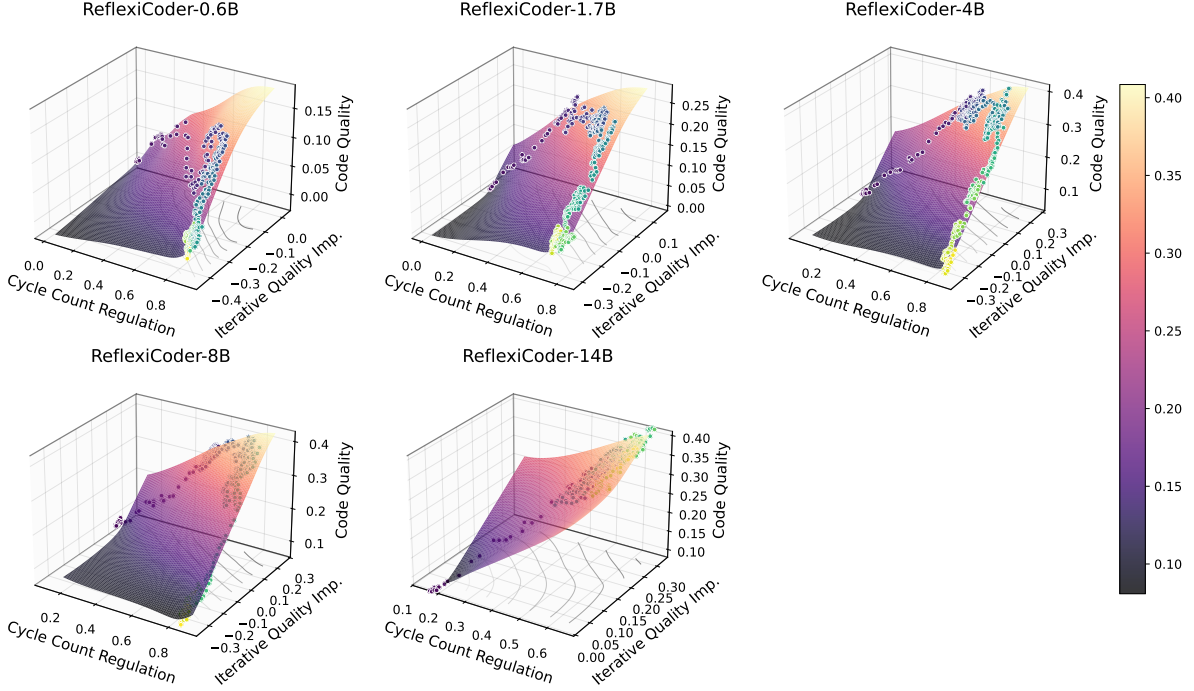


Figure 8: Reward surface induced by shaping terms. For each model scale (0.6B-14B), we fit a smooth 2D surface $\hat{z} = f(x, y)$ via RBF regression, where x is the cycle-count regulation reward $P(n)$, y is the iterative quality reward $R_{\text{trajectory}}$, and z is final code-quality reward (reward on the last rewritten code). The surface height and colormap jointly encode the predicted \hat{z} (higher/brighter indicates better code quality), while overlaid points denote the observed training samples (x_t, y_t, z_t) colored by training step; grey contour projections highlight local gradients. Consistent high- \hat{z} regions across scales indicate regimes where shaping terms synergistically improve code quality, whereas steep slopes/contour crowding reveal sensitivity to the corresponding reward component.

$R_{\text{trajectory}}(\tau)$ degrades after early training and becomes persistently negative, indicating unstable or even harmful reflection updates. In contrast, models $\geq 4\text{B}$ steadily reach and maintain positive $R_{\text{trajectory}}(\tau)$ (with 8B/14B slightly higher and more stable), suggesting that sufficient capacity is needed to internalize the “reflect-and-correct” mechanism without drifting.

- Efficiency Reward ($E(n)$): Larger models achieve higher and more stable efficiency rewards. The 0.6B model remains low throughout training, 1.7B improves but saturates at a moderate value, while 4B/8B/14B converge to a clearly higher plateau. This indicates that scaling helps produce concise yet effective deliberation, reducing redundant or oscillatory reflection traces.
- Trajectory Format ($F(\tau)$): Formatting compliance improves with scale, but the most pronounced gain appears in the smallest model: 0.6B eventually attains the highest $F(\tau)$, while larger models improve quickly and then

saturate. This suggests that format adherence is comparatively easy to learn across scales, and may not be the main bottleneck once a model reaches moderate capacity.

Overall, these dynamics reinforce that the gains of our ReflexiCoder are driven by learning structured self-correction policies—especially the ability to generate positive iterative improvements ($R_{\text{trajectory}}(\tau)$) and efficient deliberation ($E(n)$) which emerge reliably only at sufficient model scale, rather than being a mere artifact of longer rollouts or increased sampling.

G Interpretation of Reward Shaping

To understand how our shaping terms drive the performance gains of ReflexiCoder, we visualize the learned reward landscape over training. For each model scale, we fit a smooth surface $\hat{z} = f(x, y)$ with RBF regression, where $x = P(n)$ is the cycle-count regulation reward, $y = R_{\text{trajectory}}$ is the iterative quality improvement reward, and z is the final code-quality reward (reward on the last rewritten code). Figure 8 reveals three consistent patterns

971 across scales. The surface height and colormap
972 jointly encode the predicted \hat{z} (higher/brighter in-
973 dicates better code quality), while overlaid points
974 denote the observed training samples (x_t, y_t, z_t)
975 colored by training step; grey contour projections
976 highlight local gradients. Consistent high- \hat{z} regions
977 across scales indicate regimes where shaping terms
978 synergistically improve code quality, whereas steep
979 slopes/contour crowding reveal sensitivity to the
980 corresponding reward component.

981 Across 0.6B to 14B, the highest \hat{z} concentrates in
982 regions where both $P(n)$ and $R_{\text{trajectory}}$ are strong.
983 This indicates that quality gains are not explained
984 by “more iterations” or “better rewriting” alone.
985 Instead, the model benefits most when it learns a
986 structured self-debugging trajectory: allocating an
987 appropriate number of reflection cycles while mak-
988 ing each revision measurably improve the solution.
989 This supports our framing of self-correction as a
990 multi-step decision process optimized end-to-end
991 by RL.

992 Moreover, the surfaces show that pushing
993 $R_{\text{trajectory}}$ without sufficient $P(n)$ does not reliably
994 yield high z . In practice, unconstrained reflection
995 can lead to over-editing, oscillations, or verbose
996 but non-functional changes. The positive gradient
997 along $P(n)$ suggests that cycle-count regulation
998 acts as an implicit budgeting and credit assignment
999 aid, steering the policy toward reflection depths that
1000 are most likely to convert into correct final code
1001 rather than extended but low-yield “thinking”.

1002 Furthermore, as scale increases, the high- \hat{z} re-
1003 gion becomes broader and the attainable \hat{z} in-
1004 creases, indicating that larger models can convert
1005 trajectory-level improvement signals into final cor-
1006 rectness more consistently. Importantly, the same
1007 qualitative landscape persists across scales, suggest-
1008 ing the reward design is not brittle or size-specific.
1009 This helps explain why our ReflexiCoder’s RL
1010 training yields robust gains on complex bench-
1011 marks: the model is not merely optimizing for
1012 a better first-pass solution, but learning an inter-
1013 nalizable debugging strategy that generalizes with
1014 capacity.

1015 Overall, the visualization provides mechanistic
1016 evidence that the performance jump comes from
1017 our RL-optimized intrinsic self-reflection and self-
1018 correction loop. The shaping terms jointly encour-
1019 age (i) when to stop reflecting and (ii) how to
1020 make corrections that monotonically improve code,
1021 reducing reliance on external execution feedback
1022 while improving final functional correctness.

H The Use of Ai Assistants 1023

Ai Assistants (*e.g.*, LLMs) are employed solely for
polishing writing. 1024
1025