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

Juyong Jiang[1 2*] **Jiasi Shen**[2†] **Sunghun Kim**[1] **Kang Min Yoo**[3]
Jeonghoon Kim[4†] **Sungju Kim**[4†]
[1]The Hong Kong University of Science and Technology (Guangzhou)
[2]The Hong Kong University of Science and Technology
[3]Amazon AGI [4]NAVER Cloud
`csjuyongjiang@gmail.com`, `{sjs,hunkim}@cse.ust.hk`
`kangminy@amazon.com`, `{jeonghoon.samuel,sungju.kim}@navercorp.com`

Abstract

While Large Language Models (LLMs) have revolutionized code generation, standard "System 1" approaches that generate solutions in a single forward pass often hit a performance ceiling on complex algorithmic tasks. Existing iterative refinement strategies attempt to bridge this gap at inference time, yet they predominantly rely on external oracles, execution feedback, or computationally expensive prompt-response cycles. In this work, we propose ReflexiCoder, a novel reinforcement learning (RL) framework that internalizes the structured reasoning trajectory, encompassing initial generation, bug and optimization aware reflection, and self-correction, directly into the model's weights. Unlike prior methods, ReflexiCoder shifts the paradigm from *external-dependent refinement* to *an intrinsic, fully autonomous self-reflection and self-correction capabilities* at inference time. We utilize an RL-only training paradigm with granular reward functions to optimize the entire reflection-correction trajectory, teaching the model *how to debug* without reliance on ground-truth feedback or execution engines at inference time. Extensive experiments across seven benchmarks demonstrate that our ReflexiCoder-8B establishes a new state-of-the-art (SOTA) among leading open-source models in the 1.5B-14B range, achieving **94.51% (87.20%)** on HumanEval (Plus), **81.80% (78.57%)** on MBPP (Plus), **35.00%** on BigCodeBench, **52.21%** on LiveCodeBench, and **37.34%** on CodeForces in a **single-attempt** setting, rivaling or surpassing proprietary models like GPT-5.1. Notably, our framework is significantly more token-efficient than base models, reducing inference-time compute overhead by approximately **40%** through disciplined, efficient reasoning and reflection patterns. The source code and data are available at `https://github.com/juyongjiang/ReflexiCoder`.

1 Introduction

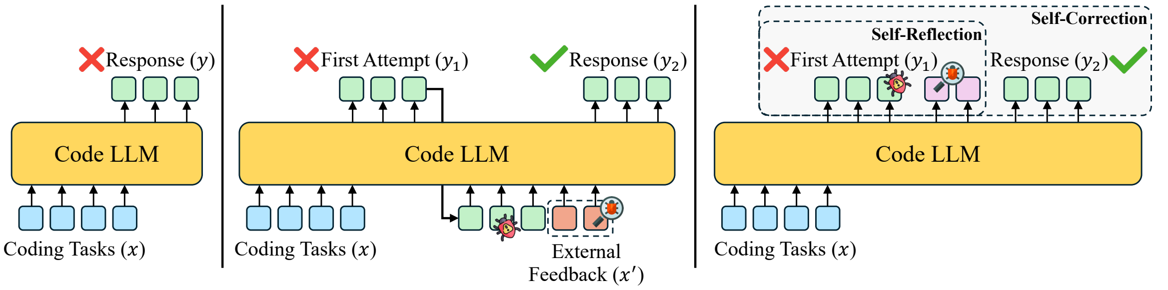
Large Language Models (LLMs) have revolutionized software engineering, demonstrating exceptional proficiency in translating natural language specifications into executable code (Chen et al., 2021; Dakhel et al., 2023; Jiang et al., 2026; Li et al., 2025a). Despite these advancements, standard "System 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; Wang et al., 2024; Park et al., 2025; 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 at inference time. 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**, including prompt-based self-reflection such as Reflexion (Shinn et al., 2024), which relies on signals from execution environments or prompting frozen models to iteratively improve code (Chen et al., 2024; Jiang et al., 2023b; Zhong et al., 2024; Madaan et al., 2024). While effective, these methods suffer from a critical bottleneck: *dependency on external oracles, environmental interaction, and excessive inference-time token consumption.* In real-world development, comprehensive unit tests are often absent, and the iterative overhead of multiple prompt-response cycles leads to significant latency and computational costs. Furthermore, re-

---

[*] Work done during a research internship at NAVER.
[†] Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 37543–37562
July 2-7, 2026



(a) Single Attempt **(b) Multi-Attempts with External Feedback** **(c) Multi-Attempts with Intrinsic Self-Reflection**

Figure 1: A comparative overview of iterative code refinement workflows at inference time. (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, reflection 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 oracles and environmental interaction.

lying on external signals prevents models from internalizing intrinsic debugging capabilities, the ability to scrutinize and correct one's own logic autonomously.

Inspired by the success of reasoning-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 internalize the structured reasoning trajectory, encompassing initial reasoning, code generation, reflection for bugs and optimization, and correction, directly into the model's weights. Unlike prior works (Shinn et al., 2024; Madaan et al., 2024), **ReflexiCoder shifts the paradigm from external-dependent refinement to intrinsic, fully autonomous self-reflection and self-correction capabilities at inference time.** By optimizing the entire correction trajectory itself rather than just the generation policy, **we teach the model the cognitive skill of "how to debug" without reliance on ground-truth feedback or external execution engines at inference time**. Figure 1 compares this paradigm shift with prior practices.

To achieve this, our ReflexiCoder utilizes an RL-only (R1-Zero) training paradigm, bypassing traditional supervised fine-tuning (SFT) to autonomously discover efficient reflection-correction patterns tailored to its own parameter space and problem-solving capabilities (Guo et al., 2025; Li et al., 2025b). We optimize the problem-solving trajectory via granular reward functions that incentivizes both accurate error detection and successful repair. It is worth noting that our approach marks a fundamental departure from prior RL methods for code generation, such as CodeRL (Le et al., 2022), PPOCoder (Shojaee et al., 2023), DeepCoder (Luo et al., 2025). While existing RL methods strictly optimize the *single-pass generation policy* using execution rewards, **they fail to cultivate the intrinsic reasoning capability to identify and analyze potential errors and iteratively correct them autonomously after an initial attempt.** Our ReflexiCoder uniquely applies RL to optimize the *reflection-correction trajectory itself*, transforming self-debugging from an environment-dependent test loop into an *intrinsic cognitive skill*.

Extensive experiments across seven benchmarks demonstrate the efficacy of our approach. ReflexiCoder-8B establishes a new state of the art 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) in a **single-attempt** setting. When utilizing our iterative reasoning-reflection setup, referred to as ReflexiCoder-8B (Multiple), performance further scales to **95.73%**, **36.84%**, **54.12%**, and **37.68%**, respectively, surpassing or remaining competitive with proprietary models like GPT-5.1 (OpenAI, 2025).

Notably, we show that these gains do not come from excessive inference-time compute overhead, as our empirical observations (Section 3.3) reveal that ReflexiCoder is **significantly more token-efficient** than base model, consuming approximately **40% fewer tokens** in iterative mode. This is driven by a nearly **50% reduction** in reasoning tokens, as our RL training teaches the model to efficiently isolate fundamental logic rather than rambling. Furthermore, ReflexiCoder demonstrates a highly *disciplined reflection* pattern, executing exactly one reflection cycle in virtually all cases. This ensures that our model achieves superior accuracy at the same computational budget in the single-attempt setting, or at an even lower budget in the multi-attempt setting, effectively transforming self-reflection and self-correction into an efficient, low-latency cognitive process. In summary, our main contributions are as follows:

- We propose ReflexiCoder, an RL-based framework that transforms self-reflection and self-correction from an environment-dependent test loop into a fully autonomous, intrinsic model capability, eliminating the need for external feedback at inference time.
- We formulate the reflection-correction loop as a multi-step trajectory and optimize it via RL. Unlike existing RL methods for code generation, our approach targets the *reflection-correction trajectory*, teaching the model the fundamental logic of self-debugging.
- Our ReflexiCoder-8B significantly outperforms leading open-source models and competes with proprietary models like GPT-5.1. We demonstrate that these gains hold even under fair or reduced token-budget comparisons.
- We release our source code and data to facilitate future research into LLMs' internal self-improvement capabilities.

2 Methodology

In this section, we formalize the proposed ReflexiCoder training pipeline, shown in Figure 2, which integrates structured self-reflection and self-correction into LLMs and optimizes the resulting trajectories via RL.

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

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

where $n \in \mathbb{N}$ denotes the number of reflection iterations. 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

$$\tau \equiv \Big(q, o^{(\text{think})}, o^{(\text{answer})}, \big\{\big(o^{(\text{reflection},j)}, o^{(\text{answer},j+1)}\big)\big\}_{j=1}^{n}\Big) \tag{2}$$

and define the set of all format-compliant trajectories as $\mathcal{T}_{\text{valid}} = \{\tau \in \mathcal{T} \mid \Phi(\tau) = \mathcal{F}^{\star}\}$ with $\Phi(\cdot)$ denotes syntax extractor and $\mathcal{F}^{\star}$ 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 predetermined structural format. Each generated response must consist of a distinct internal reasoning segment, an initial 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 $F(\tau)$:

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

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.

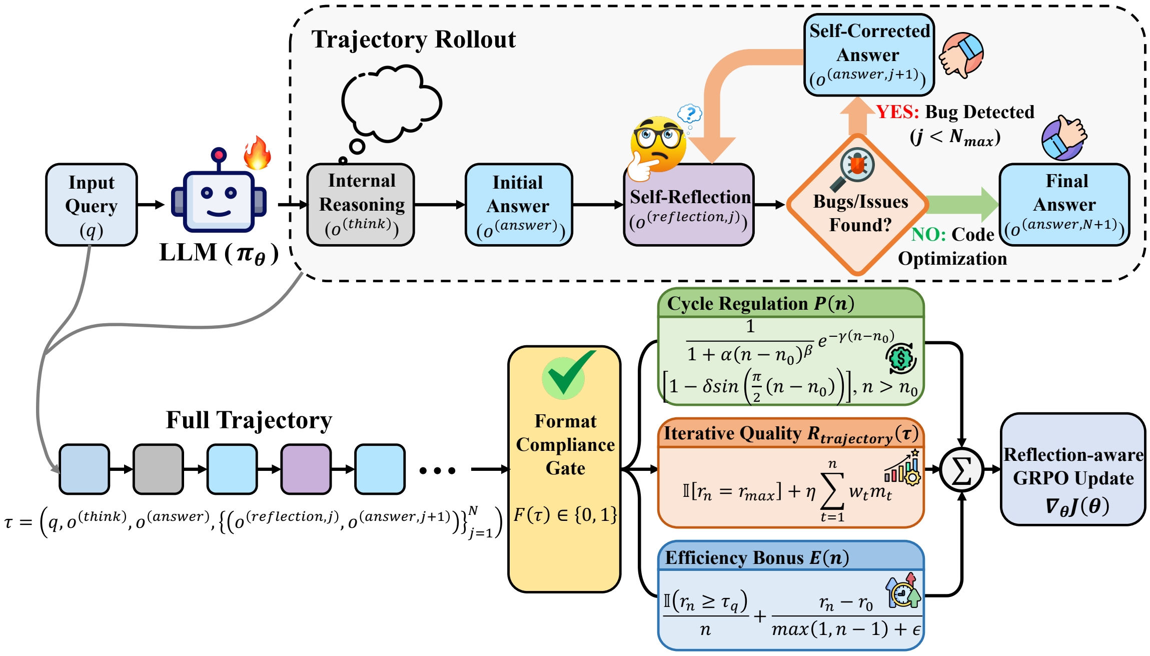


Figure 2: The architecture of ReflexiCoder formulates code generation as an RL-optimized intrinsic self-debugging trajectory. A carefully designed composite reward jointly incentivizes reflection quality and correction success.

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 $1 \le n \le n_0$, 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:

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

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 that encourages exploration over nearby iteration depths. In multi-turn RL, policies often exhibit trajectory collapse (Wang et al., 2025; Park et al., 2026), for example, repeatedly producing the same incorrect code, or oscillating between two erroneous states. A purely monotonic penalty can exacerbate this issue by encouraging premature termination once the model enters a bad local cycle. By injecting a bounded sinusoidal term, we make the per-step penalty slightly non-stationary across turns, which periodically "nudges" the policy away from repetitive local optima and promotes exploration of alternative correction paths.

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 $\mathbf{r} = (r_0, r_1, \ldots, r_n) \in \mathbb{R}^n$, where r_t represents the quality score of the t-th solution obtained through automated execution and validation. Ideally, the optimal trajectory should satisfy $r_0 \le r_1 \le \cdots \le r_n$, reflecting a progressive improvement in code quality. To emphasize the importance of later improvement stages within a trajectory, we apply exponential time-weighting

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

which yields a normalized vector $\mathbf{w} = (w_1, w_2, \ldots, w_n) \in \Delta^n$ over the probability simplex in $\mathbb{R}^n$, with the parameter λ controlling the degree to which later iterations are prioritized. The resulting weights satisfy $w_1 < w_2 < \cdots < w_n$, 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:

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

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, with h_{pos} providing a bonus when the score has effectively converged near $r_{\max}$ and h_{neg} imposing 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

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

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 B.

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 (Parthasarathi et al., 2025). To counter these problems, we introduce an efficiency term:

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

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:

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

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 how to reflect effectively across iterations and when to stop**, achieving a disciplined reflection mechanism aligned with the overarching objectives of human. The token budget is discussed in Appendix H.

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 C.

3 Experiments

3.1 Experimental Settings

Models and Benchmarks We instantiate our model by fine-tuning a recent open-source base model, Qwen3-8B (Yang et al., 2025), to obtain our ReflexiCoder-8B. We evaluate performance across a diverse set of seven widely-used 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).

Implementation Details We implement our RL pipeline using the TRL (von Werra et al., 2020). 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 D.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 8× NVIDIA H200 GPUs with a per-device batch size of 1. More details are provided in Appendix D, including the baselines (Section D.1), dataset curation (Section D.2), hyperparameter settings (Section D.3), and the system prompt (Section D.4).

Evaluation To ensure a rigorous and equitable comparison, we evaluate all baseline models and our ReflexiCoder utilizing the EvalChemy (Raoof et al., 2025). To decouple the intrinsic quality of the learned policy from gains attributable to increased inference compute, we report results under two configurations with different token budgets. First, ReflexiCoder-8B (Single) denotes that the model is evaluated in a single-attempt setting, in which the system prompt (see Appendix Figure 10) is removed. *This configuration keeps the token budget strictly identical to that of the baseline models, thereby serving as a direct measure of the model's fundamental problem-solving capability.* Second, we report ReflexiCoder-8B (Multiple), which utilizes the full iterative reasoning-reflection paradigm enabled by the system prompt. Crucially, as discussed in Appendix H, our RL process incentivizes the model to internalize an *optimal trajectory*, encouraging it to generate high-quality, bug-free solutions on the first attempt and require only concise subsequent optimization.

3.2 Main Results

Table 1 presents a comprehensive evaluation of ReflexiCoder-8B across seven diverse code generation benchmarks. Overall, our model establishes a new state-of-the-art for open-source models in the 1.5B to 14B parameter range. Compared to its base model, Qwen3-8B, ReflexiCoder-8B (Single) achieves significant absolute improvements across all metrics, most notably increasing pass@1 accuracy by 14.46% on LiveCodeBench (LCB) and 13.64% on CodeForces (CF). These gains on reasoning-intensive tasks underscore the efficacy of our RL paradigm in cultivating deep algorithmic reasoning rather than mere syntax memorization.

Furthermore, ReflexiCoder-8B consistently outperforms both specialized code LLMs, such as Qwen2.5-Coder-7B and Seed-Coder-8B, and recent RL-based models, including LeDex-RL-13B and DeepCoder-14B-Preview. In a strict single-attempt setting, ReflexiCoder-8B (Single) surpasses DeepCoder-14B-Preview by 18.16% on LCB and 23.10% on CF, despite having 40% fewer parameters. This directly validates our core hypothesis that optimizing for the reflection-correction trajectory via RL yields a significantly more robust policy than standard single-pass RL methods. Despite its compact 8B scale, ReflexiCoder-8B demonstrates highly competitive performance against much larger proprietary models. Under the iterative setup, ReflexiCoder-8B (Multiple) achieves parity with GPT-5.1 on HumanEval+ (87.80% vs. 87.20%) and MBPP+ (79.10% vs. 79.10%). Crucially, on the most challenging benchmarks, it outperforms GPT-5.1 by clear margins, scoring 54.12% (vs. 48.03%) on LCB and 37.68% (vs. 34.70%) on CF. This highlights the effectiveness of our granular reward design in empowering smaller models to handle high-complexity tasks.

Moreover, comparing the Single and Multiple inference settings reveals the intrinsic value of our internalized self-reflection mechanism. By simply activating the system prompt to trigger internal iterations without any external execution feedback, performance scales positively across all benchmarks, such as rising to 95.73% on HE and 36.84% on BigCodeBench. This confirms that our ReflexiCoder does not merely memorize code patterns but has successfully internalized a generalizable, inference-time self-improvement strategy.

For the remainder of this paper, we report the performance of ReflexiCoder-8B (Single) and refer to it simply as ReflexiCoder-8B unless otherwise specified. This choice is strategically motivated: while our *Multiple* variant yields superior results through iterative refinement, we aim to demonstrate that the performance gains of our ReflexiCoder stem from the *intrinsic quality of the foundational policy optimized via RL, rather than simply consuming a higher inference-time token budget*. By focusing on the *single-attempt* setting, we eliminate the confounding factor of iterative overhead and prove that our RL paradigm fundamentally enhances the model's reasoning capabilities under a constrained,

Table 1: Main results on seven code generation benchmarks, reporting pass@1 (%). We compare our ReflexiCoder against leading proprietary and open-source models. Our models establish a new state-of-the-art for open-source models in 1.5B-14B range and demonstrate competitive performance against much larger proprietary models. * denotes results taken from (Jiang et al., 2024). HE(+) denotes HumanEval(+). BCB is BigCodeBench, LCB is LiveCodeBench, and CF is CodeForces. ReflexiCoder-8B (Single) is evaluated in *single-attempt* without the system prompt (see Figure 10), while ReflexiCoder-8B (Multiple) uses the full *iterative reasoning-reflection* setup. We show the score improvement (±) of our model relative to its base model (Qwen3-8B). **Bold** indicates the best performance among open-source and proprietary models, respectively.

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	38.21*	37.18*	16.58	1.19	0.68
Purdue	LeDex-RL-13B*	61.71	56.68	61.98	57.89	-	-	-
	LeDex-RL-7B*	55.84	50.04	57.92	52.90	-	-	-
rLLM	DeepCoder-14B-Preview	80.49	75.00	63.40	67.46	28.33	34.05	14.24
	DeepCoder-1.5B-Preview	67.07	61.59	36.00	52.12	6.84	22.94	9.89
Ours	**ReflexiCoder-8B (Single)**	**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**$_{+13.64}$
	ReflexiCoder-8B (Multiple)	**95.73**$_{+6.71}$	**87.80**$_{+7.31}$	**82.00**$_{+11.8}$	**79.10**$_{+8.73}$	**36.84**$_{+4.21}$	**54.12**$_{+16.37}$	**37.68**$_{+13.98}$

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 trajectories.

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

equivalent computational budget.

3.3 In-depth Analysis and Insights

In this section, we conduct a series of analyses to understand the emergent behaviors and scalability of ReflexiCoder, with a case study in Appendix I.

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-answer-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 trajectories.

Impact of Reasoning & Reflection To investigate the source of ReflexiCoder's performance gains, we conduct a comparative analysis against

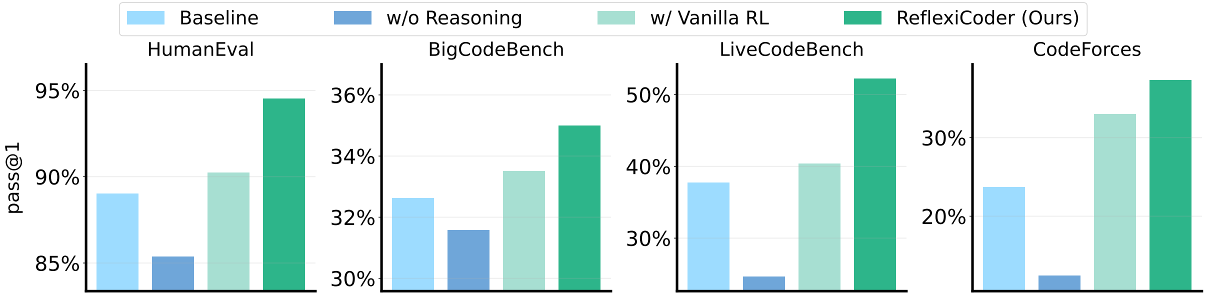


Figure 3: Analysis of the impact of reasoning and reflection. We compare ReflexiCoder against baselines that lack a structured reasoning and reflection step. Performance is pass@1 (%). The significant performance gap highlights that the structured reasoning-reflection cycle is the key driver of improvement.

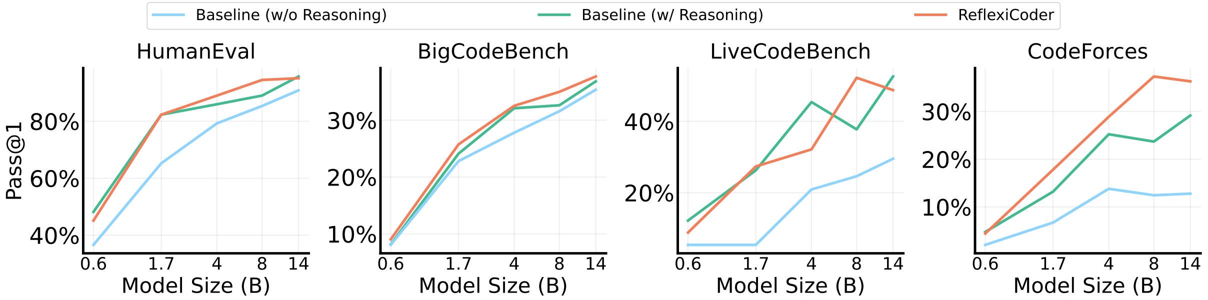


Figure 4: Scaling analysis of ReflexiCoder with model size. Performance is pass@1 (%) across four representative benchmarks. The performance grow with model scale, indicating a super-linear benefit.

three critical baselines: (1) Baseline, the base model without any additional fine-tuning; (2) No Reasoning Pattern, where the base 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-correction trajectory. The results are 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 models from 0.6B to 14B. Figure 4 reports average pass@1 over HumanEval, BigCodeBench, LiveCodeBench, and CodeForces for the base model, a *non-reasoning* variant, and our RL-trained ReflexiCoder. Across all sizes, *non-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.

Token Budget Analysis To thoroughly investigate the computational overhead of our proposed framework, we analyze the token consumption across different models and settings. As detailed in Table 3, both configurations of our model, ReflexiCoder (Single) *without* the system prompt, and ReflexiCoder (Multiple) *with* the system prompt consistently exhibit lower average full token budgets compared to the reasoning baselines, Qwen3-8B and Qwen3-14B. Notably, the "Reflection" distribution reveals that ReflexiCoder (Multiple) performs exactly one self-reflection step across nearly all tasks (164/164 on HumanEval and 1,139/1,140 on BigCodeBench). This empirically validates our previous assertion: the model successfully learns the "optimal trajectory", synthesizing a correct solution on the first attempt and executing only a single,

Table 3: Token budget statistics and self-reflection count distributions across benchmarks of varying difficulty levels. The "Reflection" column details the frequency distribution of reflection cycles executed per task. The "Full" designation refers to the token count of the entire generated response, whereas "Reasoning" isolates the specific token footprint of the reasoning process.

Model	**HumanEval (164 tasks)**				**BigCodeBench (1,140 tasks)**			
	Min	**Avg**	**Max**	**Reflection**	**Min**	**Avg**	**Max**	**Reflection**
Qwen2.5-Coder-7B-Instruct	61	289.18	820	-	116	430.02	1104	-
Qwen2.5-Coder-14B-Instruct	52	376.15	749	-	89	329.63	770	-
Qwen3-8B (Full)	690	4,170.03	16,385	-	856	5,214.01	16,385	-
Qwen3-14B (Full)	402	3,558.22	15,956	-	608	4,152.02	14,572	-
ReflexiCoder (Single) (Full)	553	3,455.13	16,381	{0: 164}	860	3,477.28	16,384	{0: 1140}
ReflexiCoder (Multiple) (Full)	551	2,214.51	11,107	{1: 164}	695	2,422.33	16,385	{0: 1, 1: 1139}
Qwen3-8B (Reasoning)	0	3,133.70	15,300	-	0	4,592.46	15,357	-
Qwen3-14B (Reasoning)	0	2,744.73	13,476	-	516	3,754.36	13,556	-
ReflexiCoder (Single) (Reasoning)	0	2,701.11	13,827	-	0	2,866.30	8,782	-
ReflexiCoder (Multiple) (Reasoning)	253	1,743.43	10,529	-	0	1,841.67	6,919	-

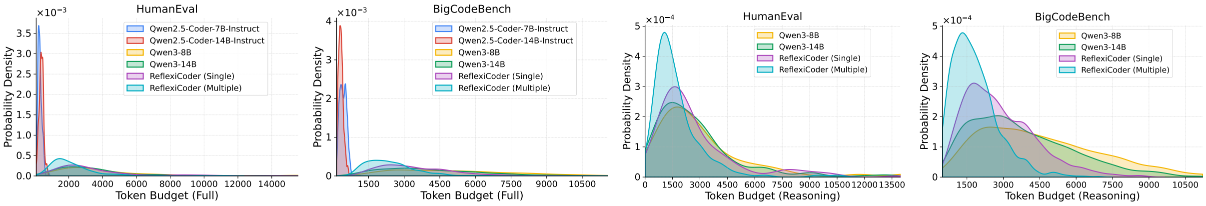


Figure 5: Probability density distributions of token consumption for both full responses (top) and reasoning segments (bottom) across the HumanEval and BigCodeBench benchmarks. The reasoning segments correspond to the `<think>...</think>` generation phases. Note that the Qwen2.5-Coder-7B/14B-Instruct baselines are not reasoning models, and thus lack corresponding reasoning token budgets.

concise optimization pass. Despite engaging in an additional reflection and optimization phase, ReflexiCoder (Multiple) consumes significantly *fewer* total tokens than ReflexiCoder (Single). To uncover the mechanism driving this efficiency, we isolate and analyze the token budget specifically allocated to the reasoning process (denoted as "Reasoning" in Table 3). A clear hierarchical reduction in reasoning tokens emerges: Qwen3-8B > Qwen3-14B > ReflexiCoder (Single) > ReflexiCoder (Multiple). This demonstrates that our RL training paradigm profoundly enhances the model's core deductive capabilities. By internalizing a more structured and goal-directed reasoning process, ReflexiCoder efficiently isolates the critical logic required to solve the problem, thereby eliminating redundant exploration and verbalization.

These statistical findings are further corroborated by the visual distribution of token budgets shown in Figure 5. The probability density curves for both the full response and the reasoning segments illustrate a distinct leftward shift for ReflexiCoder compared to the Qwen3 baselines, indicating a strong concentration toward lower token budgets. Specifically, the distribution for ReflexiCoder (Multiple) exhibits the sharpest peak at the lowest token range across both HumanEval and BigCodeBench.

Therefore, the dynamic self-reflection and self-correction mechanisms **do not** impose a computational burden. On the contrary, the highly effective RL optimization suppresses inefficient generation and improves reasoning capabilities, allowing the model to achieve superior performance with a substantially more economical token consumption.

4 Conclusion

In this work, we propose ReflexiCoder, a framework that leverages reinforcement learning to teach LLMs intrinsic self-reflection and self-correction without relying on external oracles and environmental interaction at inference time. By formulating debugging as a trainable decision-making trajectory, our ReflexiCoder-8B achieves state-of-the-art performance among open-source models in the 1.5B to 14B range and surpasses or remains competitive with proprietary models like GPT-5.1 on complex reasoning benchmarks. These results demonstrate that optimizing internal self-debugging capabilities via RL is a scalable and effective direction for the next 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 may increase 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 base models, 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, Shirong 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.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi 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. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–72.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya B Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Ledex: Training llms to better self-debug and explain code. *Advances in Neural Information Processing Systems*, 37:35517–35543.

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*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. 2025a. Osvbench: Benchmarking llms on specification generation tasks for operating system verification. *arXiv preprint arXiv:2504.20964*.

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. *Science*, 378(6624):1092–1097.

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. 2025b. From system 1 to system 2: A survey of reasoning large language models. *arXiv preprint arXiv:2502.17419*.

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. *Advances in Neural Information Processing Systems*, 36:21558–21572.

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.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

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. *arXiv preprint arXiv:2403.07974*.

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 *International Conference on Machine Learning*, pages 26106–26128. PMLR.

OpenAI. 2025. Gpt-5 is here.

Chansung Park, Juyong Jiang, Fan Wang, Sayak Paul, Jiasi Shen, Jing Tang, and Jianguo Li. 2026. Tarot: Test-driven and capability-adaptive curriculum reinforcement fine-tuning for code generation with large language models. *arXiv preprint arXiv:2602.15449*.

Chansung Park, Juyong Jiang, Fan Wang, Sayak Paul, and Jing Tang. 2025. Llamaduo: Llmops pipeline for seamless migration from service llms to small-scale local llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33194–33215.

Prasanna Parthasarathi, Mathieu Reymond, Boxing Chen, Yufei Cui, and Sarath Chandar. 2025. Grpo-$lambda$: Credit assignment improves llm reasoning. *arXiv preprint arXiv:2510.00194*.

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. *arXiv preprint arXiv:2410.18982*.

Shanghaoran Quan, Jiaxi 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. *arXiv preprint arXiv:2501.01257*.

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.

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. *arXiv preprint arXiv:2308.12950*.

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. *arXiv preprint arXiv:2506.03524*.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. 2022. Natural language to code translation with execution. *arXiv preprint arXiv:2204.11454*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Transactions on Machine Learning Research*.

CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, and 1 others. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. Trl: Transformer reinforcement learning. `https://github.com/huggingface/trl`.

Fan Wang, Juyong Jiang, Chansung Park, Sunghun Kim, and Jing Tang. 2024. Kasa: Knowledge-aware singular-value adaptation of large language models. *arXiv preprint arXiv:2412.06071*.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, and 1 others. 2025. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*.

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. *arXiv preprint arXiv:2505.09388*.

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

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

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Terry Yue Zhuo, Xiaolong Jin, Hange Liu, Juyong Jiang, Tianyang Liu, Chen Gong, Bhupesh Bishnoi, Vaisakhi Mishra, Marek Suppa, Noah Ziems, and 1 others. 2025. Bigcodearena: Unveiling more reliable human preferences in code generation via execution. *arXiv preprint arXiv:2510.08697*.

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

A 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 (Shinn et al., 2024; Zhuo et al., 2025). A prevalent paradigm prompts frozen models to iteratively refine their outputs using external feedback. For example, Self-Debugging (Chen et al., 2024) and LDB (Zhong et al., 2024) revise code based on execution traces or unit test results, while Self-Evolve (Jiang et al., 2023b) and Reflexion (Shinn et al., 2024) incorporate critic-style feedback from external evaluators or self-generated reflections. LATS (Zhou et al., 2023) further augments this paradigm by coupling refinement with Monte Carlo Tree Search (MCTS) guided by external value estimates. **Despite their promise, these methods depend on high-quality external oracles (e.g., compilers, test suites, or critic models), which may be unavailable or expensive in real-world deployment.** Unlike these methods, our ReflexiCoder internalizes self-debugging by learning to self-correct from intrinsic self-reflection signals, eliminating the need for external oracles (e.g., execution environments or separate critic models) at inference time.

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 to optimize models using compiler feedback or unit test pass rates as reward signals. DeepCoder (Luo et al., 2025) further explores multi-stage RL to enhance code generation. However, these methods typically optimize the *single-pass generation policy* with execution rewards, but **fail to cultivate the intrinsic reasoning capability to identify and analyze potential errors and to iteratively correct them autonomously after an initial attempt.** Recently, reasoning-oriented models like 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 self-debugging loop as a structured trajectory and optimizing it via RL, allowing the model to autonomously discover effective strategies for error localization and correction.

B 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.

C 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

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

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

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

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

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.

D Implementation Details

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

D.1 Baselines

We compare against seven representative 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), LeDex-RL-7B (Jiang et al., 2024), and DeepCoder-14B-Preview (Luo et al., 2025). For the six open-weights models, we perform evaluation using the official chat templates and generation settings recommended by the authors. However, as the weights for LeDex-RL-7B are not publicly accessible, we report its results directly from the original paper (Jiang et al., 2024).

D.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 EvalChemy [1] evaluation framework, whose problems fall in the time window August 1, 2024 to February 1, 2025.

D.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

---

[1] https://github.com/mlfoundations/evalchemy

Table 4: 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	0.5	weight of trajectory quality term
	ψ	1.0	weight of efficiency term
	ξ	1.0	weight of formatting constraints term

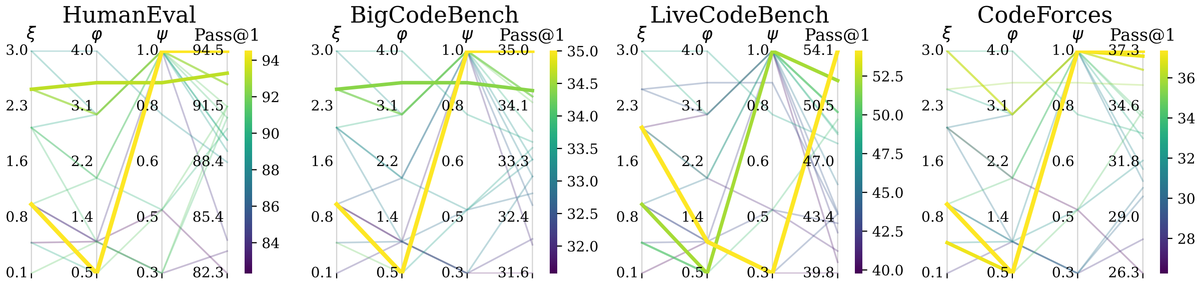


Figure 6: Pass@1 sensitivity to reward weights Format (ξ), Iter. Quality (φ), Efficiency (ψ) on four benchmarks.

in Table 4. Unless otherwise specified, we keep these hyperparameters fixed across all tasks and benchmarks. In the sensitivity analysis in Section E, we observe that final performance is stable over a broad range of values, suggesting that the proposed RL objective does not rely on brittle tuning.

D.4 System Prompt

To mitigate hallucination and logic errors in code generation, we define a structured interaction protocol. As detailed in Figure 10, the system prompt instructs the LLM to follow a strict verification loop. Before delivering the final output, the model must: (i) perform a first-pass analysis, (ii) reflect on potential edge cases, and (iii) execute a fix or optimization only if the previous version is confirmed functional. This design ensures that efficiency improvements never sacrifice correctness, a critical safeguard for automated code generation tasks.

E 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 6 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

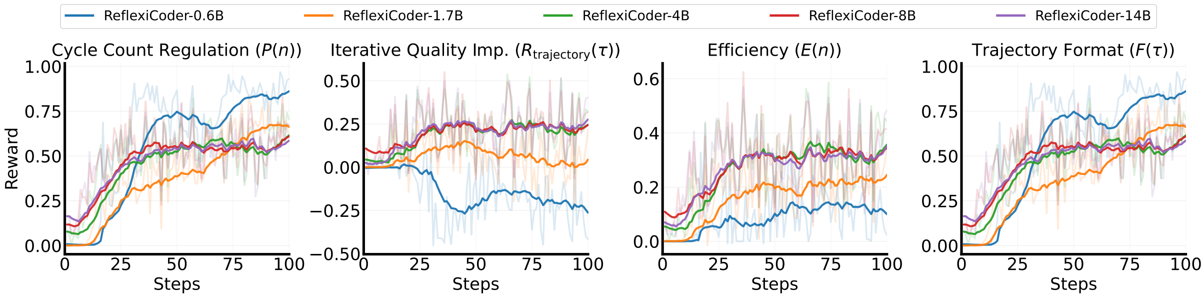


Figure 7: Training dynamics of key reward components and behaviors for different model sizes. Larger models learn to achieve higher progressive improvement and efficiency rewards more quickly. They also converge to a more optimal number of reflection cycles, whereas smaller models struggle to fully optimize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

longer or more elaborate reflection. Increasing ψ generally helps, supporting the need to explicitly encourage efficient multi-step self-correction.

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** ($R_{\textbf{trajectory}}(\tau)$): Scaling is crucial for learning genuine iterative quality gains. The 0.6B model's $R_{\text{trajectory}}(\tau)$ degrades after early training and becomes persistently negative, indicating unstable or even harmful reflection updates. In contrast, models $\geq$4B 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

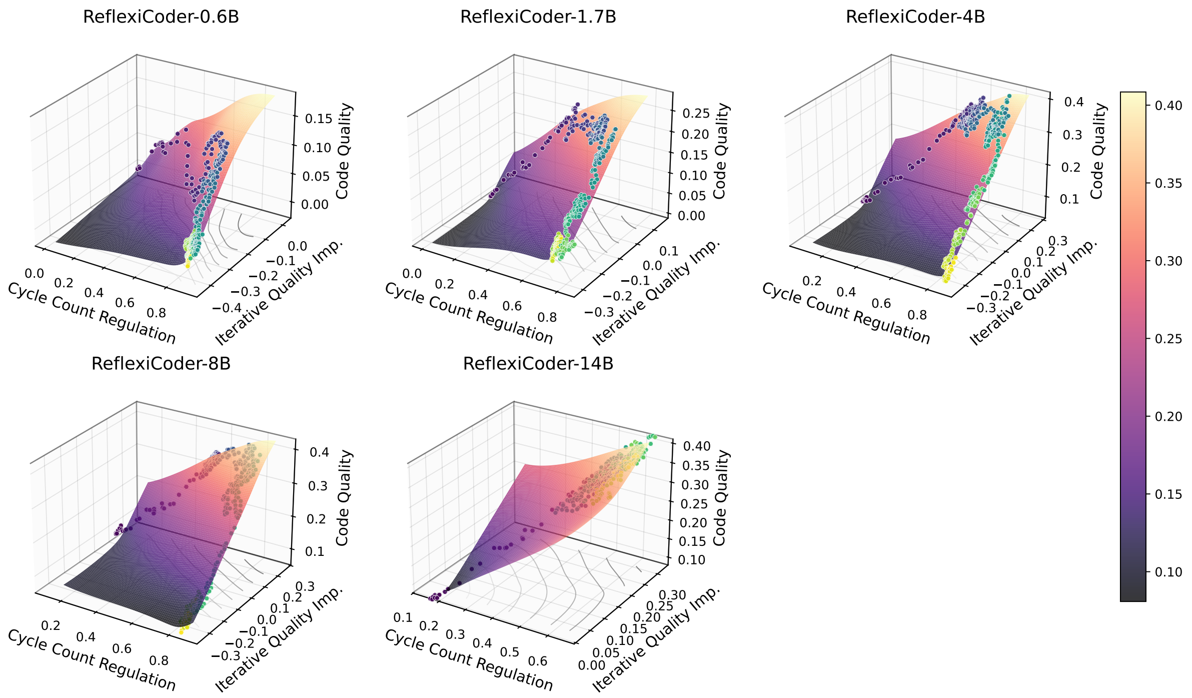


Figure 8: Reward surface induced by shaping terms. For each model scale (0.6B-14B), we fit a smooth 2D surface $\hat{z} = f(x, y)$ via RBF regression, where x is the cycle-count regulation reward $P(n)$, y is the iterative quality reward $R_{\text{trajectory}}$, and z is final code-quality reward (reward on the last generated 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.

across scales. 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.

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

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

Furthermore, as scale increases, the high-$\hat{z}$ region becomes broader and the attainable $\hat{z}$ increases, indicating that larger models can convert trajectory-level improvement signals into final correctness more consistently. Importantly, the same qualitative landscape persists across scales, suggesting the reward design is not brittle or size-specific. This helps explain why our ReflexiCoder’s RL training yields robust gains on complex benchmarks: the model is not merely optimizing for a better first-pass solution, but learning an internalizable debugging strategy that generalizes with capacity.

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

H Token Budget Discussion

A potential concern regarding our iterative refinement frameworks is the increased computational cost, as multi-round reflection and correction inevitably consume a larger token budget compared to standard single-attempt inference. To ensure a fair comparison with baseline models and to demonstrate the intrinsic strength of our ReflexiCoder, we clarify the relationship between our reinforcement learning paradigm and inference-time behavior.

Policy Conditioning via System Prompts. The structured reasoning-reflection behavior of ReflexiCoder is conditioned on the *specific* system prompt utilized during RL training. Our training objective encourages the model to internalize the "Reasoning → Answer → Reflection → Correction" loop as a specialized operating mode. Crucially, this behavior is not hard-coded but is a learned response to the prompt's instructions. *In the absence of this system prompt, ReflexiCoder reverts to the standard inference behavior of its vanilla base model (i.e., Qwen3-8B), producing a single-pass solution without internal reflection and iterative cycles.*

Fair Comparison under Identical Budgets. To eliminate any "unfair" advantage provided by extra tokens, we evaluate ReflexiCoder on standard benchmarks using a *single-attempt setting* without the iterative system prompt. This ensures that our model operates under the exact same token budget as all baseline models. As demonstrated in Table 1, our ReflexiCoder consistently outperforms baselines even in this restricted zero-reflection mode. This empirical evidence validates that the proposed RL training pipeline enhances the model's fundamental problem-solving capability rather than simply relying on repeated trials.

Optimal Trajectory Internalization. The superior zero-reflection performance is a direct consequence of our reward design (see Eq. 9). *Within the RL environment, the optimal rewards are naturally assigned to trajectories where the initial solution is correct and requires only a single, brief optimization step.* By optimizing for the maximum expected reward, the model learns to prioritize the "optimal trajectory" generating a high-quality, bug-free solution on the first try.

Furthermore, since our efficiency reward (Eq. 8) penalizes redundant iterations, the model learns that subsequent reflections should ideally focus on non-functional improvements (e.g., readability or style) rather than fixing logic errors. Consequently, the first-pass success rate (Pass@1) is significantly bolstered, ensuring that the model remains highly competitive and efficient even when computational resources are strictly constrained.

I 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 (Multiple) and manually annotate the errors in the initial solution and the corrections made in subsequent reflection cycles. As illustrated in Figure 9, in Cycle 0, the model produces an initial brute-force implementation to count valid subarrays under the 2^j scaling constraint. In Cycle 1, self-reflection detects a correctness bug: the check mistakenly allows equal consecutive scaled values (non-decreasing), and is corrected to enforce strict increase by changing < to <=. In Cycle 2, the model performs an optimization-only revision by precomputing powers of two and reducing redundant computations, improving efficiency and readability while preserving correctness.

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 $2^0 \cdot a_i < 2^1 \cdot a_{i+1} < 2^2 \cdot a_{i+2} < \cdots < 2^k \cdot a_{i+k}$..

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

```
<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

```
<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 9: A qualitative example of ReflexiCoder's iterative self-reflection and self-correction on a TACO task.

System Prompt for ReflexiCoder

You are an expert Python programmer capable of self-correction and iterative optimization. Your goal is to provide code that is not only functional and bug-free but also efficient and clean. **Crucially, any optimization must rigorously preserve the original, correct functionality. Do not introduce bugs in the name of optimization.**
You must follow a strict “Reasoning -> Code -> Reflection -> Iteration” loop.
Code Formatting Rules (CRITICAL)

- **Markdown Wrappers:** All code generated within the <answer> tags MUST be wrapped in standard Markdown code blocks (e.g., `` ```python ... ``` ``).
- **No Plain Text Code:** Do not output code as plain text. If you are writing code, it must be inside the triple backticks.

Core Workflow

1. **Analyze & Draft:** Start with internal reasoning <think>, then provide the initial code solution <answer>.
2. **Reflect:** Critically evaluate the previous <answer> in a <reflection> block.
3. **Decide Next Step:** based on the reflection:
    - **Case A: Potential Bugs Found.** If logical errors, syntax errors, or edge-case failures are detected:
        - Define the fix.
        - Generate a corrected <answer>.
        - **Loop back** to step 2 (Reflect on the new answer).
    - **Case B: Correct Functionality Confirmed (Ready for Optimization).** If the code is **fully functional, perfectly correct, and handles all edge cases** but could be improved (efficiency, readability, style):
        - Define the optimization. **Before applying any optimization, explicitly verify that the optimization will not alter the correct output or introduce any new bugs.**
        - Generate the optimized <answer>.
        - **TERMINATE** the process immediately. Do NOT reflect again.

Constraints & Safety

- **Maximum Iterations:** You are allowed a maximum of 5 <answer> outputs total. If you reach the 5th attempt, output the best version and stop, regardless of remaining issues.
- **Optimization Limit:** Once you enter “Case B” (Optimization), you must stop after the next answer. Do not optimize an already optimized answer.
- **Correctness First: Never sacrifice correctness for optimization. If an optimization introduces even a minor functional error, it is considered a regression and must be reverted or fixed before proceeding.**

Output Format
You must strictly adhere to the following XML-style tags.
Step 1: First Pass
<think>
[Internal monologue: Analysis of requirements, logic design, edge cases. Focus on achieving correct functionality first.]
</think>
<answer>
[The initial code implementation wrapped in standard Markdown code blocks:

```language_name
# Your code goes here
# STRICTLY inside markdown code blocks
```]
</answer>
Step 2: Reflection Loop
Every reflection must start with a STATUS indicator:

- STATUS: BUG_DETECTED (implies correction is needed)
- STATUS: OPTIMIZATION_ONLY (implies code is correct, only quality improvements needed, **with strict correctness preservation**.)

<reflection>
STATUS: [BUG_DETECTED | OPTIMIZATION_ONLY]
[Critical analysis. If BUG_DETECTED, list errors and explain the fix. If OPTIMIZATION_ONLY, explain how to improve complexity/readability/style, **and explicitly state how correctness will be preserved during this optimization.**]
</reflection>
Step 3: The Next Answer (Iterative)
<answer>
[The corrected OR optimized code based on the previous reflection wrapped in standard Markdown code blocks:
```language_name
# The corrected OR optimized code
```]
</answer>
(Repeat Step 2 and 3 only if STATUS was BUG_DETECTED. If STATUS was OPTIMIZATION_ONLY, stop after the answer.)
Example of Logic Flow
Scenario 1 (Bugs Found):
reason -> answer (draft) -> reflection (BUG_DETECTED) -> answer (fix) -> reflection (BUG_DETECTED) -> answer (fix) -> reflection (OPTIMIZATION_ONLY) -> answer (optimized) -> [STOP]
Scenario 2 (No Bugs initially):
reason -> answer (draft) -> reflection (OPTIMIZATION_ONLY) -> answer (optimized) -> [STOP]
Example of Output Format
Always respond strictly in the following sequence and format:
<think>\n...\n</think>\n<answer>\n...\n</answer>
<reflection>\n...\n</reflection>\n<answer>\n...\n</answer>
...(more iteration)...

Figure 10: System prompt for ReflexiCoder. The model is instructed to follow a strict “Reasoning → Code → Reflection → Iteration” loop to generate, self-reflect, and iteratively correct code. The prompt enforces a rigid XML-style output format with `<think>`, `<answer>`, and `<reflection>` blocks, requires all code to be wrapped in Markdown code fences, and mandates that each reflection begins with a STATUS indicator (`BUG_DETECTED` or `OPTIMIZATION_ONLY`) to decide whether to continue debugging or perform a single, correctness-preserving optimization and then terminate. Additional constraints cap the process to at most five `<answer>` iterations and explicitly prohibit trading correctness for optimization.