REFINERL: ADVANCING COMPETITIVE PROGRAMMING AND BEYOND WITH SELF-REFINEMENT REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

While large language models (LLMs) have demonstrated strong performance on complex reasoning tasks such as competitive programming (CP), existing methods predominantly focus on single-attempt settings, overlooking their capacity for iterative refinement. In this paper, we present **RefineRL**, a novel approach designed to unleash the self-refinement capabilities of LLMs for CP problem solving. RefineRL introduces two key innovations: (1) *Skeptical-Agent*, an iterative self-refinement agent equipped with local execution tools to validate generated solutions against public test cases of CP problems. This agent always remains skeptical of its own outputs and thereby enforces rigorous self-refinement even when validation suggests correctness. (2) A reinforcement learning (RL) solution to incentivize LLMs to self-refine with only standard RLVR data (*i.e.*, problems paired with their verifiable answers). Extensive experiments on Qwen3-4B and Qwen3-4B-2507 demonstrate that our method yields substantial gains: after our RL training, these compact 4B models integrated with the Skeptical-Agent not only outperform much larger 32B models but also approach the single-attempt performance of 235B models. Notably, although our models are trained solely on CP coding data, RefineRL also substantially improves their performance on mathematical reasoning benchmarks, suggesting that the learned self-refinement behavior can transfer beyond coding-specific tasks. These findings highlight self-refinement as a promising direction for scaling LLMs' general reasoning ability.

1 INTRODUCTION

Large language models (LLMs) have demonstrated strong reasoning abilities in math competitions and competitive programming (CP). Proprietary systems like AlphaGeometry (Trinh et al., 2024; Chervonyi et al., 2025), OpenAI o1 (El-Kishky et al., 2025), and Gemini (Comanici et al., 2025) have proven that reasoning LLMs can become medal winners in the most challenging mind competitions. Meanwhile, open-source models like DeepSeek-R1 (Guo et al., 2025) and Qwen3 (Yang et al., 2025) have achieved remarkable performance on rigorous benchmarks such as AIME and LiveCodeBench. While the success of these models is largely attributed to long chain-of-thought (Wei et al., 2022) that can consume tens of thousands of tokens during generations, LLMs' potential for *iterative improvement* remains under-explored. Indeed, prevailing evaluations use *single-attempt* settings, where models are expected to derive correct solutions in a single inference pass.

More recent studies adopt self-correction techniques for iterative improvement in mathematical proving (Lin et al., 2025; Zhou et al., 2025). In each turn, the model uses the Lean-4 prover to check the generated proof and, if incorrect, incorporates feedback to revise it. In CP, Shi et al. (2024) and Li et al. (2025) represent primary open-source efforts leveraging self-correction. They employ Reflexion (Shinn et al., 2023) to iteratively correct solutions until passing all public test cases. However, such methods generally rely on sparse visible verification signals to trigger improvement, leading to a *false-positive* issue: outputs may satisfy available checks while still failing hidden cases or more comprehensive evaluation. In CP, this manifests as solutions passing public tests but failing private tests, since public test cases are only a sparse proxy for the hidden test distribution. Further, as a purely test-time method, Reflexion relies on the base model's intrinsic iterative improvement capability, lacking explicit training to optimize the improvement process.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To address these challenges, we introduce **RefineRL**, an approach designed to unleash the latent reasoning capabilities of LLMs for iterative improvement in competitive programming. Our framework consists of two components: the ***Skeptical-Agent*** and ***Self-Refinement Reinforcement Learning***. Concretely, the Skeptical-Agent employs a fine-grained agentic workflow tailored to different judging statuses on public test cases to perform refinement. Unlike previous approaches that blindly trust public validation, our agent remains skeptical, enforcing rigorous self-refinement even after passing all public test cases. However, such self-refinement capability either does not arise naturally in LLMs or remains too weak to be effective. We thus propose Self-Refinement RL to explicitly incentivize this capability using only standard RLVR data (*i.e.*, problems paired with their verifiable answers). By constructing a dataset solely from the model's skeptical refinement trajectories and using a new *squared-incentive reward* function, we train LLMs to navigate the complex self-refinement process. Self-Refinement RL thus enables our approach to go beyond static prompt engineering.

We evaluate RefineRL on two 4B reasoning LLMs, Qwen3-4B and Qwen3-4B-2507. Results show that our method yields substantial gains with high data efficiency: using only 4.5K problems from the HardTest dataset (He et al., 2025) to construct training samples, we improve the problem-solving capabilities of compact 4B models. Notably, after Self-Refinement RL, RefineRL-4B and RefineRL-4B-2507 equipped with the Skeptical-Agent not only outperform the single-attempt performance of larger 32B models but also rival that of significantly larger 235B models. Moreover, although our models are trained solely on CP coding data, RefineRL also brings substantial improvements on mathematical reasoning benchmarks, suggesting that the learned self-refinement behavior in overcoming "false-positive reasoning" can generalize beyond coding tasks. These findings indicate that self-refinement remains an under-explored avenue for scaling LLM reasoning ability, with significant potential for improving general reasoning beyond parameter scaling alone.

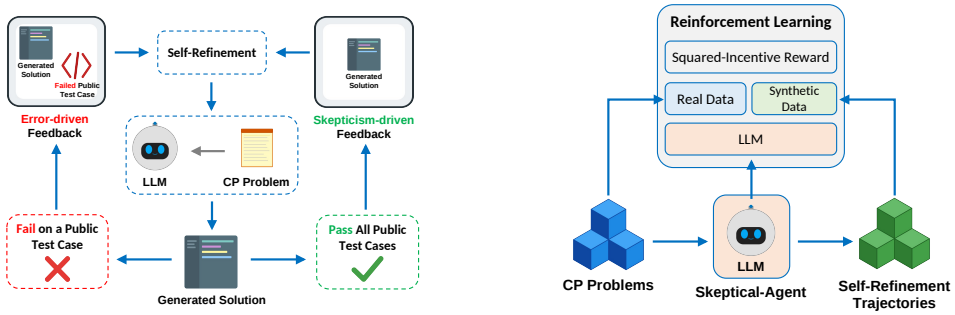
2 Related Work

Code generation of LLMs. Reasoning LLMs are good at generating program code with long chain-of-thought (CoT) (Liu et al., 2024; Yang et al., 2025; Comanici et al., 2025). For competitive programming, a specific code generation task, Ahmad et al. (2025) show that training LLMs on long CoT data distilled from frontier models improves their ability to solve CP problems. He et al. (2025), Wang et al. (2025), and Fu et al. (2025) find that synthesized high-quality test cases boost RL training with verifiable rewards. Hu et al. (2025) and Wu et al. (2026) further suggest using fully synthesized CP problems for training. Beyond model training, several test-time scaling strategies, such as example test case rejection, longest response selection, clustering, and unittest generation, are designed for CP (El-Kishky et al., 2025; Yu et al., 2025; Samadi et al., 2025; Ma et al., 2026; Zhang et al., 2026; Li et al., 2026). Our work aims to incentivize a new Skeptical-Agent capability for competitive programming via RL training.

Self-correction of LLMs. Self-correction is a powerful test-time scaling strategy that empowers LLMs to iteratively critique and improve their responses based on internal or external feedback (Shinn et al., 2023; Chen et al., 2024; Li et al., 2025; Madaan et al., 2023; Novikov et al., 2025). Previous works have demonstrated its effectiveness for coding and mathematical reasoning tasks (Shi et al., 2024; Chen et al., 2025a; Song et al., 2025), and recent studies indicate that LLMs can further internalize this capability through post-training (Chen et al., 2024; Jain et al., 2025a; Kumar et al., 2025). Traditional approaches often rely on the complete history of multi-turn interactions, leading to excessive context length that limits practicality. More recent studies in competitive-level math proving (Zhou et al., 2025; Lin et al., 2025; Shang et al., 2025; Chen et al., 2025b) suggest decomposing the correction process into multiple single-turn feedback-response pairs, allowing the model to focus on immediate corrections without retaining the full trajectory. Our paper focuses on designing a self-refinement strategy more robust than self-correction for solving CP problems and on effectively using RL (post-)training to enhance such a capability.

3 Preliminaries

Competitive Programming (CP). A CP problem can be formalized as a triple $(x, T^{\text{pub}}, T^{\text{pri}})$, where x is the problem statement, $T^{\text{pub}} = \{(i_k^{\text{pub}}, o_k^{\text{pub}})\}_{k=1}^{|T^{\text{pub}}|}$ is the set of *public* test cases, $(i_k^{\text{pub}}, o_k^{\text{pub}})$ is the input-output pair of the k-th public test case, $T^{\text{pri}} = \{(i_k^{\text{pri}}, o_k^{\text{pri}})\}_{k=1}^{|T^{\text{pri}}|}$ is the set of *private* test



(a) Skeptical-Agent for CP Problems.

(b) Self-Refinement RL.

Figure 1: Overview of our RefineRL approach. (a) **Skeptical-Agent** employs local tools to evaluate generated solutions against visible public test cases and enforces rigorous self-refinement even when evaluation results suggest correctness. (b) **Self-Refinement RL** utilizes self-generated refinement trajectory data from the Skeptical-Agent on real-world CP problems data, along with a novel *Squared-incentive reward* function, to advance the self-refinement capabilities of LLMs.

cases, and $(i_k^{\text{pri}}, o_k^{\text{pri}})$ is the input-output pair of the k-th private test case. In a CP contest, the problem statement x and the public test cases T^{pub} are visible to CP contestants, while the private test cases T^{pri} are not. The goal of a CP contestant is to write a code r_{code} based on x and T^{pub} that passes T^{pri}.

To determine whether the proposed solution code r_{code} successfully solves the given problem $(x, T^{\text{pub}}, T^{\text{pri}})$, the contest coordinator uses two oracle functions, *i.e.*, an execution function $f(\cdot,\cdot)$ and a judgment function $g(\cdot,\cdot)$, to evaluate r_{code} on the private test set T^{pri}. Concretely, for the k-th private test case $(i_k^{\text{pri}}, o_k^{\text{pri}}) \in T^{\text{pri}}$, the execution output and judging status are computed as follows,

$$\hat{o}_k^{\text{pri}} = f(r_{\text{code}}, i_k^{\text{pri}}), \qquad j_k^{\text{pri}} = g(\hat{o}_k^{\text{pri}}, o_k^{\text{pri}}), \tag{1}$$

where $\hat{o}_k^{\text{pri}}$ is the execution output, and j_k^{pri} is the judging status, indicating whether r_{code} passes $(i_k^{\text{pri}}, o_k^{\text{pri}})$ and, if not, the failure reason. We adopt six judging statuses common in real-world CP contests: "**Accepted** (AC)", "**Wrong Answer** (WA)", "**Time Limit Exceeded** (TLE)", "**Memory Limit Exceeded** (MLE)", "**Compile Error** (CE)", and "**Runtime Error** (RE)". Among them, "AC" means the code passes, while "WA", "TLE", "MLE", "CE", and "RE" specify the failure reason.

Finally, the proposed solution code x_{code} is determined to successfully solve the given problem $(x, T^{\text{pub}}, T^{\text{pri}})$ if it passes all private test cases of the problem. In other words, x_{code} is considered to be *correct* if and only if the summation $(\sum_{k=1}^{|T^{\text{pri}}|} 1[j_k^{\text{pri}} = \text{AC}])$ equals to $|T^{\text{pri}}|$.

Solving CP problems with LLMs. Let π_θ be a reasoning LLM parameterized by θ. For a given CP problem $(x, T^{\text{pub}}, T^{\text{pri}})$, one can directly prompt the LLM π_θ to solve the problem as follows,

$$(r_{\text{cot}}, r_{\text{sol}}, r_{\text{code}}) \sim \pi_\theta(x), \tag{2}$$

where r_{sol} is the generated CoT, r_{sol} is the generated text solution, and r_{code} is the solution code (*e.g.*, Python code) corresponding to the text solution. In this paper, we focus on generating Python solutions, and the prompt template that we use for Eq. (2) is presented in Figure 9 in Appendix C.

4 REFINERL

4.1 MOTIVATION

In the domain of CP, Shi et al. (2024) and Li et al. (2025) are primary open-source efforts leveraging self-correction for iterative improvement. In their approach, the model analyzes the improvement history and corrects its code until the solution passes all visible public test cases or reaches the maximum steps. However, public test cases are usually sparse and weak, failing to cover the comprehensive distribution of private test cases, particularly edge cases. This limitation creates a significant risk of *false positives*: a solution passing all public test cases can still fail on unseen private ones.

To tackle false positives, we argue that a promising strategy is to integrate *skepticism* into iterative improvement for competitive programming. An LLM should not blindly trust evaluations on public test cases but should remain skeptical, continuing refinement even when public evaluations suggest correctness. Built upon this principle, our Skeptical-Agent utilizes specialized prompts to enforce persistent *self-refinement*, while Self-Refinement RL internalizes this self-refinement capability through post-training, enabling the LLM to discover and resolve defects beyond public test cases. The high-level designs of Skeptical-Agent and Self-Refinement RL are shown in Figure 1.

4.2 SKEPTICAL-AGENT

We now present the detailed design of our Skeptical-Agent. Its overall pipeline is provided in Algorithm 1 in Appendix A.1. Given a CP problem $(x, T^{\text{pub}}, T^{\text{pri}})$, Skeptical-Agent proposes and iteratively refines solutions for up to N iterations, using only the problem statement x and the public test cases T^{pub}. At each iteration, following designs similar to those in Kumar et al. (2025), Jain et al. (2025a), and Lin et al. (2025), Skeptical-Agent constructs feedback for the next generation solely from the previous attempt.

Self-refinement in Skeptical-Agent. Let $(r_{\text{cot}}^{(n)}, r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)})$ be the solution tuple generated in the n-th refinement iteration. To construct feedback $r_{\text{fed}}^{(n)}$ for the $(n+1)$-th generation, we sequentially evaluate the generated code $r_{\text{code}}^{(n)}$ on each public test case from T^{pub}. For the k-th public test case $(i_k^{\text{pub}}, o_k^{\text{pub}})$, we first obtain an execution output $\hat{o}_k^{(n)} := f(r_{\text{code}}^{(n)}, i_k^{\text{pub}})$ (see Eq. (1)) and then calculate a judging status $j_k^{(n)} := g(\hat{o}_k^{(n)}, o_k^{\text{pub}})$, using the oracle execution and judgment functions f and g respectively. The $(n+1)$-th solution generation process of Skeptical-Agent is formalized as follows,

$$(r_{\text{cot}}^{(n+1)}, r_{\text{sol}}^{(n+1)}, r_{\text{code}}^{(n+1)}) \sim \pi_\theta(x, r_{\text{fed}}^{(n)}), \tag{3}$$

where π_θ is the reasoning LLM and $r_{\text{fed}}^{(n)}$ is the n-th feedback constructed from the evaluation of the n-th generated solution on the public test cases set T^{pub}. The prompt template used by Eq. (3) is presented in Figure 9 in Appendix C.

Then, to maintain skepticism, the Skeptical-Agent will use one of the *error-driven* feedback and *skepticism-driven* feedback during refinement, detailed as follows.

Error-driven feedback. During the sequential evaluation, if the solution code fails to pass the k-th public test case, *i.e.*, if $j_k^{(n)} \in \{\text{WA}, \text{TLE}, \text{MLE}, \text{CE}, \text{RE}\}$, then the evaluation will be terminated and the feedback $r_{\text{fed}}^{(n)}$ obtained from the n-th generation is constructed as follows,

$$r_{\text{fed}}^{(n)} = \mathcal{P}_{\text{Err}}(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)}, i_k^{\text{pub}}, o_k^{\text{pub}}, \hat{o}_k^{(n)}, j_k^{(n)}), \tag{4}$$

where $\mathcal{P}_{\text{Err}}$ is the *error-feedback function* that constructs feedback prompts for different failed judging statuses $j_k^{(n)}$ with different failure reason explanations. The prompt template for $\mathcal{P}_{\text{Err}}$ and the failure reason explanation template are in Figure 10 and Figure 11 in Appendix C. This design aims to provide as many failure details as possible to assist the LLM to generate better solutions.

Skepticism-driven feedback. If the solution code $r_{\text{code}}^{(n)}$ successfully passes all public test cases, *i.e.*, if $j_k^{(n)} = \text{AC}$ for every $(i_k^{\text{pub}}, o_k^{\text{pub}}) \in T^{\text{pub}}$, there might still be bugs or wrong ideas in the solution that can cause the code to fail in edge or extreme private test cases from T^{pri}. However, in this setting, there is no test case that can be used for the feedback and assisting the next-time solution generation in FaSC. To tackle such a challenge, we propose to instead construct the feedback solely using the generated solution text $r_{\text{sol}}^{(n)}$ and code $r_{\text{code}}^{(n)}$ as follows,

$$r_{\text{fed}}^{(n)} = \mathcal{P}_{\text{Skep}}(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)}), \tag{5}$$

where $\mathcal{P}_{\text{Skep}}$ is the *Skepticism-feedback function* that constructs prompts asking the LLM to re-check the generated solution $(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)})$. The prompt template for $\mathcal{P}_{\text{Skep}}$ is in Figure 12 in Appendix C, focusing on checking: (1) whether $r_{\text{code}}^{(n)}$ encodes public test cases as short-cuts, (2) whether $(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)})$ is correct, and (3) whether the time and space complexities of $(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)})$ are optimal.

Overall agent-loop workflow. As aforementioned, in the agent-loop workflow, the Skeptical Agent performs up to N self-refinement iterations. We further adopt the following two additional *rules*, with two additional hyperparameters M_1 and M_2, to control the workflow:

- **Restart Rule.** If the agent fails to generate a solution that passes all test cases in T^{pub} for M_1 consecutive attempts, then in the (M_1+1)-th trial, it will discard the feedback from the previous attempt and generate a solution solely based on the problem statement x.
- **Termination Rule.** If the agent generates solutions that passes T^{pub} for M_2 consecutive attempts, it will terminate the refinement and select the last attempt as the final solution.

The **Restart Rule** is designed to prevent the self-refinement process from getting trapped in local optima. Conversely, the **Termination Rule** compels the LLM to re-examine and rectify potential latent bugs in the proposed solution, even when it already satisfies all public test cases.

4.3 Self-Refinement Reinforcement Learning

We now detail our Self-Refinement RL for improving the self-refinement capability of the Skeptical-Agent. We use the GRPO (Guo et al., 2025) algorithm, excluding KL-regularization, to perform RL training. The objective function for training an LLM π_θ is formalized as follows,

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim D,\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)} \frac{1}{G}\sum_{i=1}^{G} \min\Big(s_i\hat{A}_i, \text{clip}\left(s_i, 1-\epsilon, 1+\epsilon\right)\hat{A}_i\Big),$$

where q is the input sample drawn from the training set D, $\{o_i\}_{i=1}^{G}$ is the group of responses, $s_i := \frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)}$ is the importance sampling ratio, $\hat{A}_i := \frac{R_i - \text{mean}(R)}{\text{std}(R)}$ is the advantage for the i-th response, and $\{R_i\}_{i=1}^{G}$ is the group of rewards for each response.

To help internalize the capability of skeptical self-refinement, our Self-Refinement RL also adopts two unique designs: (1) a novel RL training data synthesis paradigm, and (2) a new reward function for better RL training on CP problems.

RL training data synthesis. From the solution generation process formalized in Eq. (3) in Section 4.2, the key to advancing the Skeptical-Agent is to train LLMs to better perform self-refinement with the feedback $r_{\text{fed}}^{(n)}$ that can be either *error-driven feedback* (see Eq. (4)) or *skepticism-driven feedback* (see Eq. (5)). To this end, in our Self-Refinement RL, for a reasoning LLM π_θ, we propose to construct, from a normal CP problem dataset, a training dataset that contains prompts combined with failure/skepticism-feedback synthesized solely based on the LLM π_θ itself.

Concretely, suppose $D_{\text{CP}} := \{(x_q, T_q)\}_{q=1}^{|D|}$ is a CP problem dataset where x_q is the statement of the q-th problem and T_q is the corresponding test cases set mixing of both public and private test cases. We use the LLM π_θ to construct two new self-refinement datasets, $D_{\text{Err}} := \{(x_q, r_{\text{fed},q}^{\text{Err}}, T_q)\}_{q=1}^{|D^{\text{Err}}|}$ and $D_{\text{Skep}} := \{(x_q, r_{\text{fed},q}^{\text{Skep}}, T_q)\}_{q=1}^{|D^{\text{Skep}}|}$, where $r_{\text{fed},q}^{\text{Err}}$ is the error-driven feedback from Eq. (4) and $r_{\text{fed},q}^{\text{Skep}}$ is the skepticism-driven feedback from Eq. (5). The eventual Self-Refinement RL training dataset D_{RL} is formalized as the union of the original and self-refinement data, *i.e.*, $D_{\text{RL}} := D_{\text{CP}} \cup D_{\text{Err}} \cup D_{\text{Skep}}$. During training, for each tuple (x, r_{fed}, T) from the set D_{RL}, we use the prompt template in Figure 9 to construct the actual prompt for performing the rollout process of RL.

The construction of the two self-refinement datasets, D_{Err} and D_{Skep}, is detailed as follows:

Error-feedback set D_{Err}. For each tuple $(x_q, T_q) \in D_{\text{CP}}$ from the original problem set, we first use the LLM π_θ to generate the solution text $r_{\text{sol},q}$ and code $r_{\text{code},q}$ following Eq. (2). We then evaluate $r_{\text{code},q}$ on T_q using the execution and judging functions f and g in Eq. (1). If $r_{\text{code},q}$ passes all test cases in T_q, we exclude this problem from D_{Err}. Otherwise, for a failed test case $(i_{q,k}, o_{q,k}) \in T_q$, we record the execution output $\hat{o}_{q,k} := f(r_{\text{code},q}, i_{q,k})$ and judging status $j_{q,k} := g(\hat{o}_{q,k}, o_{q,k})$. We then construct the error-driven feedback as $r_{\text{fed},q}^{\text{Err}} := \mathcal{P}_{\text{Err}}(r_{\text{sol},q}, r_{\text{code},q}, i_{q,k}, o_{q,k}, \hat{o}_{q,k}, j_{q,k})$ following Eq. (4), and add the tuple $(x_q, r_{\text{fed},q}^{\text{Err}}, T_q)$ to D_{Err}.

Skepticism-feedback set D_{Skep}. For each tuple $(x_q, T_q) \in D_{\text{CP}}$ from the original problem set, we similarly use the LLM π_θ to generate the solution text $r_{\text{sol},q}$ and code $r_{\text{code},q}$. Without

checking whether $r_{\text{code},q}$ passes T_q, we directly construct the skepticism feedback as $r^{\text{Skep}}_{\text{fed},q} := \mathcal{P}_{\text{Skep}}(r_{\text{sol},q}, r_{\text{code},q})$ following Eq. (5). We then add the tuple $(x_q, r^{\text{Skep}}_{\text{fed},q}, T_q)$ to D_{Skep} with probability 20% if $r_{\text{code},q}$ passes all test cases in T_q, and with probability 40% otherwise.[1]

Squared-incentive reward. Existing studies of RL training on CP problems usually use a *pass-or-fail* reward function where a generated solution code x_{code} receives a non-trivial reward from a CP problem (x, T) if and only if it passes all test cases from the set T (He et al., 2025; Wang et al., 2025). While such a pass-or-fail reward is close to settings of real-world CP contests such as CodeForces or ICPC where contestants need to pass all test cases to get scores from a single problem, we note that it may hinder RL training. Specifically, for a given CP problem (x, T), if it is so difficult that the LLM being trained cannot generate any correct solution to pass the test case set T, then the LLM cannot obtain any non-trivial reward and thus cannot learn anything from (x, T).

To address this technical challenge, our Skeptical RL training adopt a new ***Squared-incentive Reward*** function that is specifically designed for RL training on CP data. Given a CP problem (x, T) and a generated solution code r_{code}, the squared-incentive reward for r_{code} is calculated as follows,

$$R(r_{\text{code}}, T) := (k^*/|T|)^2,$$

where k^* is either $|T|$ or one less than the index of the first test case in T that r_{code} fails to pass. Our squared-incentive reward has two advantages. **Firstly**, it allows the LLM to obtain meaningful learning signals from difficult CP problems even when it cannot produce a fully correct solution during RL. **Secondly**, it discourages hacking early test cases and instead incentivizes the LLM to produce the most correct possible solution to pass later, high-reward test cases.

5 Experiments

5.1 Experimental Setup

Inference-time strategies. For our proposed Skeptical-Agent (see Algorithm 1 in Appendix A.1), we denote it as **Skeptical@16** and set its query budget N to 16 and the remaining two hyperparameters M_1 and M_2 to 4 and 8 respectively.

We also adopt five inference-time baselines: (1) **Pass@1**: Directly querying the LLM once using Eq. (2) and extracting solutions from its response. (2) **Random@16**: Generating 16 responses in parallel and random choose one that passes all public test cases. (3) **LongCoT@16**: Generating 16 responses in parallel and random choose one that passes all public test cases and has the longest CoT length. (4) **RejSamp@16**: Repeatedly querying the LLM up to 16 times until a solution code that can pass all public test cases is generated. (5) **Reflexion@16** (Shi et al., 2024): Repeatedly querying the LLM up to 16 times with the full historical generation results until a solution code that can pass the problem is generated. Note that Random@16 and LongCoT@16 are parallel sampling-based baselines, while RejSamp@16 and Reflexion@16 are sequential sampling-based baselines.

Training data of RL. We select 4.5K CP problems from the HardTest dataset (He et al., 2025) that are *suitable* for learning self-refinement to form the base problem set D_{CP}. We then follow the pipeline described in Section 4.3 to construct the final Self-Refinement RL training data. We set the maximum generation length to 32,768 for Qwen3-4B and 65,536 for Qwen3-4B-2507, resulting in approximately 4.2K self-refinement trajectories for each model. See Appendix A.2 for full details.

RL training. We perform GRPO-based RL (Guo et al., 2025) on two 4B LLMs, Qwen3-4B and Qwen3-4B-2507 (Yang et al., 2025). The maximum rollout length is 32,768 tokens for Qwen3-4B and 65,536 for Qwen3-4B-2507. The resulting models are denoted **RefineRL-4B** and **RefineRL-4B-2507**. See Appendix A.4 for full RL hyperparameters and details of efficient reward calculation.

Evaluations. We use two versions of LiveCodeBench (Jain et al., 2025b), **v5** (2024.10-2025.01; 167 problems) and **v6** (2025.01-2025.03; 175 problems), and the USACO benchmark (Shi et al.,

[1]The $20\%/40\%$ rates are heuristic. For failed solutions, skepticism feedback provides no explicit failure signal, making successful correction rare and often leading to weak group-relative rewards in GRPO, as few rollouts receive substantially higher rewards than the others. We therefore keep only 40% of such samples rather than all of them. For passed solutions, preserving correctness is easier and may yield uninformative uniformly high rewards, so we keep them at a lower rate of 20%.

Table 1: Performance on various CP and math benchmarks. We compare our RefineRL models (4B and 4B-2507 versions) against baselines ranging from 4B to 235B parameters. **Bold** indicates the best performance, and underline indicates the second-best performance. Our best RefineRL model equipped with Skeptical-Agent (RefineRL-4B-2507 + Skeptical@16) outperforms most 32B baselines and achieves performance comparable to the 235B model.

	CP Benchmarks			Math Benchmarks		
Model	LCB-v5	LCB-v6	USACO	AIME24	AIME25	AIME26
Qwen3-235B-A22B (Yang et al., 2025)	67.25	59.93	60.71	87.29	77.92	82.71
Qwen3-235B-A22B-2507 (Yang et al., 2025)	79.08	72.50	75.51	93.33	90.63	90.00
Qwen3-8B (Yang et al., 2025)	55.39	49.32	46.17	77.92	64.58	64.58
Dpsk-R1-0528-Qwen3-8B (Guo et al., 2025)	59.84	51.84	43.16	79.37	66.46	69.38
X-Coder-RL-Qwen3-8B (Wu et al., 2026)	55.80	48.39	43.40	33.75	26.67	33.13
Qwen3-30B-A3B (Yang et al., 2025)	60.37	56.00	52.71	82.08	70.21	70.42
Qwen3-30B-A3B-2507 (Yang et al., 2025)	70.55	64.61	64.68	91.46	85.21	87.33
Qwen3-32B (Yang et al., 2025)	62.69	59.25	56.11	82.92	72.71	74.58
OCR-Nemotron-1.1-32B (Ahmad et al., 2025)	70.47	62.25	61.97	30.21	22.08	21.46
OR-Nemotron-32B (Ahmad et al., 2025)	67.03	66.07	**66.92**	88.54	82.50	84.79
Qwen3-4B (Yang et al., 2025)	52.51	48.29	40.76	72.71	63.96	65.00
Qwen3-4B-2507 (Yang et al., 2025)	62.65	55.21	52.44	84.38	75.63	80.83
RefineRL-4B (Ours)	56.40	48.79	44.75	71.67	65.21	64.79
+ Skeptical@16 (Ours)	64.07	56.54	50.61	78.96	72.92	72.08
RefineRL-4B-2507 (Ours)	70.40	61.57	59.81	87.92	77.50	82.08
+ Skeptical@16 (Ours)	**75.97**	**67.82**	64.66	**92.08**	**88.75**	**88.33**

2024) (307 problems) to evaluate the capability of LLMs in solving CP problems. We also use three years of AIME datasets (AIME24/25/26) to evaluate the math reasoning capability. For each LLM, we set the sampling temperature and top-p to 0.6 and 0.95. For the max generation token length, we set 32,768 for Qwen3-4B and RefineRL-4B and 65,536 for other models. All models are evaluated using our self-implemented evaluation toolkit. See Appendix A.5 for details.

5.2 Results Analysis

We present the RL training logs in Appendix B. Here, we focus on analyzing the CP problem-solving performance gains by the Skeptical-Agent and Self-Refinement RL.

CP performance against baseline models. We compare the CP performance of our RefineRL-4B and RefineRL-4B-2507 models against strong baselines in Table 1. We observe two key findings. **Firstly**, although Self-Refinement RL targets self-refinement, it also substantially improves standard Pass@1 performance. For example, RefineRL-4B achieves performance comparable to 8B models (*e.g.*, Qwen3-8B), while RefineRL-4B-2507 rivals 32B models (*e.g.*, OCR-Nemotron-1.1-32B). **Secondly**, with our inference-time Skeptical-Agent strategy (*i.e.*, Skeptical@16), RefineRL-4B-2507 outperforms most 32B baselines and approaches the larger Qwen3-235B-A22B-2507. Specifically, it reduces the performance gap to Qwen3-235B-A22B-2507 from around 17% to below 5% on both LiveCodeBench v5 and v6, and from 23% to 11% on USACO. These results show that our approach effectively unlocks the self-refinement capability of LLMs, enabling compact 4B models to achieve strong CP performance with efficient training and inference.

Generalization to mathematical reasoning. We then evaluate whether the self-refinement capability learned from CP data generalizes to mathematical reasoning. Since there are no executable test cases for math problems, we apply only skepticism-driven refinement ($M_2 = 8$ and $N = 16$) in Skeptical-Agent. As shown in Table 1, although RefineRL models are trained solely on CP coding data, they still achieve clear improvements across three years of AIME benchmarks. In particular, our best model, RefineRL-4B-2507 with Skeptical-Agent, outperforms all 30B/32B baselines and reduces the gap to Qwen3-235B-A22B-2507 from roughly 10–15% to below 2%. These results suggest that our method not only improves CP performance, but also enhances LLMs' general self-refinement capability, helping them overcome false-positive reasoning beyond coding tasks.

Comparison with inference-time baselines. We compare Skeptical-Agent (*i.e.*, Skeptical@16) with various inference-time baselines on RefineRL-4B-2507 in Table 2 (performance) and Figure 2 (time cost). Similar results on RefineRL-4B are provided in Appendix B.3. Our Skeptical@16 consistently outperforms parallel-sampling baselines (*i.e.*, Random@16 and LongCoT@16) across all three benchmarks. Notably, compared with Random@16, Skeptical@16 achieves both better

Table 2: Performance of inference-time strategies on RefineRL-4B-2507. Our Skeptical@16 strategy outperforms all compared baselines across all settings.

Model & Strategy	LCB-v5	LCB-v6	USACO
RefineRL-4B-2507 (Ours)	70.40	61.57	59.81
+ Random@16	74.06	64.14	61.39
+ LongCoT@16	69.65	58.82	51.97
+ RejSamp@16	74.25	64.71	62.32
+ Reflexion@16	74.36	64.29	62.77
+ Skeptical@16 (Ours)	**75.97**	**67.82**	**64.66**

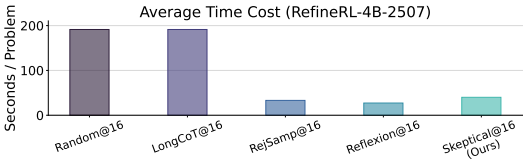


Figure 2: Time cost of inference-time strategies on RefineRL-4B-2507. Skeptical@16 achieves a speed comparable to other sequential-sampling baselines while being faster than parallel-sampling baselines.

performance and about $4\times$ faster inference. This shows that our gains come from more effective use of the sampling budget, rather than simply increasing parallel test-time compute.

We further compare with other sequential-sampling baselines (*i.e.*, RejSamp@16 and Reflexion@16) to isolate the source of improvement. Skeptical@16 outperforms RejSamp@16, indicating that the gains do not merely come from stochastic re-sampling until public tests pass. It also surpasses Reflexion@16, which may prematurely accept false-positive solutions, showing that persistent skepticism helps uncover and fix latent bugs beyond public test cases.

Effect of RL training on model responses. We analyze the behavioral changes induced by Self-Refinement RL in Table 3. After RL training, the average number of self-refinement rounds slightly decreases, while the average response length increases substantially. This suggests that our objective does not simply encourage the model to "try more times", but promotes deeper reasoning within each attempt. By generating longer and more detailed CoT, the model effectively trades the quantity of attempts for reasoning quality. Crucially, this substantial increase in reasoning length for RefineRL-4B-2507 correlates with the superior performance gains reported in our main results, reinforcing the hypothesis that extended, comprehensive CoT is instrumental in solving complex CP problems.

Table 3: Impact of Self-Refinement RL on reasoning depth. We compare the average number of self-refinement rounds and the response length per round for models before and after RL training.

Model & Strategy	Avg. Refine Rounds	Avg. Response Len.
Qwen3-4B	10.38	10,297.97
RefineRL-4B (Ours)	9.78	11,371.62
Qwen3-4B-2507	9.26	10,330.52
RefineRL-4B-2507 (Ours)	8.66	17,558.65

Degradation analysis of Skeptical-Agent. We examine whether skepticism-driven refinement may hurt correct solutions. Table 4 reports two degradation metrics on LiveCodeBench-v6: *round-level degradation rate*, measuring how often a correct solution becomes incorrect after one refinement round, and *problem-level degradation rate*, measuring the fraction of problems degraded by over-refinement. We find that degradation is rare for both models, with round-level rates below 1.2% and problem-level rates below 2.3%. This suggests that over-refinement is not a major practical issue for Skeptical-Agent.

Table 4: Degradation rates of Skeptical-Agent (Skeptical@16) on LCB-v6.

Model (w/ Skeptical@16)	(%) / Round	(%) / Problem
RefineRL-4B	1.13	1.86
RefineRL-4B-2507	0.53	2.29

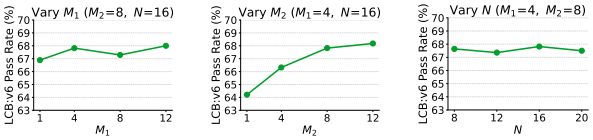
Effect of RL training on model responses. We analyze the behavioral changes induced by Self-Refinement RL in Table 3. After RL training, the average number of self-refinement rounds slightly decreases, while the average response length increases substantially. This suggests that our objective does not simply encourage the model to "try more times", but promotes deeper reasoning within each attempt. By generating longer and more detailed CoT, the model effectively trades the quantity of attempts for reasoning quality. Crucially, this substantial increase in reasoning length for RefineRL-4B-2507 correlates with the superior performance gains reported in our main results, reinforcing the hypothesis that extended, comprehensive CoT is instrumental in solving complex CP problems.

5.3 Ablation Studies

Hyperparameters of Skeptical-Agent. We study how the three key hyperparameters of Skeptical-Agent, *i.e.*, M_1, M_2, and N, affect its performance. Figure 3 reports the results on RefineRL-4B-

Table 5: Performance of Qwen3-4B-2507 on trained with and without self-refinement trajectory data.

Model & Strategy	LCB-v5	LCB-v6
Qwen3-4B-2507	62.65	55.21
+ Standard RL	66.95	57.96
+ Self-Refinement RL (Ours)	**70.40**	**61.57**
Qwen3-4B-2507 + Skeptical@16	71.29	62.07
+ Standard RL	72.90	63.79
+ Self-Refinement RL (Ours)	**75.97**	**67.82**



(a) Results of M_1. (b) Results of M_2. (c) Results of N.

Figure 3: Impact of (N, M_1, M_2) of Skeptical-Agent on RefineRL-4B-2507 and LCB-v6. Notably, $M_2 = 1$ disables skepticism-driven refinement.

Table 6: Performance of models trained via Skeptical-RL with different reward designs.

Qwen3-4B-2507 Self-Refinement RL	No-strategy		Skeptical@16	
	LCB-v5	LCB-v6	LCB-v5	LCB-v6
+ Pass-or-fail Reward	64.22	55.36	70.55	61.43
+ Linear Reward	68.23	59.86	74.03	65.07
+ Squared-incentive Reward	**70.40**	**61.57**	**75.97**	**67.82**

Table 7: Ablation on model scale. RefineRL-8B is trained with the same setting as RefineRL-4B.

Model	LCB-v5	LCB-v6
Qwen3-8B	55.39	49.32
RefineRL-8B	57.67	51.79
RefineRL-8B + Skeptical@16	**66.02**	**59.39**

2507 and LCB-v6. We find that Skeptical-Agent is relatively stable with respect to M_1 and N. In contrast, it is more sensitive to M_2: increasing M_2 from 1 to 8 substantially improves the pass rate (see Figure 2b). Recall that M_2 controls the degree of skepticism-driven refinement, and $M_2 = 1$ disables it. This confirms that enforcing rigorous self-refinement is a key step of Skeptical-Agent.

Effect of self-refinement trajectory data. We then analyze whether the gains of Self-Refinement RL mainly come from self-generated refinement trajectories. As a baseline, we train "Standard RL" models only on the original 4.5K HardTest CP problems, using the same squared-incentive reward for a fair comparison. Table 5 shows that models trained with self-refinement trajectories consistently outperform Standard RL baselines, both under Skeptical-Agent and in the single-attempt Pass@1 setting. This suggests that refinement trajectories help the model internalize debugging behavior, leading to stronger reasoning than training on problem statements alone.

Effect of reward functions. We next study the efficacy of our squared-incentive reward by comparing it with two alternatives: a binary pass-or-fail reward and a linear reward (see Appendix A.3 for reward definitions), while keeping the self-refinement trajectory data fixed. As shown in Table 6, the binary reward yields only weak improvements, while the linear reward performs better with denser feedback. However, both are consistently worse than our squared-incentive reward across all settings. This suggests that dense feedback alone is insufficient: by down-weighting easy gains on early test cases and amplifying rewards for later ones, the squared-incentive reward better discourages shallow test-case hacking and provides stronger guidance for complex CP tasks.

Ablation on model scale. We further examine whether our method remains effective beyond 4B models. To this end, we train RefineRL-8B from Qwen3-8B using the same pipeline and hyperparameters as RefineRL-4B. As shown in Table 7, RefineRL-8B + Skeptical@16 improves Qwen3-8B from 55.39% to 66.02% on LCB-v5 and from 49.32% to 59.39% on LCB-v6. These consistent gains show that the benefits of Self-Refinement RL and Skeptical-Agent are not limited to 4B models, but can also scale to stronger 8B backbones.

6 Conclusion

This paper presents RefineRL, an approach to incentivize LLM self-refinement in competitive programming. To mitigate unreliable public tests, we introduce *Skeptical-Agent*, which maintains skepticism toward generated solutions and enforces refinement even when local evaluations suggest correctness. We further propose *Self-Refinement RL*, which internalizes this "doubt and debug" behavior through self-generated refinement trajectories and a squared-incentive reward. RefineRL enables 4B models to outperform 32B baselines and approach 235B models on CP benchmarks, while also improving math performance despite training only on CP data. These results suggest self-refinement as a promising, data-efficient path toward stronger reasoning beyond parameter scaling.

AI USE STATEMENT

In this work, we used generative AI tools for implement methods and manuscript polishing. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. OpenCodeReasoning: Advancing data distillation for competitive coding. In *Second Conference on Language Modeling*, 2025.

Jiefeng Chen, Jie Ren, Xinyun Chen, Chengrun Yang, Ruoxi Sun, Jinsung Yoon, and Sercan O Arik. SETS: Leveraging self-verification and self-correction for improved test-time scaling. *Transactions on Machine Learning Research*, 2025a. ISSN 2835-8856.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025b.

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

Yuri Chervonyi, Trieu H. Trinh, Miroslav Olšák, Xiaomeng Yang, Hoang H. Nguyen, Marcelo Menegali, Junehyuk Jung, Junsu Kim, Vikas Verma, Quoc V. Le, and Thang Luong. Gold-medalist performance in solving olympiad geometry with AlphaGeometry2. *Journal of Machine Learning Research*, 26(241):1–39, 2025.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, et al. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*, 2025.

Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. Klear-CodeTest: Scalable test case generation for code reinforcement learning. *arXiv preprint arXiv:2508.05710*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. Hardtests: Synthesizing high-quality test cases for llm coding. *arXiv preprint arXiv:2505.24098*, 2025.

Hanxu Hu, Xingxing Zhang, Jannis Vamvas, Rico Sennrich, and Furu Wei. QueST: Incentivizing llms to generate difficult problems. *arXiv preprint arXiv:2510.17715*, 2025.

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

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

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. In *The Thirteenth International Conference on Learning Representations*, 2025.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 15964–15978, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7.

Qingyao Li, Xinyi Dai, Weiwen Liu, Xiangyang Li, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. ATGen: Adversarial reinforcement learning for test case generation. In *The Fourteenth International Conference on Learning Representations*, 2026.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-Prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *arXiv preprint arXiv:2508.03613*, 2025.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Zeyao Ma, Jing Zhang, Xiaokang Zhang, Jiaxi Yang, Zongmeng Zhang, Jiajun Zhang, Yuheng Jing, Lei Zhang, Hao Zheng, Wenting Zhao, Junyang Lin, and Binyuan Hui. Scaling agentic verifier for competitive coding. In *Forty-third International Conference on Machine Learning*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Mehrzad Samadi, Aleksander Ficek, Sean Narenthiran, Siddhartha Jain, Wasi Uddin Ahmad, Somshubra Majumdar, Vahid Noroozi, and Boris Ginsburg. Scaling test-time compute to achieve IOI gold medal with open-weight models. *arXiv preprint arXiv:2510.14232*, 2025.

Shijie Shang, Ruosi Wan, Yue Peng, Yutong Wu, Xiong-hui Chen, Jie Yan, and Xiangyu Zhang. StepFun-Prover preview: Let's think and verify step by step. *arXiv preprint arXiv:2507.20199*, 2025.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Ben Shi, Michael Tang, Karthik R Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? In *First Conference on Language Modeling*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Xiaoshuai Song, Yanan Wu, Weixun Wang, Jiaheng Liu, Wenbo Su, and Bo Zheng. ProgCo: Program helps self-correction of large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 944–959, Vienna, Austria, 2025. Association for Computational Linguistics. ISBN 979-8-89176-252-7.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. CodeContests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Jie Wu, Haoling Li, Xin Zhang, Jiani Guo, Jane Luo, Steven Liu, Yangyu Huang, Ruihang Chu, Scarlett Li, and Yujiu Yang. X-Coder: Advancing competitive programming with fully synthetic tasks, solutions, and tests. *arXiv preprint arXiv:2601.06953*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhaojian Yu, Yinghao Wu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. Z1: Efficient test-time scaling with code. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 2688–2712. Association for Computational Linguistics, 2025. ISBN 979-8-89176-333-3.

Qianfan Zhang, Tianyu Guo, Xuandi Ren, Jiale Chen, Ming Ding, Ran Xin, and Xia Xiao. Scaling reasoning tokens via rl and parallel thinking: Evidence from competitive programming. *arXiv preprint arXiv:2604.01302*, 2026.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

A Additional Implementation Details

This section provides additional implementation details omitted from the main text.

A.1 Algorithm of Skeptical-Agent

We provide the complete procedure of Skeptical-Agent (see Section 4.2) in Algorithm 1.

Algorithm 1 Skeptical-Agent for CP Problem Solving

Input: Reasoning LLM π_θ, problem statement x, public test cases set T^{pub}, hyperparameters (N, M_1, M_2), oracle execution and judging functions f and g.

Output: Optimal solution text $r_{\text{sol}}^{(*)}$ and code $r_{\text{code}}^{(*)}$.

$c_1 \leftarrow 0,\ c_2 \leftarrow 0$ ▷ Initialize temporary counters
$r_{\text{sol}}^{(*)}, r_{\text{code}}^{(*)}, r_{\text{fed}}^{(0)} \leftarrow$ Null ▷ Initialize as empty value
for n **in** $1, \cdots, N$ **do**
  $r_{\text{cot}}^{(n)}, r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)} \sim \pi_\theta(x, r_{\text{fed}}^{(n-1)})$ ▷ Eq. (3)
  $(r_{\text{sol}}^{(*)}, r_{\text{code}}^{(*)}) \leftarrow (r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)})$ ▷ Update final solution
  $i_*, o_*, \hat{o}_*, j_* \leftarrow$ Null ▷ Store judging result
  for $(i_k^{\text{pub}}, o_k^{\text{pub}})$ **in** T^{pub} **do**
    $\hat{o}_k^{(n)} \leftarrow f(r_{\text{code}}^{(n)}, i_k^{\text{pub}}),\ j_k^{(n)} \leftarrow g(\hat{o}_k^{(n)}, o_k^{\text{pub}})$
    if $j_k^{(n)} \neq$ AC **then**
      $(i_*, o_*, \hat{o}_*, j_*) \leftarrow (i_k^{\text{pub}}, o_k^{\text{pub}}, \hat{o}_k^{(n)}, j_k^{(n)})$
      break
    end if
  end for
  if $j_* \neq$ Null **then**
    $c_1 \leftarrow ((c_1 + 1) \mod M_1),\ c_2 \leftarrow 0$
  else
    $c_1 \leftarrow 0,\ c_2 \leftarrow (c_2 + 1)$
  end if
  if $c_2 = M_2$ **then**
    break
  end if
  if $c_1 > 0$ **then**
    $r_{\text{fed}}^{(n)} \leftarrow \mathcal{P}_{\text{Err}}(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)}, i_*, o_*, \hat{o}_*, j_*)$ ▷ Eq. (4)
  else if $c_2 > 0$ **then**
    $r_{\text{fed}}^{(n)} \leftarrow \mathcal{P}_{\text{Pass}}(r_{\text{sol}}^{(n)}, r_{\text{code}}^{(n)})$ ▷ Eq. (5)
  else
    $r_{\text{fed}}^{(n)} \leftarrow$ Null
  end if
end for
return $(r_{\text{sol}}^{(*)}, r_{\text{code}}^{(*)})$

A.2 Training Data for Self-Refinement RL

We construct the training dataset for Skeptical RL based on a subset of the HardTest dataset (He et al., 2025), consisting of CP problems that are **suitable for learning self-refinement**. Formally, a problem is considered to be *suitable* if it contains at least 8 test cases with both input and output lengths not exceeding 512 tokens. The idea behind this is that short input-output pairs are essential for both humans and AI models to effectively diagnose errors and facilitate debugging.

To form the base CP problem dataset D_{CP}, we select suitable problems from the first 1/8 of HardTest. For each selected problem, we retain all test cases shorter than 512 tokens and randomly sample up to 8 additional test cases from the remainder. This process results in a base set D_{CP} consisting of approximately 4.5K CP problems.

Finally, we employ the pipeline described in Section 4.3 to synthesize the final Self-Refinement RL training data. We focus on generating Python code, and set the maximum generation length to 32,768 for Qwen3-4B and 65,536 for Qwen3-4B-2507. For each model, this process generates approximately 4.2K self-refinement trajectories. Combining these with the base problems yields a total of roughly 8.7K training samples per model. Detailed statistics of the final training datasets are presented in Table 8.

Table 8: Statistics of the training dataset for Self-Refinement RL. This table details the sample counts for each interaction category. "WA/RE/TLE/CE/MLE" represent trajectories triggered by error-driven feedback. "AC → WA/RE/TLE/CE/MLE" denote trajectories involving skepticism-driven feedback, where the solution passed public test cases but failed on private ones due to the indicated errors.

Data Type	**Qwen3-4B**	**Qwen3-4B-2507**
Initial CP Problems from D_{CP}	4,500	4,500
Error-driven Trajectory	2,811	2,721
[True Negative] Wrong Answer (WA)	2,127	2,077
[True Negative] Runtime Error (RE)	377	364
[True Negative] Time Limit Exceeded (TLE)	307	280
[True Negative] Compile Error (CE)	0	0
[True Negative] Memory Limit Exceeded (MLE)	0	0
Skepticism-driven Trajectory	1,449	1,471
[True Positive] Accepted (AC)	309	370
[False Positive] AC → WA	863	856
[False Positive] AC → RE	148	139
[False Positive] AC → TLE	129	106
[False Positive] AC → CE	0	0
[False Positive] AC → MLE	0	0
Total Number	**8,760**	**8,692**

A.3 Reward Function Details

This section provides the definitions of the reward functions used in our ablation study in Section 5.3. Given a CP problem (x, T) and a generated solution code r_{code}, let k^* denote either $|T|$ if r_{code} passes all test cases, or one less than the index of the first test case in T that r_{code} fails to pass. We compare the following three reward functions.

Pass-or-fail reward. This is the standard sparse reward commonly used in code RL:

$$R_{\text{pass-or-fail}}(r_{\text{code}}, T) := 1\left[k^* = |T|\right],$$

where $1[\cdot]$ is an indicator function. It gives reward 1 only when the solution passes all test cases, and 0 otherwise.

Linear reward. The linear reward gives partial credit according to the fraction of test cases passed before the first failure:

$$R_{\text{linear}}(r_{\text{code}}, T) := k^*/|T|.$$

Compared with the pass-or-fail reward, it provides denser feedback for partially correct solutions.

Squared-incentive reward (ours). Our proposed reward squares the fraction of passed test cases:

$$R(r_{\text{code}}, T) := \left(k^*/|T|\right)^2.$$

This design down-weights easy gains from early test cases while amplifying the reward for passing later ones, encouraging the model to solve the problem more completely.

A.4 RL Training Details

We use the VeRL framework [2] (Sheng et al., 2024) to perform RL training via the GRPO (Guo et al., 2025) algorithm, where the KL-regularization term is excluded. both the batch size and mini-

[2] https://github.com/volcengine/verl

batch size are set to 32 and For the rollout hyperparameters of GRPO, we set the rollout number to 8, the maximum response length to 32,768 for Qwen3-4B and 65,536 for Qwen3-4B-2507, the temperature to 1.0, and the top-p to 1.0. AdamW is adopted as the optimizer, where the learning rate is fixed to 10^{-6} throughout the RL training. The two RL trained models from Qwen3-4B and Qwen3-4B-2507 are denoted as **RefineRL-4B** and **RefineRL-4B-2507**, respectively. Each RL training experiment is conducted on 32 NVIDIA A100@80G GPUs.

Calculating rewards for CP data requires executing LLM-generated code against multiple private test cases, which is a CPU-intensive process. To enable efficient and scalable reward computation during RL training, we integrate our evaluation logic with the VeRL framework, which is built upon the distributed computing engine Ray [3]. We leverage Ray to schedule distributed code evaluations, assigning isolated CPU cores for individual execution tasks. To strictly manage resource consumption during training, we employ `timeout` [4] and `prlimit` [5] to enforce time and memory constraints, set to 8 seconds and 512MB, respectively. Finally, to ensure reasoning integrity, we enforce complete CoT generation: only responses containing the CoT closing delimiter (*i.e.*, the `</think>` tag) are submitted for reward calculation; otherwise, a reward of 0 is assigned.

A.5 Evaluation Details

Existing evaluations on the LiveCodeBench benchmark typically rely on the official LCB evaluation toolkit [6] or the Hugging Face LightEval package [7]. However, we argue that these packages are inefficient for large-scale CP solution judging, as they lack native support for high-throughput parallel inference and fail to fully leverage multi-core CPUs for computationally intensive solution code execution. To address these limitations, we implemented a custom, scalable toolkit for efficient CP performance evaluation.

Our implementation decouples the evaluation pipeline into two distinct stages to ensure robustness and efficiency:

- **Step 1: LLM solution generation.** We utilize the SGLang inference engine [8] to perform high-throughput parallel LLM generation. We adopt the standard LiveCodeBench solution generation prompts from the LightEval package for LLMs to generate Python solutions, with the exception of our Skeptical-Agent, which incorporates the additional feedback prompts detailed in Appendix C to perform self-refinement.
- **Step 2: Concurrent code evaluation.** Consistent with the training setup described in Appendix A.4, we employ the Unix CLI tools `timeout` and `prlimit` to enforce strict time and memory constraints. To maximize resource utilization, we leverage native multiprocessing for the Python's `asyncio` [9] package to evaluate multiple solutions concurrently.

Regarding evaluation settings, for every CP problem, we set a time limit of 6s and a memory limit of 512MB. All code executions are performed on a 2.5GHz multi-core CPU. To ensure a fair comparison, all models in our experiments are evaluated using our custom toolkit, with results averaged over 16 runs to ensure stability. The LLM sampling parameters are set to a temperature of 0.6 and a top-p of 0.95. The maximum generation length is set to 32,768 tokens for Qwen3-4B and our RefineRL-4B, and 65,536 tokens for all other models. Furthermore, given that the evaluated models are reasoning-oriented, we strictly enforce complete Chain-of-Thought (CoT) integrity; specifically, only responses containing a detected CoT completion tag are submitted for evaluation; otherwise, they are assigned a score of zero.

Additionally, for both CP and math evaluations, each evaluation was conducted on a single node with 8 NVIDIA A100 GPUs.

[3] https://github.com/ray-project/ray
[4] https://man.archlinux.org/man/timeout.1
[5] https://man.archlinux.org/man/prlimit.1
[6] https://github.com/LiveCodeBench/LiveCodeBench
[7] https://github.com/huggingface/lighteval
[8] https://github.com/sgl-project/sglang
[9] https://docs.python.org/3/library/asyncio.html

B ADDITIONAL EXPERIMENTAL RESULTS

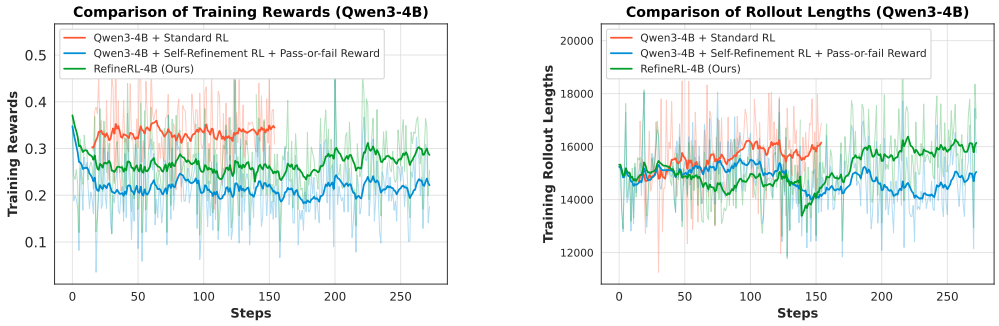
This section presents additional experimental results omitted from the main text.

B.1 TRAINING DYNAMICS

We provide a detailed visualization of the training dynamics for our Skeptical RL models and other RL baselines. Table 9 summarizes the configurations for each strategy, detailing the dataset size and reward functions used for the Qwen3-4B and Qwen3-4B-2507 base models.

Table 9: Comparison of RL Training Strategies. We contrast three RL setups: Vanilla RL (direct generation), Self-Refinement RL with pass-or-fail rewards, and our Self-Refinement RL (with squared-incentive rewards). Vanilla RL uses only the initial generation subset (∼4.5k), while refinement methods use the full trajectory dataset (∼8.7k). Our method employs a squared reward to incentivize robust self-verification.

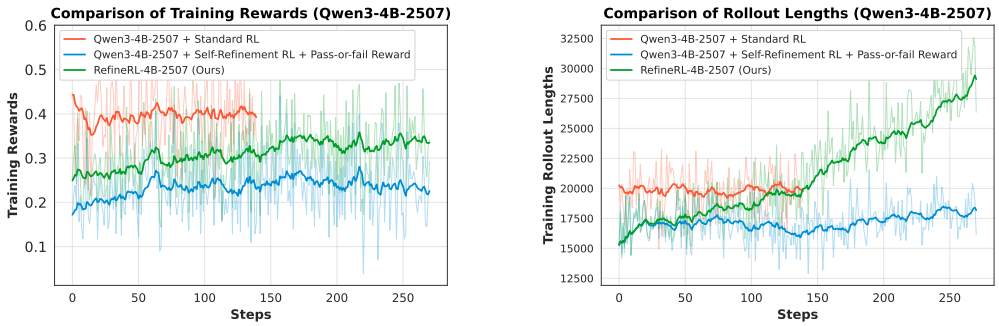
Method	Data Size	Trajectory Type	Reward Function
Vanilla RL	∼4.5K	None	Squared-incentive
Refinement (Binary)	∼8.7K (4.5K Real Data + 4.2K Syn. Data)	Refinement	Pass-or-fail
RefineRL (Ours)	∼8.7K (4.5K Real Data + 4.2K Syn. Data)	Refinement	Squared-incentive



(a) Average reward per step.

(b) Average rollout length (tokens) per step.

Figure 4: Training dynamics of Qwen3-4B. The left plot shows the steady increase in reward, while the right plot shows the corresponding growth in response length, reflecting the model's internalization of the skeptical reasoning process.



(a) Average reward per step.

(b) Average rollout length (tokens) per step.

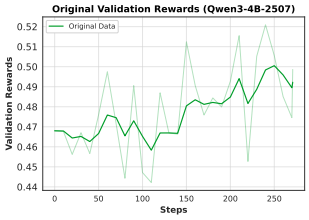
Figure 5: Training dynamics of Qwen3-4B-2507. Similar to the base model, the 2507 variant demonstrates consistent learning progress in terms of reward maximization and increased reasoning length during the RL process.

Figures 4 and 5 illustrate the evolution of the average reward and the average rollout token length during the training process for **Qwen3-4B** and **Qwen3-4B-2507**, respectively.

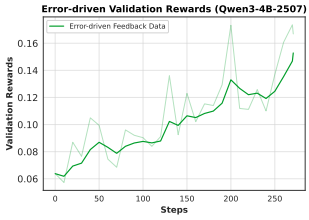
Across both models, we observe a consistent upward trend in the reward curves of our Self-Refinement RL (*i.e.*, RefineRL-4B and RefineRL-4B-2507), indicating that the models are effectively learning to solve increasingly complex competitive programming problems and adapting to the squared-incentive reward signal. Simultaneously, the rollout token lengths of our Self-Refinement RL generally exhibit an increasing trend. This suggests that as the models improve, they learn to generate more comprehensive solution analyses and engage in more extensive self-refinement loops to debug and verify their code before submission, rather than attempting to guess short, likely incorrect solutions.

B.2 Validation Dynamics

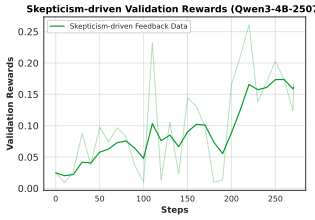
We further examine the validation dynamics of Self-Refinement RL to understand whether the model continues to improve during training or merely overfits to the constructed refinement trajectories. Specifically, we track validation rewards on three held-out settings: original CP problems, error-driven refinement trajectories, and skepticism-driven refinement trajectories. As shown in Figure 6 and Figure 7, the validation rewards generally increase across all three settings for both Qwen3-4B-2507 and Qwen3-4B. This suggests that RL training improves the model's general problem-solving and self-refinement capabilities, rather than only memorizing the training trajectories.



(a) Validation reward per step on the original held-out problems.

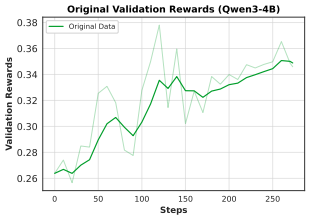


(b) Validation reward on held-out error-driven refinement data.

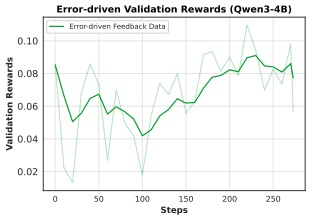


(c) Validation reward on held-out skepticism-driven refinement data.

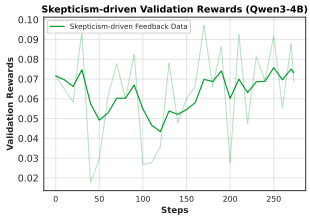
Figure 6: Validation reward curves across three held-out validation settings of Qwen3-4B-2507 throughout RL training. In each subplot, the solid line shows the smoothed trend, while the lighter transparent line shows the raw validation reward. These results indicate that the model continues to acquire self-refinement ability during later training stages, rather than merely overfitting to the training trajectories.



(a) Validation reward per step on the original held-out problems.



(b) Validation reward on held-out error-driven refinement data.



(c) Validation reward on held-out skepticism-driven refinement data.

Figure 7: Validation reward curves across three held-out validation settings of Qwen3-4B throughout RL training. In each subplot, the solid line shows the smoothed trend, while the lighter transparent line shows the raw validation reward. These results indicate that the model continues to acquire self-refinement ability during later training stages, rather than merely overfitting to the training trajectories.

B.3 Inference-time Strategies on RefineRL-4B

We report the performance (Table 10) and time cost (Figure 8) of different inference-time strategies on our RefineRL-4B model, following the similar analysis in Section 5.2 (see Table 2 and Figure 2). We observe trends similar to those on RefineRL-4B-2507: Skeptical@16 achieves the best overall performance, maintains a speed comparable to those of other sequential baselines, and is substantially faster than parallel-sampling baselines such as Random@16 and LongCoT@16.

Table 10: Performance of inference-time strategies on RefineRL-4B. Our Skeptical@16 strategy outperforms all compared baselines across all settings.

Model & Strategy	LCB-v5	LCB-v6	USACO
RefineRL-4B (Ours)	56.40	48.79	44.75
+ Random@16	62.95	52.68	47.72
+ LongCoT@16	60.14	50.32	34.93
+ RejSamp@16	61.49	51.75	47.72
+ Reflexion@16	62.39	51.86	47.94
+ Skeptical@16 (Ours)	**64.07**	**56.54**	**50.61**

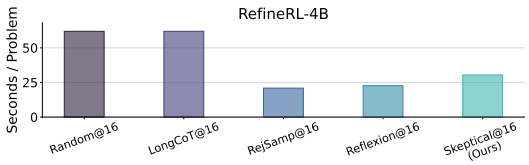


Figure 8: Time cost of inference-time strategies on RefineRL-4B. Skeptical@16 achieves a speed comparable to other sequential-sampling baselines while being faster than parallel-sampling baselines.

B.4 Case Study on IOI 2024

We further evaluate RefineRL on the International Olympiad in Informatics (IOI) 2024 problems as a challenging case study. Compared with standard CP benchmarks, IOI problems are substantially more difficult and adopt a subtask-based scoring protocol with multiple submissions allowed for each problem. We follow the official IOI scoring protocol and report **Score@50**, where the score of each subtask is maximized over 50 submissions and then aggregated across all problems. The maximum total score of IOI 2024 is 600.

For Skeptical-Agent, we use the stronger Skeptical@32 setting with $N = 32$, $M_1 = 2$, and $M_2 = 32$ to accommodate the increased difficulty of IOI problems. Table 11 summarizes the results.

Model & Strategy	**IOI 2024 Score@50**
Qwen3-235B-A22B-2507	148.5
OCR-Nemotron-1.1-32B	132.0
Qwen3-4B-Thinking-2507	90.0
+ Skeptical@32 (Ours)	109.0
RefineRL-4B-2507 (Ours)	109.0
+ Skeptical@32 (Ours)	**117.0**

Table 11: Performance on IOI 2024 under the official subtask-level scoring protocol with up to 50 submissions per problem.

Self-Refinement RL and Skeptical-Agent independently improve the 4B model from 90.0 to 109.0 points. Their combination further reaches 117.0, giving a 27.0-point gain over the original model. This complementary improvement suggests that RL training and inference-time skeptical refinement contribute distinct benefits even on highly challenging IOI problems.

C PROMPTS

In this section, we provide the detailed prompt templates used in our Skeptical-Agent.

C.1 PROMPTS FOR COMPETITIVE PROGRAMMING

Figure 9 presents the base system prompt used for initializing the problem-solving process and generating the initial solution. When the agent enters the refinement loop, the feedback it receives depends on the outcome of the local testing. If the solution fails the local test cases, the agent receives a detailed error report constructed using the template in Figure 10, which incorporates specific failure reasons (*e.g.*, Wrong Answer, Runtime Error) as detailed in Figure 11. Conversely, if the solution passes the local tests, our Skeptical-Agent does not immediately accept it. Instead, it triggers a "skeptical" review using the template in Figure 12, forcing the model to self-verify against potential edge cases, hard-coded cheats, or complexity issues before final submission.

Prompt Template for Direct Generation (Python)

You are a competitive programmer. You will be given a question (problem statement) and will generate a correct python program that matches the specification and passes all tests.
Please first give a solution analysis and then provide the python solution code.
Ensure that when your solution program runs, it reads the input from stdin and writes the output (answer) to stdout. Enclose your code within delimiters as follows:

```
```python
# YOUR CODE HERE
```
```

Question: {problem statement}

Additional Messages
{prompt for feedback}

Figure 9: Standard system prompt used for initial solution generation and subsequent refinement steps.

Prompt Template for Error-driven Feedback Function $\mathcal{P}_{\text{Err}}$

In the previous round, a solution has been proposed but failed to pass the local testing process. Below provides detailed **Solution Analysis**, **Solution Code**, and **Failure Reason** for this failed solution. Please consider leveraging all these messages to debug the old solution and further design a better solution.

```
## Solution Analysis
```text
{analysis}
```

Solution Code
```python
code
```

Failure Reason
{error reason}
```

Figure 10: Prompt for error-driven feedback construction.

Error Reasons for Different Evaluation Results

Compile Error (CE)

The proposed solution code received **Compile Error** during local testing.

Wrong Answer (WA)

The proposed solution code received **Wrong Answer Error** during local testing. The error occurred when the proposed code was running on the following test input:
```text
{test input}
```
The expected correct test output is:
```text
{test output}
```
However, the output of the proposed solution for this test case is:
```text
{sol test output}
```
which is different from the correct output.

Please DO NOT hard-code this test input/output to your solution code. This is cheating and would not help you pass the final stress testing evaluation.

Runtime Error (RE)

The proposed solution code received **Runtime Error** during local testing. The error occurred when the proposed code was running on the following test input:
```text
{test input}
```
Please pay attention to the details of your code.

Time Limited Exceed (TLE)

The proposed solution code received **Time Limited Exceed Error** during local testing. The error occurred when the proposed code was running on the following test input:
```text
{test input}
```
Please be aware of the time limit of the problem when solving it.

Memory Limited Exceed (MLE)

The proposed solution code received **Memory Limited Exceed Error** during local testing. The error occurred when the proposed code was running on the following test input:
```text
{test input}
```
Please be aware of the memory limit of the problem when solving it.

Figure 11: Detailed templates for constructing the "Error Reason" field based on specific execution errors (WA, TLE, RE, etc.).

Prompt Template for Skepticism-driven Feedback Function $\mathcal{P}_{\text{Skep}}$

In the previous round, a solution has been proposed and successfully passed all local test cases. Below provides detailed **Solution Analysis** and **Solution Code** for this solution and an additional **Check List**. Please analyze the proposed solution **step-by-step** following the provided **Check List**.
If the previously proposed solution can pass all conditions in the **Check List**, please explain why and output the solution code. If NOT, please also explain why and leverage all these messages to debug the old solution and further design a better solution to pass the **Check List**.

Solution Analysis
```text
{analysis}
```

Solution Code
```python
{code}
```

##Check List
- Ensure the solution code does NOT hard-code answers for some specific testcases and directly output them to cheat the judging system.
- Ensure the **CORRECTNESS** of the proposed solution by carefully thinking if there are any edge-testcases that can cause the proposed solution code encounter **Wrong Answer Error**.
- Ensure the time and space complexities of the proposed solution are **OPTIMAL** by carefully thinking if there are any edge-testcases that can cause the proposed solution code encounter **Time Limit Exceed Error** or **Memory Limit Exceed Error**.

Figure 12: The skeptical-driven feedback prompt ($\mathcal{P}_{\text{Skep}}$) triggered when a solution passes local tests (on all public test cases), instructing the agent to self-critique against a checklist.

C.2 Prompts for Mathematical Reasoning

Figure 13 presents the base prompt used for initial solution generation and subsequent refinement on AIME problems. Unlike CP tasks, mathematical problems like AIME problems do not provide executable test cases, so we cannot construct error-driven feedback from local testing results. Therefore, for mathematical reasoning, we enforce refinement solely through skepticism-driven feedback. Specifically, after each proposed solution, the agent is prompted to re-check the previous reasoning step by step using the template in Figure 14, and either justify the existing answer or identify errors and propose a corrected solution.

Prompt Template for Direct Generation (AIME)

Solve the following math problem efficiently and clearly. The last line of your response should be of the following format: 'Therefore, the final answer is: $\boxed{ANSWER}$. I hope it is correct' (without quotes) where ANSWER is just the final number or expression that solves the problem. Think step by step before answering.

{problem statement}

Additional Messages
{prompt for feedback}

Figure 13: Standard system prompt used for initial solution generation and subsequent refinement steps in solving AIME problems.

Prompt Template for Skepticism-driven Feedback Function $\mathcal{P}_{\text{Skep}}$ for AIME

In the previous round, a solution and its corresponding answer has been proposed. Please analyze the proposed solution **step-by-step** to analyze whether it is indeed correct or not.
If you believe the previously proposed solution is correct, please explain why and output the previous answer. If NOT, please also explain why and leverage all these messages to debug the old solution and propose a better solution.

Previous Solution
```text
{previous solution}
```

Figure 14: The skeptical-driven feedback prompt ($\mathcal{P}_{\text{Skep}}$) triggered every time when the agent tries to perform refinement.