TIPCODER: Reinforcement Learning Boosted Test-time Instruction Proposer for Code Generation

Anonymous ACL submission

Abstract

Test-time scaling for code generation typically explores the solution space by sampling multiple programs from a fixed instruction. We study a complementary direction: instance-level instruction-space exploration. Our observation is that many coding failures stem from missing constraints, overlooked edge cases, or misleading reasoning paths induced by the original prompt. To address this, we propose TIPCODER, a test-time instruction proposer that generates problem-specific auxiliary tips before code synthesis. TIPCODER distills multi-turn debugging trajectories into proactive guidance and further optimizes the Proposer with reinforcement learning using a marginal-utility reward. At inference time, it generates both a base solution and a tip-guided solution, and applies a Reward Model for post-hoc selection. This exploration-selection design allows tips to expose additional candidate potential while reducing regressions from unnecessary guidance. Across multiple code-generation benchmarks and target Code LLMs, TIPCODER provides a robust instruction-level test-time scaling strategy, comparing favorably with stochastic sampling and generic prompt optimization baselines under a shared reward-model-based selection protocol. [1]

1 Introduction

Code Large Language Models (Guo et al., 2024; Hui et al., 2024) have become increasingly effective at translating natural-language specifications into executable programs. However, their outputs remain highly sensitive to the initial problem instruction, which may omit implicit constraints, under-specify boundary cases, or induce a spurious reasoning trajectory. Existing test-time scaling methods, such as Best-of-N sampling (Brown et al., 2024), improve reliability by allocating additional computation to explore the *solution space* (Snell et al., 2025; Muennighoff et al., 2025). Since these methods typically sample from a fixed conditioning context, they may remain concentrated around the same flawed interpretation when the original instruction is misleading. This motivates a complementary scaling axis: adapting the instruction that guides solution search.

Prompt optimization demonstrates that instructions can be optimized to improve model behavior (Zhou et al., 2022; Yang et al., 2023; Agrawal et al.; Hu et al., 2025). For code generation, however, the limiting factor is often not the absence of a globally better prompt, but the absence of instance-specific guidance. A failure may hinge on a particular boundary case, index convention, library behavior, numerical constraint, or ambiguity in the specification. Such localized failure modes are poorly captured by fixed dataset-level prompts or generic reminders such as "consider edge cases". We therefore study *instance-level instruction proposing*: generating a problem-specific auxiliary instruction before code synthesis.

We introduce TIPCODER, a test-time instruction proposer for code generation. Given a programming problem, a `Proposer` generates an auxiliary tip that highlights relevant constraints, likely pitfalls, or implementation considerations. The target Code LLM then produces two candidates: a base solution conditioned on the original instruction and a tip-guided solution conditioned on the augmented instruction. The tip is not intended to specify the solution; instead, it modifies the conditioning context so that the generator explores an alternative region of the solution space.

To learn such tips, TIPCODER distills multi-turn debugging trajectories into proactive guidance. In these trajectories, an initial solution fails, feedback exposes a missing requirement or incorrect assumption, and a subsequent solution repairs the error. This process reveals information that would have been useful before the initial generation. We

[1] We will release our code and data upon publication.

first initialize the Proposer with Supervised Fine-Tuning on distilled repair guidance, and then optimize it with RL in a frozen Code LLM environment. The reward measures the marginal utility of the generated tip relative to base generation, encouraging corrective instructions while penalizing harmful or redundant perturbations.

At inference time, the utility of a tip is instance-dependent. A tip-guided candidate may recover a solution that is unlikely under the original prompt, but can also perturb an already correct base trajectory. Thus, the base and tip-guided candidates define an oracle-selection upper bound for instruction-guided scaling, while practical deployment requires instance-wise selection. TIPCODER uses a Reward Model (RM) for this post-hoc selection. The RM does not create instruction-level exploration, but approximates the selector needed to realize the candidate potential exposed by the Proposer.

TIPCODER operates as an external prompt-level module and requires no access to the target Code LLM's parameters, gradients, or execution feedback during inference. Across multiple code-generation benchmarks and backbones, TIPCODER consistently improves pass rates over base generation and stochastic sampling. Under the same RM reranking protocol, it also outperforms self-hinting and generic context-optimization baselines, indicating that the gains arise from more effective instance-level instruction proposals rather than from the mere availability of an additional candidate selector.

Our contributions are summarized as follows:

- **Instance-level instruction-space exploration.** We formulate test-time instruction proposing as a complementary scaling axis for code generation, where problem-specific auxiliary instructions guide solution search beyond fixed-prompt stochastic sampling and dataset-level prompt optimization.

- **Debugging-distilled and utility-optimized Proposer.** We distill multi-turn debugging trajectories into proactive tips that capture latent constraints, edge cases, and failure modes. We then optimize the Proposer with RL using a marginal-utility reward.

- **Exploration-selection decomposition.** We use RM-based reranking to approximate instance-wise selection between base and tip-guided candidates, and evaluate TIPCODER against stochastic sampling, self-hinting, and context-optimization baselines under the same reranking protocol.

2 Methodology

2.1 Problem Formulation

Let $\mathcal{D} = \{(x, T)\}$ denote a dataset where x is the problem description and T is the unit test set. A Code LLM $\mathcal{M}$ generates a solution y according to $P(y|x) = \mathcal{M}(y|x)$. While standard one-off generation often fails due to ambiguous constraints, empirical evidence suggests that a correct solution y^* remains reachable (Austin et al., 2021; Larbi et al., 2024). This can be achieved through iterative debugging with execution feedback, where error messages are utilized to refine the code over multiple turns (Chen et al., 2023; Shinn et al., 2023).

Our goal is to achieve the reliability of this multi-turn process without its computational overhead. We posit that the sparse reasoning signals within the interaction history can be compressed into a concise latent instruction c (a "tip"). Consequently, we formulate the problem as learning a proposer policy $\pi_\phi(c|x)$ that maximizes the expected pass rate of the downstream generation:

$$\max_{\phi} \mathcal{J}(\phi) = \mathbb{E}_{c\sim\pi_\phi(\cdot|x)}\Big[\mathbb{E}_{y\sim\mathcal{M}(\cdot|x,c)} [\nVdash(\text{Pass}(y, T))]\Big].$$

Intuitively, this objective drives the proposer to discover optimal shortcuts in the instruction space that functionally replicate the utility of the expensive debugging history.

2.2 Tip Supervision from Debugging Trajectories

Training the Proposer requires high-quality pairs (x, c) where the auxiliary instruction c captures information that is missing from the original problem instruction but useful for code generation. We construct such supervision by distilling successful multi-turn debugging trajectories into single-turn proactive tips. The construction pipeline consists of three stages: mining repair trajectories, extracting debugging observations through masked distillation, and verifying the causal utility of the tips.

Mining Repair Trajectories. We use AceCoder-87K, a large-scale code instruction dataset constructed through evolutionary instruction generation from open-source repositories, as the seed corpus. To obtain dense reasoning signals, we focus

on challenging samples where the student model exhibits low one-pass performance, filtering for problems with Pass@1 $< 50\%$. For each retained problem, we employ Gemini-3-Pro as the teacher model to perform iterative debugging with execution feedback.

For a problem x with unit tests T, we denote a debugging trajectory as

$$\tau = \{(y^0, e^0), (y^1, e^1), \ldots, (y^K, e^K)\}, \quad (1)$$

where y^t is the solution generated at turn t and e^t is the corresponding execution feedback. We retain trajectories where the initial solution fails but is successfully repaired within three turns:

$$\mathrm{Pass}(y^0, T) = 0, \qquad \mathrm{Pass}(y^K, T) = 1. \quad (2)$$

Such trajectories capture the transition from an incorrect initial interpretation to a correct solution, and therefore expose the information that the original instruction failed to elicit.

Abductive Reasoning via Masked Distillation. A successful repair trajectory contains a *debugging observation*: the hidden constraint, edge case, or invalid assumption that explains why the initial solution failed and how the later solution repaired it. The goal of distillation is to convert this retrospective observation into a proactive instruction that can be provided before code generation. In this sense, the tip serves as a compact substitute for the foresight normally obtained only after costly iterative debugging.

To prevent solution leakage, we apply masked distillation. Implementation-level code blocks and solution-specific details are masked before the teacher extracts the observation. Formally, the teacher first infers the debugging observation from the problem and the masked trajectory:

$$o_\tau = \mathrm{Obs}_{\mathrm{teacher}}(x, \mathrm{Mask}(\tau)). \quad (3)$$

The observation is then verbalized as an auxiliary tip:

$$c \sim q_{\mathrm{teacher}}(\cdot \mid x, o_\tau). \quad (4)$$

The generated tip is required to describe high-level constraints, edge cases, or likely failure modes rather than prescribe the final implementation. This process distills the retrospective knowledge revealed by multi-turn repair into a single-turn guidance signal available at the initial generation stage.

Iterative Feedback Verification. The distilled tip is retained only if it is causally useful for the student model. We append the generated tip c to the original instruction x and run the student model again. If the generated code still fails, the execution feedback is returned to the teacher model to refine the tip, with at most two refinement rounds. A pair (x, c) is added to the final training set if and only if the verified tip guides the student model to pass all unit tests:

$$\mathrm{Pass}(y_{\mathrm{tip}}, T) = 1. \quad (5)$$

This filtering process yields 2,126 high-quality problem-tip pairs for supervised initialization.

2.3 Proposer Optimization

We optimize the Proposer in two stages. Supervised Fine-Tuning initializes the policy with verified debugging-derived tips, and Reinforcement Learning further aligns the policy with downstream code correctness.

Supervised Fine-Tuning (SFT). We initialize the Proposer policy π_ϕ using the distilled dataset $\mathcal{D}_{train}$. This stage teaches the Proposer the form and patterns of useful tips. However, it does not directly optimize downstream correctness, motivating the subsequent RL stage.

Latent Instruction Formulation. We formulate instruction generation as latent-variable inference. The tip c mediates the relationship between the problem x and the correct solution y^*. This view connects the supervised initialization and the downstream correctness objective: SFT provides a coherent prior over natural-language tips, while RL searches within this instruction space for tips that improve the target Code LLM.

We formulate the optimization goal as maximizing the marginal log likelihood $\log P(y^*|x)$. To make this tractable, we use the Proposer policy $\pi_\phi(\cdot|x)$ as the variational posterior and align the prior distribution with the frozen SFT reference model $\pi_{\mathrm{ref}}(\cdot|x)$ to preserve semantic stability. Applying Jensen's inequality in the discrete instruction space gives the Evidence Lower Bound:

$$\begin{aligned}
\log P(y^*|x) &= \log \sum_c \pi_\phi(c|x) \frac{P(y^*|x,c)\pi_{\mathrm{ref}}(c|x)}{\pi_\phi(c|x)} \\
&\geq \sum_c \pi_\phi(c|x) \log \frac{P(y^*|x,c)\pi_{\mathrm{ref}}(c|x)}{\pi_\phi(c|x)} \\
&= \underbrace{\mathbb{E}_{c\sim\pi_\phi}[\log P(y^*|x,c)]}_{\text{Reconstruction Objective}} - \underbrace{D_{KL}(\pi_\phi(\cdot|x)||\pi_{\mathrm{ref}}(\cdot|x))}_{\text{Regularization Constraint}}.
\end{aligned}$$

Here $P(y^*|x, c)$ denotes the likelihood of the fixed Code LLM generating the correct solution given the tip. The ELBO provides the bridge from SFT to RL. The reconstruction term corresponds to the downstream utility of the sampled tip: a useful instruction should increase the probability that the frozen generator produces a correct solution. The KL term regularizes the Proposer toward the SFT reference distribution, preventing the policy from drifting. Since $P(y^*|x, c)$ is not directly observable in code generation, we approximate this term with execution-based rewards in the RL stage.

Reinforcement Learning Objective. Since $P(y^*|x, c)$ is not directly accessible in code generation, we replace the reconstruction term with an execution-based reward. We define the instruction proposer $\pi_\phi(c|x)$ as the **policy** and the combination of the frozen Code LLM and the deterministic execution sandbox as the **environment**. The environment receives a tip c, generates code, and returns a non-differentiable reward based on execution results.

Guided by the ELBO formulation, our objective $\mathcal{J}(\phi)$ is to maximize the expected reward subject to a KL constraint:

$$\begin{aligned}\mathcal{J}(\phi) =& \mathbb{E}_{c\sim\pi_\phi(\cdot|x)}[R(c)] \\ &- \beta D_{KL}(\pi_\phi(\cdot|x)||\pi_{\text{ref}}(\cdot|x)).\end{aligned}$$

We optimize this objective with Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For each input x, we sample a group of outputs $\{c_1, \ldots, c_G\}$ and estimate the baseline using the group average.

Marginal-Utility Reward. A naive reward based only on the pass rate of the tip-guided solution is noisy: on simple tasks where the base model already succeeds, the policy can receive high reward even for redundant or empty tips. We therefore define the reward by the marginal utility of the tip relative to base generation.

Let P_{base} denote the pass rate of the code generated with the original problem description and P_{curr} denote the pass rate with the generated tip c. We define the Utility Gain Δ to explicitly measure the improvement brought by the tip:

$$\Delta = \begin{cases} 1.0 & \text{if } P_{\text{base}} < 1 \text{ and } P_{\text{curr}} = 1 \\ P_{\text{curr}} - P_{\text{base}} & \text{otherwise} \end{cases}$$

This reward approximates the marginal contribution of the proposed tip before RM-based selection. A positive value indicates that the tip improves the tip-guided branch relative to the base branch, while a negative value indicates regression. Subtracting the base performance filters out redundant tips on already solved tasks and makes the reward depend on the relative contribution of the instruction rather than absolute task difficulty. This centering also reduces reward variance, encouraging policy updates to focus on whether the tip changes the outcome.

Finally, to further encourage conciseness, we incorporate a length penalty using a ReLU style threshold. The total reward $R(c)$ is defined as

$$R(c) = \Delta - \lambda \cdot \max(0, \mathcal{L}(c) - \mathcal{L}_{\text{th}}),$$

where $\mathcal{L}(c)$ is the token length of the tip and $\mathcal{L}_{\text{th}} = 2000$ is the soft threshold.

2.4 Reward-Guided Reranking Inference

At inference time, TIPCODER follows a generate-and-rerank protocol. The Proposer generates a problem-specific tip c for every input problem x. We do not require the Proposer to decide whether the base model would fail on the original prompt, since problem difficulty and failure modes are model-dependent. Instead, TIPCODER constructs two branches: the base branch from the original instruction and the tip-guided branch from the augmented instruction.

- **Base branch:** the target Code LLM generates y_{base} from the original instruction x.
- **Tip-guided branch:** the same target Code LLM generates y_{tip} from the augmented instruction $x \oplus c$.

TIPCODER then applies a Reward Model R as a post-hoc selector:

$$\hat{y} = \begin{cases} y_{\text{tip}}, & R(x, y_{\text{tip}}) > R(x, y_{\text{base}}), \\ y_{\text{base}}, & \text{otherwise.} \end{cases} \quad (6)$$

The deployed objective is therefore to improve the pass rate of the selected output,

$$\max_{\phi} \ \mathbb{E}_{c\sim\pi_\phi(\cdot|x)}\mathbb{E}_{y_{\text{base}},y_{\text{tip}}}\left[\mathbb{1}\big(\text{Pass}(\hat{y}, T)\big)\right]. \quad (7)$$

This protocol completes the exploration-selection decomposition of TIPCODER: the Proposer exposes an alternative instruction-conditioned candidate, and the Reward Model selects between the base and tip-guided solutions for

each instance. This post-hoc selection reduces regressions from unnecessary or noisy tips without requiring the Proposer to predict whether the base model will fail.

TIPCODER is black-box with respect to the target Code LLM. It operates only through prompt-level augmentation and candidate scoring, requiring no access to model parameters, gradients, or execution feedback during inference. Because it explores the instruction dimension rather than replacing solution-level sampling, it remains compatible with test-time scaling methods such as Best-of-N.

3 Experiments

3.1 Setup

Datasets and Metrics. We evaluate on three code-generation benchmarks: HumanEval+ (Chen, 2021), MBPP+ (Austin et al., 2021), and BigCodeBench-Instruct (Zhuo et al.). For HumanEval+ and MBPP+, we use the EvalPlus implementation (Liu et al., 2023), which provides more rigorous test suites than the original benchmarks. HumanEval and MBPP primarily cover algorithmic and entry-level Python programming tasks, while BigCodeBench-Instruct evaluates more realistic programming scenarios involving library usage and complex function calls. Since TIPCODER targets instruction refinement, we evaluate on the instruction-following split of BigCodeBench rather than the completion split. We use Pass@1 as the primary metric. Evaluation tests are used only for reporting final correctness and are not available to TIPCODER during inference.

Proposer Training. The Proposer is initialized from `Qwen3-4B-Instruct-2507`. We use a general instruction model rather than a code-specialized model because tip generation requires identifying constraints, edge cases, and failure modes in natural language. During RL, the Proposer interacts with a frozen code-generation environment instantiated by `Qwen2.5-7B-Coder-Instruct`. This environment is used only for training-time rollouts and reward computation; it is not the sole target model used in evaluation.

For RL training, we use the hard subset of AceCoder-87K and exclude all samples used in the SFT stage. We further downsample extreme cases where the base pass rate is either 0 or 1, so that the policy receives informative marginal-utility signals rather than learning from tasks that are already solved or consistently unsolved by the rollout environment. This yields 13,376 RL training samples. The evaluation benchmarks are held out from Proposer training.

We implement training with OpenRLHF and vLLM. The rollout group size is set to 8. The environment Code LLM uses temperature 0 to provide deterministic execution feedback during RL, while the Proposer samples tips with temperature 1.0 during training rollouts and uses greedy decoding during evaluation. Full hyperparameters are provided in Appendix C.

Target Code LLMs. We evaluate TIPCODER on four open-source Code LLM backbones: `Qwen2.5-Coder-7B-Instruct`, `DeepSeek-Coder-7B-Instruct-v1.5`, `Qwen3-Coder-30B-A3B-Instruct`, and `DeepSeek-Coder-V2-Lite-Instruct`. Unless otherwise specified, the same trained Proposer is applied to all target backbones without target-specific fine-tuning. This setting tests whether the learned instruction proposer transfers across different Code LLMs.

Baselines. We compare against the following baselines. **Base** directly generates code from the original instruction. **Best-of-N** samples multiple candidate solutions from the original instruction. In practice, we set $N = 2$. **Self-Hint** is a zero-shot instruction-proposing baseline. It uses the same tip-generation prompt as TIPCODER, but replaces the trained Proposer with an untrained instruction model to generate the auxiliary tip; the target Code LLM then generates a tip-guided candidate from the resulting augmented instruction. To compare with generic prompt or context optimization, we include **OPRO** (Yang et al., 2023), which treats prompt design as an optimization problem and iteratively proposes improved prompts; **GEPA** (Agrawal et al.), which evolves prompts through reflective feedback; and **CTRL-32B** (Xie et al., 2025), a critique-based model that produces contextual guidance for code revision.

Reward Model. For **all baselines** and TIPCODER, the candidate set consists of the base solution and the corresponding augmented solution. We use the same `AceCodeRM-32B` to select the best candidate, without access to execution feedback or hidden tests at inference time. This keeps the selector fixed across methods, so performance differences reflect the generated candidate set rather than the reward model.

Table 1: Main results on code-generation benchmarks across diverse backbones. Results are reported as mean±std over three independent runs. All methods that require candidate selection use the same `AceCodeRM-32B` Reward Model. Avg. denotes the unweighted average of the three benchmark means.

Model	Method	HumanEval+	MBPP+	BigCodeBench	Avg.
DeepSeek-Coder-7B	Base	62.40±1.86	61.55±2.12	37.43±0.05	53.79
	Best-of-2	69.92±1.86	66.93±2.38	32.54±0.00	56.46
	OPRO	65.85±0.00	68.25±0.00	39.80±0.05	57.97
	GEPA	65.85±0.00	68.25±0.00	39.94±0.05	58.01
	CTRL	67.89±0.93	68.96±0.15	38.39±0.48	58.41
	Self-Hint	69.92±1.86	69.40±0.31	38.77±0.61	59.36
	TIPCODER-SFT	71.34±0.00	71.08±0.81	39.44±0.05	60.62
	TIPCODER-RL	**72.36**±1.76	**71.16**±1.15	**40.00**±0.05	**61.17**
DeepSeek-Coder-V2-Lite	Base	74.39±1.61	68.78±1.40	38.19±0.33	60.45
	Best-of-2	78.25±0.93	71.34±1.00	34.21±0.00	61.27
	OPRO	76.22±1.61	72.13±0.61	40.88±0.09	63.08
	GEPA	77.03±0.35	72.49±0.00	40.61±0.69	63.38
	CTRL	78.66±0.00	72.93±0.40	39.04±0.55	63.54
	Self-Hint	80.49±1.61	71.60±0.15	40.96±1.14	64.35
	TIPCODER-SFT	78.66±0.61	72.13±1.10	39.82±0.23	63.54
	TIPCODER-RL	**80.89**±1.04	**73.46**±0.31	**41.46**±0.43	**65.27**
Qwen2.5-Coder-7B	Base	78.86±1.27	64.46±1.00	41.64±0.05	61.65
	Best-of-2	85.77±1.27	70.37±0.92	36.32±0.00	64.15
	OPRO	85.57±0.35	**73.81**±0.00	43.57±0.05	67.65
	GEPA	85.57±0.35	**73.81**±0.00	42.87±0.05	67.42
	CTRL	85.57±0.35	73.54±0.46	42.08±0.27	67.06
	Self-Hint	86.38±0.35	73.54±0.46	41.49±0.00	67.14
	TIPCODER-SFT	84.96±0.35	73.19±1.07	42.34±0.05	66.83
	TIPCODER-RL	**88.21**±0.35	**73.81**±0.92	**43.89**±0.56	**68.64**
Qwen3-Coder-30B-A3B	Base	86.99±0.93	73.72±0.40	48.39±0.05	69.70
	Best-of-2	87.60±0.70	74.60±1.15	43.16±0.00	68.45
	OPRO	89.02±0.35	74.07±0.46	48.71±0.20	70.60
	GEPA	89.02±0.35	74.34±0.00	48.54±0.18	70.63
	CTRL	87.80±0.00	73.46±0.31	48.22±0.28	69.83
	Self-Hint	88.62±1.27	73.90±0.40	46.29±0.37	69.60
	TIPCODER-SFT	89.02±0.70	74.96±0.15	47.98±0.30	70.65
	TIPCODER-RL	**89.23**±0.93	**75.31**±0.33	**48.77**±0.09	**71.10**

3.2 Main Results

Table 1 reports the main results across four target Code LLMs. All methods requiring candidate selection use the same `AceCodeRM-32B` Reward Model. Prompt- and context-level optimization generally improves over base generation, but the gains vary across backbones and benchmarks. OPRO and GEPA are competitive on `Qwen2.5-Coder-7B` and `Qwen3-Coder-30B-A3B`, but are less stable on `DeepSeek-Coder-7B`. Best-of-2 improves HumanEval+ and MBPP+ in many cases, yet consistently degrades BigCodeBench, suggesting that merely sampling from the original instruction is insufficient for more realistic programming tasks. To further address stronger solution-space scaling baselines, we include an additional Best-of-N study on MBPP+ in Appendix A.

TIPCODER-RL achieves the best average performance on all four backbones, reaching 61.17 on `DeepSeek-Coder-7B`, 65.27 on `DeepSeek-Coder-V2-Lite`, 68.64 on `Qwen2.5-Coder-7B`, and 71.10 on `Qwen3-Coder-30B-A3B`. Compared with the strongest non-TIPCODER baseline on each backbone, this yields average improvements of 1.81, 0.92, 0.99, and 0.47 points, respectively. These results indicate that instance-level instruction proposing provides a more stable scaling direction than generic prompt optimization.

Self-Hint uses the same tip-generation prompt without training the Proposer, serving as a zero-shot instruction-refinement baseline. The trained TIPCODER variants consistently improve over Self-Hint in average performance, showing that the gains come not merely from adding auxiliary in-

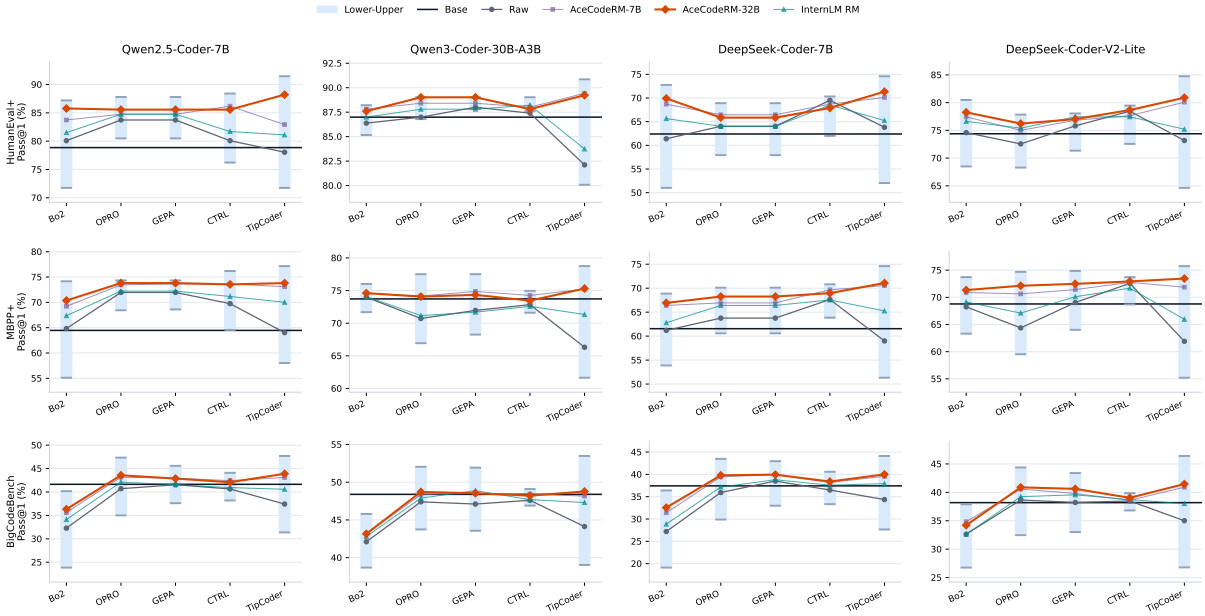

Figure 1: Disentangling candidate exploration from post-hoc selection. For each target model and benchmark, we fix the candidate set produced by each method and vary only the selection rule. The figure explicitly reports base performance, raw adoption of the additional branch, reward-model-selected performance under different Reward Models, and oracle upper/lower bounds over the candidate set. The shaded region denotes the oracle lower-to-upper range, where the upper bound measures the best achievable performance with an ideal selector and the lower bound measures the risk of selecting the worse candidate. A higher oracle upper bound indicates that a method exposes additional correct candidates, while the gap between RM-selected performance and the oracle upper bound reflects the remaining selection bottleneck.

structions, but from learning problem-specific tips aligned with downstream code correctness.

3.3 Disentangling Exploration from Selection

Figure 1 decomposes test-time performance into two factors: candidate exploration and post-hoc selection. For each method, the figure explicitly reports the base performance, raw adoption of the additional branch, reward-model-selected performance under different Reward Models, and the oracle upper/lower bounds over the candidate set. The oracle upper bound measures the best achievable performance if an ideal selector could choose between the base and additional candidates, while the lower bound measures the worst possible selection. Thus, the gap between base performance and the oracle upper bound reflects the exploration potential introduced by the method, whereas the gap between reward-model selection and the oracle upper bound reflects how much of this potential remains unrealized by the selector.

The results show that instruction-guided generation can expose useful candidates beyond the original prompt. In many settings, TIPCODER-RL raises the oracle upper bound, indicating that learned tips create complementary solution paths rather than merely perturbing the prompt. However, this potential is not automatically realized: raw adoption can select harmful candidates, and weaker selectors may fail to recover the improved branch. Stronger code-oriented Reward Models, such as `AceCoderM-32B`, generally convert more of the oracle potential into final pass-rate gains than weaker or less aligned selectors. Therefore, the Reward Model is not the source of the exploration benefit; it determines how much of the candidate-space improvement opened by the Proposer can be realized at inference time.

3.4 Inference Cost Analysis

We report inference cost in Table 2. To make the comparison interpretable, we decompose token usage into four parts: *native code generation*, the original code-generation branch; *extra code generation*, additional code candidates introduced by the method; *introduced non-code generation*, such as tips or critiques; and *RM scoring*, the reward-model selection cost. TIPCODER adds two inference components: tip generation and tip-guided code generation. As shown in Table 2, this leads to

Table 2: Inference cost breakdown on Humaneval+ dataset. Panel (a) reports token cost as prompt/completion/total (P/C/T). Panel (b) reports wall-clock latency in seconds. All methods in this table use reward-model reranking with the same selector. GEPA's offline prompt-search cost is excluded because it is not incurred at test time.

(a) Token cost breakdown, reported as P/C/T					
Method	**Native code gen.**	**Extra code gen.**	**Introduced non-code**	**RM**	**Total**
TIPCODER	206.22 / 276.43 / 482.65	428.67 / 293.14 / 721.81	259.54 / 229.66 / 489.20	1118.48 / 0.00 / 1118.48	2012.91 / 799.23 / 2812.13
Best-of-2	206.22 / 276.43 / 482.65	0.00 / 276.43 / 276.43	0.00 / 0.00 / 0.00	1118.48 / 0.00 / 1118.48	1324.70 / 552.85 / 1877.55
GEPA	228.66 / 253.98 / 482.65	340.80 / 279.16 / 619.96	0.00 / 0.00 / 0.00	952.77 / 0.00 / 952.77	1522.23 / 533.14 / 2055.38
CTRL	228.66 / 434.05 / 662.71	1060.46 / 417.30 / 1477.75	778.67 / 356.35 / 1135.04	1086.67 / 0.00 / 1086.67	3154.46 / 1207.70 / 4362.17
(b) End-to-end wall-clock latency, reported in seconds					
Method	**Native code**	**Extra code**	**Introduced non-code**	**RM**	**E2E wall**
TIPCODER	1.282	1.411	0.683	0.333	2.428
Best-of-2	1.282	0.165	0.000	0.333	1.781
GEPA	1.282	1.376	0.000	0.302	1.679
CTRL	1.401	1.884	1.586	0.332	5.203

higher token usage and latency than Best-of-2 and GEPA, but substantially lower cost than CTRL, whose critique-and-revision pipeline introduces long non-code generation and an additional revision stage. Thus, TIPCODER should be viewed as a moderate-cost instruction-level exploration method rather than a latency-minimizing alternative to sampling. Together with the Best-of-N scaling study in Appendix A, these results suggest that instruction-guided exploration provides an orthogonal search direction to solution-space sampling.

4 Related Work

Iterative Refinement with Code LLMs Iterative refinement constitutes a post-hoc scaling strategy where models systematically correct outputs based on feedback. Originating in general reasoning domains approaches like Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) prompted models to critique their own generations using intrinsic knowledge. In the context of code generation this methodology has evolved to leverage objective execution signals such as compiler errors to enhance debugging capabilities (Zheng et al., 2024). Contemporary research further internalizes this refinement through training objectives exemplified by SCoRe (Kumar et al.) which optimizes the entire generation trajectory via reinforcement learning and ReflectionCoder (Ren et al., 2025) which distills knowledge from execution-based reflection sequences. CTRL (Xie et al., 2025) trains LLMs to critique without human supervision, enabling them to supervise stronger models and achieve test-time scaling through iterative critique-revisions. However, these methods necessitate deployed execution environments and explicit feedback loops introducing significant workflow complexity. TIPCODER addresses this by distilling the foresight from iterative repair into a lightweight proactive tip achieving refinement benefits in a single inference pass without runtime execution.

Prompt Optimization and Scaling Prompt optimization functions as inference-time scaling by refining input queries to elicit superior responses. General domain research treats instruction generation as a search problem where models iteratively propose and score candidate prompts (Zhou et al., 2022). Recent frameworks like DIPPER (Hu et al., 2025) further demonstrate that enforcing prompt diversity effectively unlocks varied reasoning paths. These advancements establish that allocating compute to explore the instruction space yields predictable gains comparable to increasing model size (Snell et al., 2025). Despite this success, systematic instruction-level scaling remains underexplored in code generation where methods typically rely on stochastic solution sampling. TIPCODER addresses this gap by introducing *Test-time Instruction Proposing* to code generation. We learn a proposer policy to efficiently search for reasoning tips and convert inference compute into correctness gains through instruction-space optimization.

5 Conclusion

We presented TIPCODER , a model-agnostic framework that shifts test-time scaling to the orthogonal instruction space. By training a Proposer via SFT and RL to generate latent auxiliary instructions, TIPCODER effectively guides models out of local optima without requiring runtime execution. Experiments confirm it achieves superior sample efficiency compared to stochastic sampling. Future directions include building robust environment with multiple coder LLMs, stronger rerank models, and extending this paradigm to general domains.

Limitations

TIPCODER has three main limitations. First, it introduces additional inference cost by generating an auxiliary tip and a tip-guided code candidate before reward-model selection. Second, its realized gains depend on the quality of the post-hoc Reward Model: learned tips may expose better candidates, but an imperfect selector may fail to choose them. Third, although the Proposer transfers across multiple target Code LLMs in our experiments, it is trained with a specific rollout environment and debugging-data construction pipeline, so its guidance may still reflect the error patterns of the training environment. Extending TIPCODER to broader model families, programming languages, and repository-level tasks remains future work.

Acknowledgments

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, and 1 others. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *First Workshop on Foundations of Reasoning in Language Models*.

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*.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Self-debugging: Large language models for code generation with self-debugging. *arXiv preprint arXiv:2303.17651*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, 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*.

Wenyang Hu, Gregory Kang Ruey Lau, Liu Diwen, Chen Jizhuo, See Kiong Ng, and Bryan Kian Hsiang Low. 2025. Dipper: Diversity in prompts for producing large language model ensembles in reasoning tasks. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 35546–35560.

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*.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, and 1 others. Training language models to self-correct via reinforcement learning. In *The Thirteenth International Conference on Learning Representations*.

Sarah Larbi, Julian Smith, and Mark Thompson. 2024. When prompts go wrong: Evaluating code model robustness to ambiguous and incomplete task descriptions. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

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.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. 2025. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 20286–20332.

Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. 2025. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9999–10020.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling llm test-time compute optimally can be more effective than scaling parameters

for reasoning. In *The Thirteenth International Conference on Learning Representations*.

Zhihui Xie, Jie Chen, Liyu Chen, Weichao Mao, Jingjing Xu, and Lingpeng Kong. 2025. Teaching language models to critique via reinforcement learning. In *International Conference on Machine Learning*, pages 68559–68577. PMLR.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 12834–12859.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*.

Terry Yue Zhuo, Vu Minh Chien, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *The Thirteenth International Conference on Learning Representations*.

A Comparison with Best-of-N Scaling on MBPP+

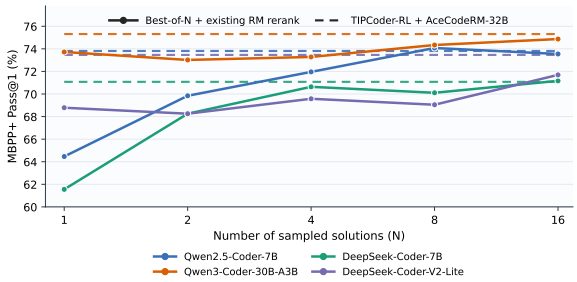

Figure 2: Comparison between solution-space scaling and instruction-guided scaling on MBPP+. Solid curves show Best-of-N sampling with RM-based reranking for $N \in \{1, 2, 4, 8, 16\}$. Dashed horizontal lines show TIPCODER-RL with `AceCodeRM-32B`.

Figure 2 compares TIPCODER-RL with Best-of-N sampling on MBPP+ across four target Code LLM backbones. Best-of-N improves performance by expanding the solution space under the original instruction, while TIPCODER changes the conditioning context through an instance-specific auxiliary tip. The comparison shows that TIPCODER-RL reaches a performance level competitive with larger sampling budgets, indicating that instruction-guided exploration can provide an efficient alternative search direction beyond repeatedly sampling from the same prompt.

This result complements the main evaluation: solution-space sampling and instruction-space exploration are not mutually exclusive. Best-of-N increases the number of candidate programs, whereas TIPCODER aims to expose a more useful candidate by modifying the problem-specific instruction. The MBPP+ scaling curve suggests that learned tips can recover part of the benefit of larger sampling budgets with a different form of test-time computation.

B Discussion about Post-hoc Verification Strategy

The post-hoc verification strategy is favored over pre-hoc refusal to maintain proposer diversity and adaptability. Training a refusal mechanism often leads to reward hacking, resulting in conservative policies that suppress high-value guidance. Furthermore, since problem difficulty is model-dependent, the proposer cannot accurately predict failure without access to the generator's internal state. By utilizing a specialized RM to filter candidates, TIPCODER implements a dynamic, model-adaptive scaling strategy that maximizes the available potential range without parameter intervention.

C Hyperparameter Settings

To ensure the reproducibility of our results, we detail the hyperparameter configurations used in our experiments. Our training pipeline consists of two stages: Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL).

In the SFT stage, we fine-tuned the `Qwen3-4B` model using the Adam optimizer with state offloading to optimize memory efficiency. We employed a maximum sequence length of 4096 to accommodate long code contexts and set the learning rate to 3e-5.

For the RL stage, we utilized the OpenRLHF framework to implement the GRPO algorithm. To maintain stability during the alignment process, we used a lower learning rate of 1e-6 for the actor model and initialized the KL divergence coefficient

at 1e-3. We employed group normalization for advantage estimation and applied reward normalization to stabilize the training scale. Additionally, a length penalty of 1e-5 was applied to the reward model to discourage verbosity. We conducted all experiments on a computational node equipped with 4 NVIDIA A800 GPUs. To efficiently compute the KL divergence constraint during the GRPO phase we allocated one dedicated GPU for the reference model inference while the remaining three GPUs were utilized for optimizing the policy model. Consequently the global batch sizes were specifically configured to be divisible by 3 to ensure balanced data parallelism across the active training devices.

Table 3 presents the comprehensive list of hyperparameters.

Hyperparameter	**Value**
Common Settings	
Base Model	Qwen3-4B-Instruct-2507
Precision	BF16
Zero Stage	2
Gradient Checkpointing	True
Stage 1: Supervised Fine-Tuning	
Optimizer	Adam (Offload)
Learning Rate	3×10^{-5}
Max Sequence Length	4096
Micro Batch Size	4
Max Epochs	1
Logging Steps	1
Stage 2: Reinforcement Learning (GRPO)	
Framework	OpenRLHF
Actor Learning Rate	1×10^{-6}
Initial KL Coefficient (β)	1×10^{-3}
Advantage Estimator	GRPO
Reward Normalization	True
Reward Length Penalty	1×10^{-5}
Batch Configuration	
Train Batch Size	96
Rollout Batch Size	192
Micro Train Batch Size	8
Micro Rollout Batch Size	8
Samples per Prompt (N)	8
Generation Specs	
Prompt Max Length	4096
Generate Max Length	4096
Training Loop	
Max Epochs	1
Num Episodes	1

Table 3: Comprehensive hyperparameter settings for SFT and RL stages.

D Case Study

Figure 3 presents a representative example where the original problem instruction is ambiguous about whether the input lists have equal lengths. The base model assumes equal lengths and iterates over the first list, which can cause an index error when the other lists are shorter. TIPCODER generates tips that explicitly warn about variable-length inputs and index alignment. Conditioned on these tips, the generator uses the minimum list length and handles the edge case correctly. This example illustrates the main role of the Proposer: it does not provide the solution code, but surfaces problem-specific constraints and pitfalls that steer the generator away from a likely failure mode.

E Prompt Templates

We present the comprehensive prompt specifications utilized throughout the TIPCODER framework. The templates are categorized into offline data construction protocols and online inference workflows. For data construction we design two distinct mechanisms where the first employs direct contrast between incorrect and correct solutions to distill static tips while the second leverages multi-turn debugging trajectories to extract execution-based reasoning logic. The inference process follows a structured three-stage pipeline involving a Proposer for tip generation and a Generator for code synthesis followed by an Evaluator for quality scoring. Visualizations of these templates are provided in Figures 4, 5, 6 to illustrate the input-output format and system instructions.

CASE STUDY: HANDLING IMPLICIT CONSTRAINTS (MBPP/142)

[User Instruction]

Write a function to count number items that are identical in the same position of three given lists.

Note: The prompt is ambiguous regarding whether the three lists have equal lengths.

GENERATED TIPS (TIPCODER):

1. Index Alignment: Ensure that you compare elements at the same index across all lists. If the lists are of unequal length, you must handle cases where one list ends before another.
2. Variable Lengths: Always check if the lists are empty or have different lengths before attempting to iterate. Avoid accessing indices that do not exist.
3. Edge Cases: Handle cases where the lists are empty or have different lengths gracefully.

BASELINE (FAIL)

Logic: Iterates by List 1

```python
def count_samepair(list1, list2, list3):
    count = 0
    # DANGER: Implicitly assumes equal lengths
    # Will crash if list2/list3 are shorter
    for i in range(len(list1)):
        if list1[i] == list2[i] == list3[i]:
            count += 1
    return count
```

Analysis: The model assumes fixed lengths, leading to potential `IndexError`.

TIPCODER (SUCCESS)

Logic: Robust Length Handling

```python
def count_samepair(list1, list2, list3):
    # Handle edge cases
    if not list1 or not list2 or not list3:
        return 0

    # SAFE: Use minimum length
    min_len = min(len(list1), len(list2), len(list3)
        )

    count = 0
    for i in range(min_len):
        if list1[i] == list2[i] == list3[i]:
            count += 1
    return count
```

Analysis: Guided by the tip, the model uses `min()` to prevent crashes.

Figure 3: Comparison of code generation with and without TIPCODER. The baseline model fails to account for variable list lengths due to prompt ambiguity, while TIPCODER proactively generates warnings about "Variable Lengths", leading to a robust solution.

Data Construction A: Direct Contrast Distillation

[System]
You are given a coding problem and a solution that failed some test cases.
Task: Please fix the code to pass all test cases.

[User]
<Problem Instruction>
Problem:
Implement a function `valid_parentheses(s)` that takes a string of parentheses...

<Your Previous Solution>
Code:

```
    def valid_parentheses(s):
        count = 0
        for char in s:
            if char == '(': count += 1
            elif char == ')': count -= 1
        return count == 0
# Bug: No negative check
```

<Execution Feedback>
Failed Test Cases:
1. `assert valid_parentheses("))(()") == False` (Expected: False, Got: True)
2. `assert valid_parentheses("())(") == False` (Expected: False, Got: True)

[Assistant]
<Analysis & Fix>
The previous solution fails to handle cases where a closing parenthesis appears before an opening one... To fix this, we must ensure `count` never drops below zero.
Fixed Code:

```
    def valid_parentheses(s):
        count = 0
        for char in s:
            if char == '(': count += 1
            elif char == ')': count -= 1
            if count < 0: return False # Fixed
        return count == 0
```

Figure 4: The prompt template used to distill tips by directly comparing an incorrect solution with a correct one. This simulates the teacher identifying the gap between the student's attempt and the ground truth.

Data Construction B: Trajectory-based Distillation

[System Directive]
You are analyzing a programming debugging conversation where a human and an AI assistant are working together to fix code issues. The conversation has been **masked** to hide specific code implementations and test cases.

Task: Extract and summarize key programming tips. Focus on: Problem Analysis, Missing Requirements, Best Practices, and Common Pitfalls.
Constraint: Do not provide solutions or code snippets. Be concise.

[User Input: Masked Debugging Conversation]
<Step 1>
User: Implement a function to find the longest substring without repeating characters.
Assistant: Here is the solution:
`[CODE_BLOCK_MASKED]`
System: Execution failed.
`[TEST_CASES_MASKED]`

<Step 2>
User: The previous code failed on empty string input. Please fix it.
Assistant: I missed the edge case checks. Here is the fixed version:
`[CODE_BLOCK_MASKED]`
System: Success! All tests passed.

[Assistant Response: Extracted Tips]
Based on the debugging process, here are the key tips for this problem:
- **Edge Case Handling:** Always validate if the input string is empty or None at the beginning to avoid runtime errors.
- **Sliding Window Strategy:** When maintaining the window of unique characters, ensure the left pointer is updated correctly to optimize efficiency.
- **State Management:** Use a dictionary to store the most recent index of each character for $O(1)$ lookups.

Figure 5: Trajectory-based Distillation. The template employs a masking strategy to abstract away specific implementation details from multi-turn debugging logs, forcing the model to synthesize high-level optimization tips based on the execution feedback loop.

Inference Workflow

<Step 1: Proposer (Tip Generation)>
[Input]
System: You are an expert programming mentor. Analyze the following coding instruction and provide specific tips to help developers implement this correctly. Do not provide solution or any solution-related ideas.
Your only task is to provide tips to clarify the instruction and identify potential pitfalls, common mistakes, and key considerations during solving the problem. Your tips should be problem-specific, not general programming principles. Do not conclude debug/test/verify tips.
User: {PROBLEM_DESCRIPTION}

[Output]
Assistant (Proposer): `{GENERATED_TIP}`
(The model predicts a tip based on the problem description.)

<Step 2: Generator (Code Synthesis)>
[Input]
User: `{GENERATED_TIP}` {PROBLEM_DESCRIPTION}
(The tip is prepended to the user query as a hint.)

[Output]
Assistant (Generator):

```
def solution(nums):
    # Generated code...
    return result
```

<Step 3: Evaluator (Reward Model Scoring)>
[Input Format]
The Reward Model accepts the interaction history as a structured chat template:
`[ {"role": "user", "content": PROBLEM}, {"role": "assistant", "content": GENERATED_CODE} ]`

[Process]
`tokenizer.apply_chat_template(chats, return_tensors="pt")` → `AceCodeRM(input_tokens)`

[Output]
Scalar Reward: `23.0688`
(A higher score indicates higher confidence in correctness, used for reranking or RL feedback.)

Figure 6: The complete inference pipeline. In Step 1, the Proposer predicts a tip. In Step 2, the Generator synthesizes code using the tip. Finally, in Step 3, the AceCodeRM evaluates the quality of the (Problem, Code) pair to assigning a scalar reward.