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

Anonymous ACL submission

Abstract

Conventional test-time scaling strategies primarily focus on solution-space expansion through stochastic sampling. In contrast, we explore the untapped potential of the instruction space with TIPCODER, a framework that utilizes a specialized Proposer agent to synthesize auxiliary tips. The Proposer is trained via Supervised Fine-Tuning to capture the linguistic structure of effective guidance and further optimized through Reinforcement Learning to search the sparse instruction landscape for tips that maximize code correctness. During inference, TIPCODER employs a Reward Model to select the optimal solution from generated candidates. As a model-agnostic external module, our framework enhances Code LLMs through instruction-level intervention without requiring parameter access or runtime execution feedback. Extensive evaluations across diverse benchmarks demonstrate that TIPCODER significantly outperforms baselines and achieves performance parity with high-cost stochastic sampling strategies at a fraction of the computational cost, establishing instruction-space exploration as a highly effective and scalable dimension for test-time enhancement. [1]

1 Introduction

The emergence of Code Large Language Models (Guo et al., 2024; Hui et al., 2024) has fundamentally reshaped software development by demonstrating an unprecedented ability to translate high-level human intent into functional implementations. This transformative capability is rooted in the model's sensitivity to the initial natural language instruction, which serves not merely as a task description but as the primary anchor that dictates the entire subsequent reasoning trajectory. Standard test-time strategies such as Best-of-N sampling (Brown et al., 2024) attempt to improve performance by allocating additional compute to explore the *solution space* (Snell et al., 2025; Muennighoff et al., 2025). However, this brute-force approach ignores the latent potential of the instruction space and faces diminishing returns when the initial prompt fails to elicit a proper reasoning path.

We argue that exploring the instruction space through learned intervention offers a cleaner and more computationally efficient alternative to exhaustive solution sampling. This perspective treats the instruction as a high-dimensional control surface where precise adjustments can significantly steer the internal reasoning of the model before generation even begins. By operating in the instruction space, we can effectively prune the search space and guide the model toward correct implementations without the need for expensive repeated trials. Furthermore, this intervention is inherently model-agnostic because it occurs at the input level and requires no access to the internal weights or gradients of the target Code LLM, making it a versatile solution that can be seamlessly integrated into existing model architectures.

Building on this paradigm shift, we propose TIPCODER, a framework that explores the instruction space by generating task-specific technical hints. We introduce a `Proposer` agent that distills the reasoning gains typically found in multi-turn debugging trajectories into a single-turn intervention. Instead of relying on brute-force sampling to find a correct program, our `Proposer` synthesizes auxiliary instructions, namely *tips*, that preemptively refine the reasoning trajectory and steer the generation distribution. This approach not only activates the latent knowledge within the model but also ensures high sample efficiency by optimizing the starting point of the code generation process.

However, operationalizing this is non-trivial. The landscape of high-utility tips is highly sparse. While Supervised Fine-Tuning (SFT) allows the Proposer to mimic the linguistic format of debugging advice, it often fails to capture the causal util-

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

ity required to fix specific logic errors. To address this, we employ Reinforcement Learning (RL) to actively search this sparse landscape for latent instructions that maximize downstream correctness. Furthermore, since aggressive instruction injection carries the risk of introducing noise, we incorporate a robust *Generate-and-Rerank* mechanism. By leveraging a reward model (Zeng et al., 2025) to perform a Best-of-2 selection between the base and tip-augmented outputs, we establish a safety gate ensuring that tips are adopted only when they effectively resolve ambiguity.

Our contributions are summarized as follows:

- **Test-time Instruction Proprosing** We systematically investigate the instruction space as a critical dimension for code generation and demonstrate that steering the reasoning trajectory at the input level offers superior sample efficiency compared to brute-force solution sampling.

- **Progressive Policy Learning** We propose a training pipeline where the `Proposer` first distills multi-turn debugging insights via supervised fine-tuning and subsequently optimizes the guidance policy through reinforcement learning to effectively compress expensive iterative reasoning into a single pass of natural language intervention.

- **Model-Agnostic Framework** We introduce an architecture that enhances various Code LLMs through prompt-level intervention and this allows for seamless integration with off-the-shelf models as it eliminates the need for internal weight access or complex runtime execution environments.

2 Methodology

In this section, we introduce TIPCODER, a framework that distills the reasoning capabilities of multi-turn interactions into a deterministic instruction augmentation process, as shown in Figure 1. This section begins by establishing the problem formulation and the underlying optimization objective. We then explicate the training methodology, focusing specifically on systematic data construction and policy optimization. Finally, we introduce the Reward-Guided Reranking framework employed during inference.

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 Data Construction

Training the proposer requires high-quality pairs of (x, c) where the instruction c encapsulates the feedback information implicit in multi-turn trajectories.. We construct this dataset by distilling effective repair strategies from multi-turn interaction trajectories using a three-phase pipeline consisting of mining, abductive reasoning, and verification.

Mining Repair Trajectories We utilize AceCoder-87K, a large-scale code instruction dataset constructed through evolutionary instruction generation from open-source repositories, as our seed corpus. To ensure the training data contains sufficiently dense reasoning signals, we focus exclusively on challenging samples where the student model exhibits reasoning deficits. We specifically filter for problems where the standard one-off generation yields a low pass rate (Pass@1 $< 50\%$). For these samples, we employ Gemini-3-Pro as teacher model to perform iterative debugging. In each turn, the model receives execution feedback

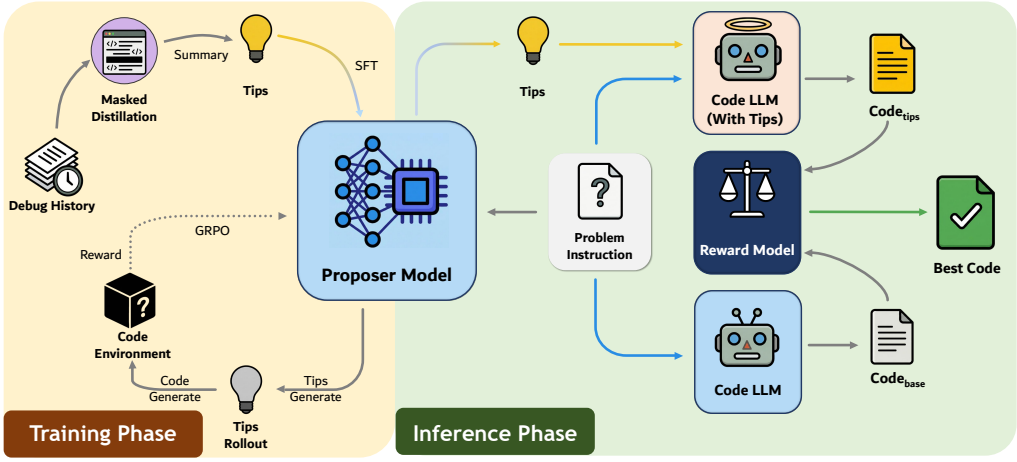

Figure 1: The overall framework of TIPCODER. The training phase (left) consists of two stages: initializing the Proposer Agent via SFT, followed by Reinforcement Learning using pass rates from the code environment. The inference phase (right) employs the Proposer Model to generate guiding tips, and a Reward Model is used to select the best solution between the base and guided codes.

and generates a refined solution. We retain trajectories where the code initially failed but was successfully repaired within three turns as these instances capture the valid reasoning path required to resolve the error.

Abductive Reasoning via Masked Distillation We employ an abductive reasoning approach to construct our training data. The core objective is to condense the foresight information typically gained through costly iterative refinement into a single instructional cue. We prompt a powerful teacher model to analyze the problem solution pair and synthesize a reasoning tip that carries equivalent utility to a multi turn repair dialogue. This process effectively distills the retrospective knowledge of the correct solution into a proactive guidance signal allowing the student model to access critical algorithmic insights at the initial generation stage without the need for actual interaction.

We enforce a strict *Logical Abstraction* principle to prevent solution leakage. By rigorously masking code blocks, we ensure tips serve as high-level conceptual guidance. Specifically, the generated instructions focus on identifying edge cases and potential errors rather than dictating specific implementations. This approach preserves the intrinsic coding capability of the base model while ensuring the guidance remains model-agnostic.

Iterative Feedback Verification To ensure the distilled tips are causally effective, we perform a closed-loop verification. We append the generated tip c to the original instruction x and feed it to the target student model. If the generated code still fails, the execution feedback is returned to the teacher model to further refine the tip, limited to a maximum of 2 rounds. A pair (x, c) is added to the final training set if and only if the tip successfully guides the student model to pass all unit tests. This rigorous filtering process yielded a final high-quality dataset of 2,126 pairs.

2.3 Policy Optimization

We formulate the instruction generation as a probabilistic inference problem where the goal is to search for a latent reasoning tip that maximizes the code generation quality. The optimization pipeline proceeds in two stages: Supervised Fine-Tuning for initialization and Reinforcement Learning for utility maximization.

Supervised Fine-Tuning (SFT) We initialize the Proposer policy π_ϕ using the distilled dataset $\mathcal{D}_{train}$. This phase is crucial for warm-starting the policy ensuring the model learns the linguistic format of effective tips and internalizes the reasoning patterns of the teacher. The objective is to minimize the standard negative log-likelihood loss on the high-quality repair trajectories. While SFT enables the model to generate plausible tips it does not strictly align with the downstream objective of code correctness which motivates the subsequent reinforcement learning phase.

Latent Instruction Formulation To bridge the gap between the supervised initialization and the downstream correctness objective we transition to a probabilistic framework. We treat the reasoning tip c as a latent variable that mediates the relationship

between the problem x and the correct solution y^*.

We formulate the optimization goal as maximizing the marginal log likelihood $\log P(y^*|x)$. To make this tractable we employ variational inference. Crucially we utilize the Proposer policy $\pi_\phi(\cdot|x)$ initialized during the SFT phase as the variational posterior. We further align the prior distribution $P(c|x)$ with the frozen SFT reference model $\pi_{\text{ref}}(\cdot|x)$ to ensure semantic stability. By applying Jensen's inequality in the discrete instruction space we derive the Evidence Lower Bound (ELBO):

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

Here $P(y^*|x,c)$ denotes the likelihood of the fixed Code LLM generating the correct solution given the tip. This formulation explicitly maps the SFT policy π_ϕ into the RL objective: the reconstruction term drives the policy to maximize the expected correctness reward while the regularization term anchors it to the π_{ref} prior preventing deviations from coherent natural language.

Reinforcement Learning Objective To operationalize the optimization of the derived ELBO we cast the problem within a reinforcement learning framework. 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 from the policy generates code y 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:

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

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

Reward Function Design To align the proxy reward with the reconstruction objective we design a utility focused reward function. A naive approach of directly using the pass rate P_{curr} as reward is suboptimal because it introduces significant noise. On simple tasks where the base model already succeeds the policy receives high rewards even for generating meaningless or empty tips. To mitigate this we propose a differential reward scheme driven by **Marginal Utility**.

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. Crucially we amplify the reward when the tip successfully repairs a previously failing solution

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

This piecewise design offers three distinct advantages. First by subtracting the baseline performance it acts as a noise filter that assigns zero reward to redundant tips on already solved problems. Second it imposes a natural penalty on performance regression where $\Delta < 0$ effectively discouraging the generation of misleading instructions. Third from an optimization perspective this baseline subtraction centers the reward distribution near zero. This significantly reduces the variance of gradient estimation ensuring that the policy is updated based on the relative contribution of the tip rather than the absolute task difficulty.

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

Unlike previous methods that rely on the model's internal confidence to decide when to intervene, TIPCODER adopts a Generate-and-Rerank paradigm. This architecture delegates the responsibility of "judging task difficulty" from the generative policy to a specialized Reward Model, allowing for a more aggressive exploration of the solution space. The inference process operates as follows:

- **Aggressive Proposal.** We employ an aggressive proposer π_ϕ that unconditionally generates a reasoning tip c for every input problem x. The core insight here is to utilize the tip as a perturbation vector to significantly shift the

generative distribution of the Code LLM. By transforming the conditioning context from $P(y|x)$ to $P(y|x, c)$, we force the model to exit its default reasoning path and explore a distinct region of the solution space. This allows the model to potentially access correct solutions that were assigned low probability in the original distribution.

- **Parallel Generation & Scoring.** We execute two parallel generation paths: a *Base Path* using the original instruction x to generate y_{base}, and an *Augmented Path* using the tip-enhanced instruction $x \oplus c$ to generate y_{prop}. We then employ a reword model, AceCoder-RM (Zeng et al., 2025) $\mathcal{R}$ to independently evaluate both candidates. The RM serves as a high-precision discriminator, assigning scalar scores s_{base} and s_{prop} based on the likelihood of functional correctness.

- **Robust Reranking.** We perform a **Best-of-2** reranking of the generated candidates based on their scalar scores. The final output $\hat{y}$ is determined by promoting the highest scoring candidate to the top rank

$$\hat{y} = \begin{cases} y_{prop} & \text{if } s_{prop} > s_{base} \\ y_{base} & \text{otherwise} \end{cases}$$

This framework offers two key advantages. First, it effectively mitigates the risk of performance regression on simple tasks by falling back to the base solution whenever the tip introduces noise. Second, the method is **model-agnostic** and orthogonal to scaling techniques. Since TIPCODER operates purely on the prompt level without requiring access to the base model's weights, it can be seamlessly integrated with test-time scaling strategies (e.g., Best-of-N sampling on the solution level) to achieve cumulative performance gains.

3 Experiments

3.1 Setup

Datasets We conduct evaluations on three diverse code generation benchmarks including HumanEval+ (Chen, 2021) and MBPP+ (Austin et al., 2021). For these datasets we leverage the rigorous implementation from EvalPlus (Liu et al., 2023). HumanEval and MBPP consists of algorithmic problems or entry level python tasks. Furthermore we incorporate BigCodeBench (Zhuo et al.) to evaluate the model's capability on complex tasks involving intricate library function calls and realistic programming scenarios. Given that our proposed method targets instruction refinement we specifically test on the BigCodeBench-Instruct split rather than the completion subset. We adopt Pass@1 as the primary evaluation metric across all datasets.

Implementation Details We utilize `Qwen3-4B-Instruct-2507` as the backbone for our Proposer policy. We selected the general model as it outperformed the code specific model during SFT while offering broader non-code reasoning capabilities. For the environment of code generation, we utilize `Qwen2.5-7B-Coder-Instruct`. The choice of this specific model size is driven by the computational efficiency requirements of the rollout phase which necessitates sequential inference calls for both the policy and the environment.

For the reinforcement learning training data we utilize the hard subset from the AceCoder-87K dataset. To prevent data leakage we strictly exclude any samples used during the SFT phase. Furthermore to ensure the model learns from meaningful advantage signals we apply strategic downsampling to extreme cases where the baseline pass rate is either 0 or 1. This filtering process eliminates tasks that are either too trivial or completely unsolvable for the current model resulting in a final curated dataset of 13,376 samples.

We implement the training pipeline based on the OpenRLHF framework integrated with vLLM 0.11. During the Reinforcement Learning phase we set the rollout group size to 8. To ensure deterministic feedback from the environment the Code LLM operates with a temperature of 0 during both training and inference. Conversely the Proposer policy utilizes a temperature of 1.0 during training rollout to encourage exploration but switches to greedy decoding during final inference to ensure stability. Comprehensive hyperparameter configurations are provided in the Appendix.

Baselines We assess the generalizability of our framework by applying it across four state of the art open source code models. Specifically we evaluate `Qwen2.5-Coder-7B-Instruct` and `DeepSeek-Coder-7B-Instruct-v1.5` as representative 7B parameter models. To test scalability we also include larger and more capable models including `Qwen3-Coder-30B-A3B-Instruct` and `DeepSeek-Coder-V2-Lite-Instruct`. We report the performance in a standard Zero shot setting

Table 1: Main results on code generation benchmarks across diverse backbones. TIPCODER (SFT+RL) denotes the ensemble strategy that aggregates candidates from both SFT and RL-optimized proposers followed by reward model reranking. The results highlight the superior sample efficiency of instruction-space exploration compared to the computationally expensive Best-of-4 stochastic sampling.

Model	HumanEval (+)		MBPP (+)		BigCodeBench-I		Avg.
	Base	Plus	Base	Plus	Full	Hard	
DeepSeek-Coder-7B-v1.5	70.1	62.2	74.3	65.6	37.2	14.2	53.9
Best-of-4	75.6	70.7	79.1	68.3	**40.7**	**15.5**	58.3
TIPCODER (Instruct)	75.0	68.3	78.8	68.2	37.5	14.9	57.1
TIPCODER (SFT)	79.9	72.0	78.8	70.1	38.0	**15.5**	59.1
TIPCODER (RL)	78.7	70.1	80.4	70.1	38.7	**15.5**	58.9
TIPCODER (SFT+RL)	**81.1**	**73.8**	**81.8**	**71.2**	39.0	**15.5**	**60.4**
DeepSeek-V2-Lite	81.1	75.0	82.0	68.8	37.5	17.6	60.3
Best-of-4	83.5	78.0	82.8	69.6	38.8	22.3	62.5
TIPCODER (Instruct)	86.0	78.0	**86.0**	71.4	38.9	18.2	63.1
TIPCODER (SFT)	85.4	78.7	84.7	70.6	39.1	21.6	63.4
TIPCODER (RL)	86.0	79.3	85.2	72.5	40.4	22.3	64.3
TIPCODER (SFT+RL)	**88.4**	**81.1**	**86.0**	**72.8**	**41.3**	**25.7**	**65.9**
Qwen2.5-Coder-7B-Instruct	91.5	84.8	82.3	70.1	41.3	16.9	64.5
Best-of-4	90.9	83.5	86.5	71.2	40.0	19.6	65.3
TIPCODER (Instruct)	92.1	**86.0**	86.5	72.2	41.6	16.9	65.9
TIPCODER (SFT)	92.7	85.4	86.5	72.8	42.0	18.9	66.4
TIPCODER (RL)	92.7	**86.0**	87.8	73.3	42.0	19.6	66.5
TIPCODER (SFT+RL)	**93.9**	**86.0**	**88.4**	**74.1**	**42.2**	**20.3**	**67.5**
Qwen3-Coder-30B-A3B	93.3	87.2	88.4	73.0	48.2	30.4	70.1
Best-of-4	**93.9**	87.2	87.3	72.8	**50.3**	30.4	70.3
TIPCODER (Instruct)	**93.9**	86.6	88.6	73.3	49.0	30.4	70.3
TIPCODER (SFT)	93.3	87.2	89.7	73.3	48.1	30.4	70.3
TIPCODER (RL)	**93.9**	**87.8**	89.7	**74.9**	49.0	**31.1**	71.1
TIPCODER (SFT+RL)	**93.9**	**87.8**	**90.5**	**74.9**	49.0	**31.1**	**71.2**

where the model generates code directly from the original instruction. Furthermore, we compare our method against Best-of-N (Brown et al., 2024) sampling to distinguish the improvements gained from reasoning guidance versus solution reranking.

3.2 Main Results

We present a comprehensive evaluation of TIPCODER across multiple benchmarks in Table **??**, where the experimental data demonstrate significant and consistent performance leaps across state-of-the-art backbones. Notably, on the high-performance Qwen2.5-Coder-7B-Instruct model, the full TIPCODER framework further enhances the average performance by 3.0%, markedly outperforming both the base model and the Best-of-4 sampling strategy despite the latter's higher computational overhead. Such superior sample efficiency validates that targeted intervention in the instruction space is more effective than stochastic searching in the solution space, as it elicits the latent reasoning potential of high-performing models.

Regarding complex programming scenarios with intricate logical dependencies, TIPCODER demonstrates exceptional capabilities in logic repair and knowledge compensation. Specifically, on the challenging BigCodeBench-Hard subset, the `DeepSeek-V2-Lite` model achieves an 8.11% absolute performance gain under the guidance of RL-optimized instructions. This significant improvement directly reflects the ability of the proposer agent to identify reasoning blind spots of advanced models when handling specific libraries or deep logical pitfalls, providing precise supplements to the original prompt to steer the generator away from potential erroneous paths. Even for state-of-the-art industrial models, this repair mechanism consistently provides performance gains, underscoring the universal value of the framework in addressing high-difficulty, realistic programming challenges.

Performance trends across optimization stages underscore the decisive role of reinforcement learning in enhancing instruction quality and model alignment. By directly optimizing for downstream correctness, the RL phase ensures that synthesized

tips align with the latent preferences of black-box models. This systematic exploration consistently pushes system performance toward theoretical upper bounds determined by posterior analysis. These results establish instruction-space exploration as a pivotal dimension for test-time scaling, offering an efficient way to unlock model potential without parameter expansion while remaining synergistic with stochastic sampling.

3.3 Analysis of Reranking Performance Bounds

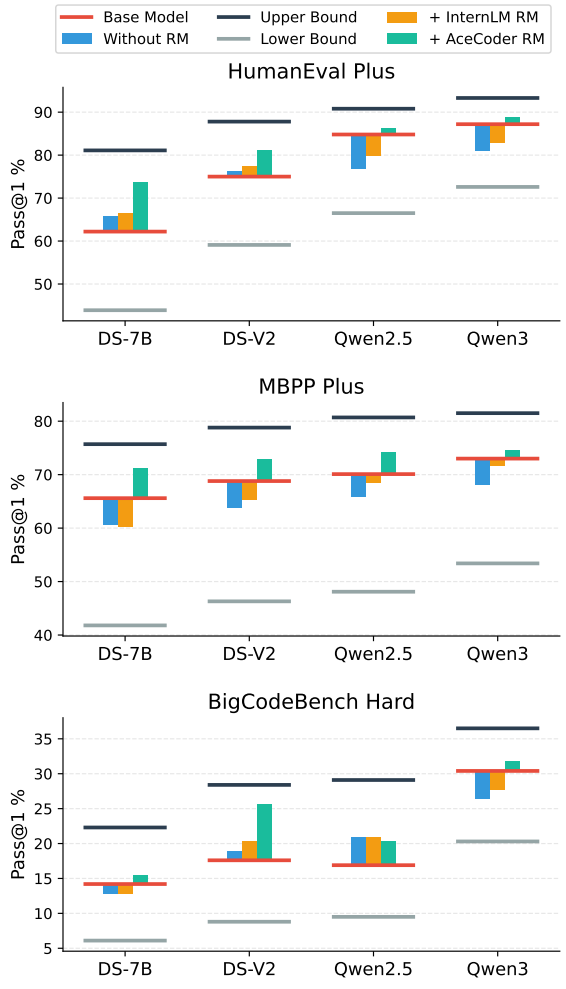

Figure 2: Analysis of performance bounds and ablation of reward models. The potential range is defined by the theoretical upper and lower bounds of instruction-space exploration. We compare the stabilization effects of our domain-optimized AceCoder RM against the general-purpose InternLM-reward-8b, demonstrating the necessity of specialized verification for maximizing the utilization of the available potential range.

To rigorously evaluate the impact of instruction augmentation, we analyze the performance bounds of TIPCODER as visualized in Figure 2. The substantial gap between the baseline and the theoretical upper bounds confirms the immense potential of *Test-time Instruction Proposing*. For instance, on the challenging BigCodeBench-Hard subset, instruction space exploration unlocks a potential gain of approximately 6.1% even for the powerful `Qwen3-Coder-30B`. Crucially, the theoretical upper bound of the lightweight `Qwen2.5-Coder-7B` on HumanEval+ reaches 90.8%, effectively surpassing the 87.2% baseline performance of the significantly larger `Qwen3-Coder-30B`. This suggests that conditioning on high-quality logic tips allows smaller models to achieve reasoning capabilities that rival or even exceed those of much larger models, effectively bridging the performance gap through instruction-level scaling.

Our ablation analysis further reveals the critical role of the reranking mechanism as a stabilizer for instruction injection. As indicated by the lower bounds, blindly adopting generated tips carries a severe risk of performance regression, potentially degrading HumanEval+ performance by up to 18.3% for the `Qwen2.5-Coder-7B`. The *Without RM* configuration highlights this dilemma, where the proposer successfully fixes several incorrect cases but simultaneously breaks previously correct ones due to over-correction or noise. We observe that while a general-purpose reward model like InternLM-reward-8b often fails to align with code-specific logic, even causing performance degradation on models like `Qwen3-Coder-30B`. A stronger domain-optimized RM acts as an effective safety gate. It consistently filters out destructive tips and retains positive guidance, allowing the system to robustly move toward the theoretical upper bound across all benchmarks.

3.4 Analysis of Exploring Efficiency

We benchmark TIPCODER against standard Best-of-N scaling strategies to evaluate exploring efficiency. Since our three-step inference process, which comprises tip generation, code synthesis, and reward reranking, approximates the computational cost of sampling three solutions, Best-of-4 serves as a conservative and fair baseline for comparison. As illustrated in Figure 3, TIPCODER consistently matches or exceeds the performance of higher-cost sampling strategies across all backbones. Specifically, it achieves parity with Best-of-16 sampling on EvalPlus and matches Best-of-4 to Best-of-8 on the more challenging BigCodeBench. These results demonstrate that heuristic exploration

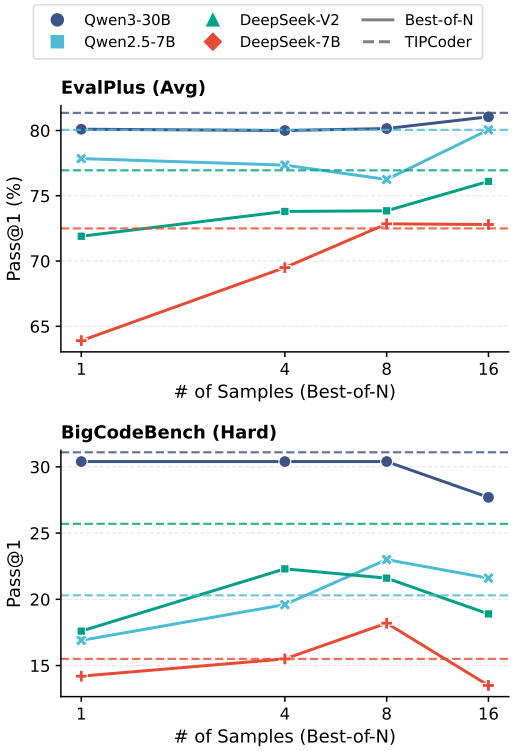

Figure 3: **Efficiency Comparison against Test-time Scaling.** The solid curves represent standard Best-of-N sampling ($N = \{1, 4, 8, 16\}$), while the dashed lines indicate TIPCODER performance.

in the instruction space offers a significantly more sample-efficient scaling dimension than brute-force stochastic exploration in the solution space. Furthermore, as our method fundamentally enhances the quality of single-pass generation by addressing logical blind spots, it remains orthogonal to stochastic sampling and can be seamlessly combined with Best-of-N strategies to reach even higher performance ceilings at larger computational scales.

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 enviroment with multiple coder LLMs, stronger rerank models, andextending this paradigm to general domains.

Limitations

We acknowledge two primary limitations in this work. First, the inference process of TIPCODER is inherently sequential. Unlike parallelizable Best-of-N sampling, this serial dependency introduces unavoidable wall-clock latency, potentially constraining its utility in time-sensitive applications. Second, our Proposer is currently trained on data constructed using a single model scale (7B), which may bias the learned reasoning patterns towards the specific error modes of this architecture. Whether such distilled guidance can universally generalize to significantly larger models or diverse architectures without specific adaptation requires further validation.

Acknowledgments

References

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

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.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. ACECODER: acing coder RL via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 12023–12040. Association for Computational Linguistics.

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 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.

B 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 `Qwen2.5-7B-Instruct` 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 PPO 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-6 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 PPO 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 2 presents the comprehensive list of hyperparameters.

Hyperparameter	**Value**
Common Settings	
Base Model	Qwen2.5-7B-Instruct
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 (PPO)	
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 2: Comprehensive hyperparameter settings for SFT and RL stages.

C 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.

D Case Study

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.

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(l1), len(l2), len(l3))

    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 7: 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.