Noisy Test-Time Reinforcement Learning for Code LLMs

Xikai Yang[1], Hieu Trung Nguyen[1], Dunyuan Xu[1], Yuzhi Zhao[2], Jinpeng Li[1], Wenao Ma[1]*, Pheng-Ann Heng[1]

[1]The Chinese University of Hong Kong, [2]Huazhong University of Science and Technology
yangxk@link.cuhk.edu.hk

Abstract

Large language models (LLMs) have demonstrated remarkable performance across various code-related tasks. However, unlike carefully curated datasets that are typically high-quality and error-free, real-world user instructions are often vague and error-prone, posing significant challenges to the robustness of code LLMs. Furthermore, robustness-oriented fine-tuning relies on paired clean-noisy samples, which are costly to curate and require sophisticated noisy simulation techniques. To address these challenges, we propose the Noisy Test-time Reinforcement Learning framework (NTRL-Code), which enables robust self-evolution of code LLMs using only unlabeled noisy data during the testing stage. Specifically, NTRL-Code uses conservative self-denoising to obtain a cleaner semantic anchor for target estimation, and employs an abstract-syntax-tree (AST)-based structural aggregation mechanism to estimate a proxy target from multiple candidate programs. The policy is then optimized on the original noisy prompts with a hybrid reward that combines format validity, code similarity, and anti-repetition signals. Extensive experiments on three benchmarks, each incorporating character-level, word-level, and paragraph-level perturbations, demonstrate that NTRL-Code yields robust and consistent improvements, stabilizing the predictions of various base models. Our code is available at https://github.com/Xikai97/NTRL-Code.

1 Introduction

Recently, LLMs have become a prominent research focus, driving advancements across various domains, including software engineering (Jin et al., 2024), robotics (Kim et al., 2024), healthcare (Goyal et al., 2024; Shool et al., 2025). Among these applications, code generation stands out as one of the most significant tasks, where LLMs demonstrate the potential to achieve, or even surpass, human-level intelligence. Several widely used LLMs, including Claude series models (Anthropic, 2025), GPT series models (OpenAI, 2025), and Qwen series models (Team, 2025) have proven exceptional performance in generating code snippets given specific textual instructions. However, previous studies have shown that the quality of the generated code of LLMs is highly dependent on the precision of the input instruction prompt (Chen et al., 2026; Paleyes et al., 2026; Zhuo et al., 2025). Even minor changes, such as altering a few words or rephrasing the prompt with equivalent descriptions, can result in a significant drop in performance (As shown in the top of Figure 1). Moreover, real-world users often have varying levels of expertise. For instance, compared with expert, junior users may provide vague or erroneous instructions, posing new challenges to the performance of code LLMs.

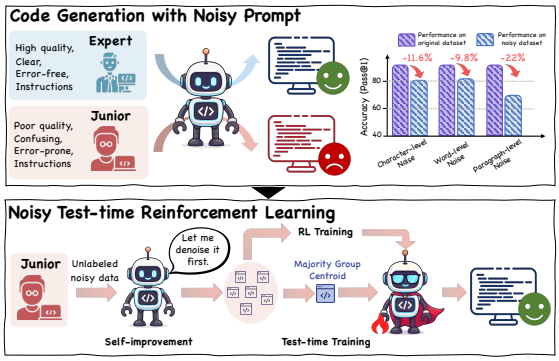

Figure 1: Impact of noisy prompts on Qwen2.5-coder-32B evaluated on HumanEval (top). To address the challenge of noisy inputs, we propose a noisy test-time reinforcement learning framework that trains exclusively on unlabeled noisy data (bottom).

Despite the existence of various approaches aimed at enhancing the robustness of LLMs (Sun et al., 2024; Zhao et al., 2024; Agrawal et al., 2025), a large body of prior robustness works rely on

*Corresponding author.

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing, pages 19656–19677
October 24-29, 2026 ©2026 Association for Computational Linguistics

supervised training with high-quality noisy-clean data pairs, which imposes a substantial burden on dataset preparation (Zhao et al., 2024; Bai et al., 2025) and often leads to overfitting on specific noise patterns. Another category of approaches focuses on aligning features between clean and noisy samples and calibrating hidden layer embeddings (Sun et al., 2024; Liu et al., 2025b,a). Yet, directly modifying the parameters of specific hidden layers risks disrupting the reasoning process of LLMs, potentially leading to degraded performance. Other works propose to incorporate self-denoising (Agrawal et al., 2025) in the framework and leverage the model's inherent denoising capabilities through prompt engineering to iteratively refine noisy inputs. The main concern with this method is that it fails to fundamentally improve the model's robustness. Moreover, the iterative denoising process significantly increases inference time. Therefore, a label-efficient approach to enhance the performance of Code LLMs against various types of perturbations is highly desired.

Two major classes of methods focus on such model evolution without supervision: (1) entropy-based methods, such as Tent (Wang et al., 2020), EMPO (Zhang et al., 2026), and RENT (Prabhudesai et al., 2025); (2) self-rewarding methods, like Test-Time Reinforcement Learning (TTRL)(Zuo et al., 2026), TTRV(Singh et al., 2026), etc. However, both types of methods lack specific designs to handle perturbations within the prompt. Noisy inputs can negatively affect entropy variation and degrade the quality of self-reward mechanisms. Moreover, compared with the mathematical datasets typically used in these methods, coding problems present additional complexity in identifying an appropriate reward for self-supervision, particularly for approaches relying on majority voting. For instance, two code snippets with different variable names or minor formatting differences but identical execution logic can still be grouped together. This makes identifying the most frequent output, as commonly applied in mathematical problems (Zuo et al., 2026), unsuitable for coding tasks.

Therefore, in this work, we propose **NTRL-Code**, a novel test-time training framework designed to steadily improve code LLMs' performance using only unlabeled, noise-perturbed test samples. Specifically, NTRL-Code incorporates a self-improvement module that performs a self-denoising call before each rollout to mitigate the impact of perturbed prompts. Subsequently, a dual-rollout is conducted on both the original noisy prompt and denoised prompt to decouple reward calculation from proxy-target estimation. Additionally, NTRL-Code leverages the Abstract Syntax Tree (AST) to parse each code snippet and employs a majority voting mechanism to generate pseudo-labels. Finally, we design a hybrid reward system that combines format reward, code similarity reward, and no-repetition reward, enabling the policy model to be updated in a code-friendly manner. Our contributions are summarized as follows:

1. We formulate noisy test-time reinforcement learning for code generation, where a code LLM adapts from unlabeled noisy prompts without clean-noisy pairs or executable feedback.
2. We propose **NTRL-Code**, a decoupled denoising-optimization framework that estimates proxy targets from a denoised branch while optimizing the policy on the original noisy branch.
3. We introduce structure-aware proxy target estimation based on AST canonicalization and clustering, together with a hybrid reward for stable test-time RL.
4. Experiments on three benchmarks under diverse perturbations show consistent gains across multiple base models, validating the effectiveness of our design.

2 Related Work

2.1 LLM Robustness

With the remarkable performance achieved by LLMs across various fields, recent studies have begun to explore the impact of input perturbations on generated outputs (Nookala et al., 2023; Rauba et al., 2024; Chaudhary et al., 2024; Alahmari et al., 2025; Wang et al., 2025). Given that diverse noise is likely to arise in real-world deployments, it is crucial to evaluate the robustness of LLMs and develop reliable AI systems to ensure resilience against such perturbations. Zhu et al. (2024) systematically built a library for LLM evaluation, which also supports research on adversarial prompt attacks at various levels, such as character-level and semantic-level attacks. Zhu et al. (2023) employed adversarial textual attacks to generate noisy prompts for LLM evaluation and demonstrated that LLMs are not robust to such adversarial prompts. Chatterjee et al. (2024) developed a Prompt Sensitivity Index to measure the sensitivity of LLMs to variations in prompts. To enhance the robustness of LLMs, Agrawal et al. (2025) compared several prevailing

approaches and found that iterative self-denoising is effective in improving generation quality when dealing with noisy prompts. Other approaches, such as hidden state alignment (Zhao et al., 2024), adversarial training (Bukharin et al., 2025), and prompt optimization (Wu et al., 2026), have also been explored in recent works to mitigate the impact of perturbations in instructions.

2.2 AI coding Agent

The advancement of AI coding agents has brought a revolution to programming users. Nearly all successful large models claim superior performance in solving coding problems (Guo et al., 2024; Hui et al., 2024; Team, 2025; OpenAI, 2025). Since coding tasks are highly user-interactive and impose rigorous demands on the output, evaluating the robustness of coding agents has gradually emerged as a critical research direction (Yang et al., 2024; Liu et al., 2025c; Lin et al., 2025; Chen et al., 2025). However, few studies have focused on enhancing the robustness of coding agents. Hossen and Hei (2024) leveraged GPT models to generate adversarial prompts and constructed corresponding instruction-tuned datasets for coding agent training. Liu et al. (2025b) proposed a model editing approach that updated the most unstable layers during supervised training on paired original and perturbed prompts.

3 Methodology

3.1 Overview of NTRL-Code Framework

To address the performance degradation in code generation caused by noisy instructions, we propose the NTRL-Code framework to improve model's performance by using only unlabeled noisy data. Figure 2 provides an overview of the NTRL-Code framework. Specifically, we first utilize the inherent denoising capability of the current policy model to rectify input noisy prompt. Afterward, two rollouts are performed separately: one based on the original noisy prompt and the other on the denoised prompt. Next, we perform AST-based structural aggregation on the denoised rollouts and select the representative program from the largest structural cluster as a proxy target, which is a label-free estimate of the most stable program structure under the denoised semantic intent. Finally, we update the policy using GRPO on candidate programs generated from the original noisy prompt. The reward compares noisy-branch candidates against the proxy target through a hybrid reward function.

3.2 Prompt Correction via Self-denoising

Unlike previous work (Agrawal et al., 2025), which employed self-denoising to iteratively refine noisy prompts and improve inference results, NTRL-Code utilizes self-denoising solely to enhance the reliability of proxy target estimation from a group of candidates. This design ensures that self-denoising in NTRL-Code does not interfere with the actual inference stage. Furthermore, it directly contributes to model evolution during the test-time training phase, enhancing the model's inherent robustness.

Figure 6 in the Appendix illustrates the prompt used to call the current policy model $\pi_\theta(\cdot|x)$ to denoise input perturbed instructions. More concretely, during the denoising process, we temporarily modify the system prompt, transforming its role from a code generator to an instruction denoiser. However, for certain models with limited instruction-following capabilities, such as DeepSeek-1.3B-Coder, it is challenging for them to accurately identify errors and effectively correct input prompts. More critically, they may generate excessive redundant content, leading to distribution shifts during the RL update process. To mitigate such adverse effects, we apply a conservative recheck mechanism before using the denoised prompt for target estimation. Instead of relying on a semantic similarity model, we use a lightweight length-change ratio (r_{change}) to detect overly aggressive rewrites.

$$r_{\text{change}} = \frac{|\text{len}(p) - \text{len}(p_{\text{denoise}})|}{\max(\text{len}(p), 1)}. \quad (1)$$

We set the failure threshold to $\tau_{\text{len}} = 0.5$. If $r_{\text{change}} > \tau_{\text{len}}$, the denoising result is considered unreliable and we revert to the original noisy prompt as shown in Eq. (2), where $D(\cdot, \phi)$ denotes the self-denoising operator with ϕ as the denoising prompt. $p, p_{\text{denoise}}, \tilde{p}$ denote the original noisy prompt, the denoised prompt, and the final output prompt after recheck, respectively.

$$p_{\text{denoise}} = D(p, \phi),\ \tilde{p} = \begin{cases} p, \text{if } r_{\text{change}} > \tau_{\text{len}}, \\ p_{\text{denoise}}, \text{otherwise.} \end{cases} \quad (2)$$

This rule is designed to filter out destructive denoising cases, such as over-deletion of constraints, over-expansion with hallucinated requirements, or large semantic drift caused by excessive rewriting.

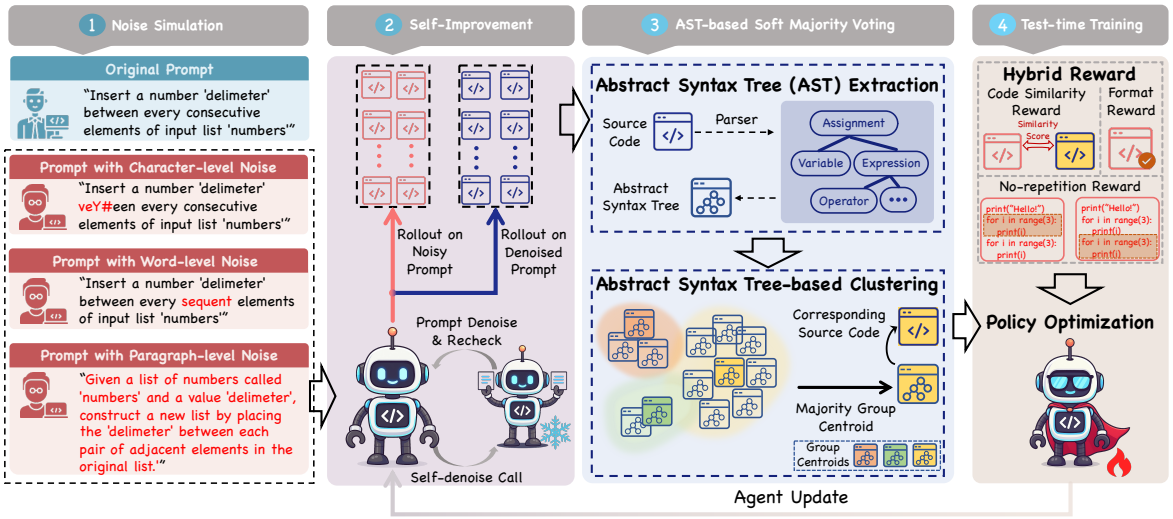

Figure 2: (1) Examples of three types of simulated perturbation and (2-4) an overview of the proposed NTRL-Code framework. The framework consists of three key modules: (2) a self-improvement module for initial prompt correction to improve proxy-target estimation; (3) an Abstract Syntax Tree–based majority voting mechanism to identify the centroid of the largest group for reward calculation; and (4) policy optimization using a hybrid reward signal to enhance the model with unlabeled data.

We then perform two separate rollouts: noisy rollouts conditioned on the original prompt p, and target-estimation rollouts conditioned on the rechecked prompt $\tilde{p}$. This decoupling ensures that policy optimization is performed on the noisy input distribution, while the proxy target is estimated from a cleaner and more stable prompt distribution.

3.3 AST-based Structural Aggregation

To construct a label-free reward signal for code generation, we estimate a proxy target from multiple programs sampled from the rechecked denoised prompt $\tilde{p}$. Since programs with different variable names, comments, or formatting may implement the same structure, raw textual majority voting is unsuitable for code. We therefore perform structure-aware aggregation based on Python ASTs. For each candidate program $\tilde{y}_i$ generated from rechecked prompt $\tilde{p}$, we first extract the code block and attempt to parse it into a Python AST. If parsing succeeds, we canonicalize the tree by replacing user-defined variable names, function names, and argument names with deterministic placeholders. This normalization reduces spurious differences caused by renaming. We then serialize the canonicalized tree using ast.dump() function (excluding attributes and annotations) and compute an MD5 hash of the resulting string as its structural fingerprint. Candidates with the same structural fingerprint are assigned to the same cluster. Some generated programs may be syntactically incomplete and therefore cannot be parsed into an AST. For these cases, we use a lexical fallback. Specifically, we normalize the code by removing comments, condensing whitespace, and lowercasing text, then compute token-level Jaccard similarity. Two unparsable candidates are assigned to the same cluster if their Jaccard similarity is at least $\epsilon = 0.92$. We selected ϵ by a small parameter sweep over candidate thresholds and used the value that achieved the best Pass@1. After clustering, we select the largest cluster and use its representative candidate as the proxy target $\tilde{y}^*$. Algorithm 1 summarizes the procedure, where $\mathrm{ExtractCode}$, AST, and Canon denote the extraction, parsing, and canonicalization steps described above.

3.4 Test-time RL with Hybrid Rewards

Similar to a standard RL training framework, test-time RL also updates the policy model based on reward signals. These reward signals are computed by comparing proxy target $\tilde{y}^*$ with the candidate rollout responses $\{y_1, y_2, \cdots, y_N\}$ generated from the original noisy prompt by the current policy model. In particular, we design a hybrid reward including three parts to reduce the risk of optimizing toward a single brittle signal.

Format reward. We determine whether a non-empty code block can be successfully extracted from the response; if successful, $R_{format} = 1$, otherwise $R_{format} = 0$.

Code Similarity Reward. Let the extracted code

Algorithm 1 AST-based Structural Aggregation

Input: Candidate set $\mathcal{Y} = \{\tilde{y}_1, \tilde{y}_2, \cdots, \tilde{y}_N\}$; lexical threshold ϵ
Output: Proxy target $\tilde{y}^*$

```
C ← ∅; M ← ∅
for i ← 1 to N do
    c_i ← ExtractCode(ỹ_i)
    if c_i is parsable then
        h_i ← MD5(Canon(AST(c_i)))
        k ← the cluster with key h_k = h_i
    else
        T_i ← Tok(Nor(c_i))
        k ← first lexical cluster with Jac(T_i, T_k) ≥ ε
    end if
    if k exists then
        C_k ← C_k ∪ {i}
    else
        C_new ← {i}
        C ← C ∪ {C_new};
    end if
end for
k_max ← arg max_k |C_k|
i* ← index of the first candidate assigned to the largest cluster C_{k_max}
ỹ* ← ỹ_{i*}
return ỹ*
```

for the candidate sample y_i and the proxy target $\tilde{y}^*$ be denoted as C_i and C_{gt}, respectively. The similarity score is computed as the sum of the normalized Levenshtein distance (Levenshtein, 1966) and the Jaccard similarity (Kosub, 2019). Specifically, let $Nor(\cdot)$ represent the normalization process, which removes comments, condenses spaces, and converts text to lowercase. The normalized code can then be expressed as $C'_i = Nor(C_i)$ and $C'_{gt} = Nor(C_{gt})$. The normalized Levenshtein similarity reward is computed as shown in Eq. (3). Similarly, let $Tok(\cdot)$ denote the tokenization operation. Using this, $T_i = \mathrm{Tok}(C_i)$ and $T_{gt} = \mathrm{Tok}(C_{gt})$ represent the token sets. The Jaccard similarity reward is then calculated as shown in Eq. (4). The final fused similarity reward is computed as Eq. (5).

$$r_{lev} = 1 - \min\left(1, \frac{Lev(C'_i, C'_{gt})}{\max(1, |C'_i|, |C'_{gt}|)}\right). \quad (3)$$

$$r_{jac} = \frac{|T_i \cap T_{gt}|}{\max\{1,\ |T_i \cup T_{gt}|\}}. \quad (4)$$

$$R_{sim} = 0.5 \cdot r_{lev} + 0.5 \cdot r_{jac}. \quad (5)$$

No-repetition Reward. We utilize sliding windows to measure the extent of repetition within the generated code snippet. Let $P' = Nor(P)$ represent the processed input prompt. The window length W and step size S are defined as shown in Eq. (6).

$$W = \max\Big(10,\ \min\big(80,\ \lfloor |P'|/10 \rfloor + 10\big)\Big), \quad S = \max\Big(5,\ \lfloor W/2 \rfloor\Big). \quad (6)$$

We then define the number of overlaps M by applying sliding windows of size W over P' with stride S. The normalized overlap ratio M' is computed as shown in Eq. (7). The reward for no-repetition is defined as: $R_{rep} = 1 - M'$.

$$M' = \frac{M}{1 + \lfloor |P'|/S \rfloor}. \quad (7)$$

Hybrid Reward. The final hybrid reward is given as in Eq. (8), where we let $(\alpha, \beta) = (0.5, 0.25)$. Specifically, format reward filters structurally invalid responses, similarity reward aligns noisy-branch candidates with the proxy target, and no-repetition reward penalizes degenerate repetitions that frequently appear under noisy prompts. The complete update is driven by both code validity and structural consistency rather than raw textual matching alone.

$$R_{total} = \alpha R_{sim} + \beta R_{format} + (1 - \alpha - \beta) R_{rep}. \quad (8)$$

3.5 Why Decoupled Test-Time RL Improves Robustness

NTRL-Code differs from generic test-time self-improvement in the source of its reward anchor. In standard TTRL, both the candidate rollouts and the pseudo-target are obtained from the same input distribution, which makes the reward signal itself noise-dependent under noisy prompts. If the perturbation changes the model's interpretation, majority voting may reinforce a perturbed or spurious pattern. NTRL-Code instead optimizes the policy under a decoupled objective. Let p and $\tilde{p}$ denote the original noisy prompt and the rechecked denoised prompt, respectively. The proxy target is estimated from denoised rollouts,

$$\tilde{y}^* = A(\{\tilde{y}_i\}_{i=1}^N), \quad \tilde{y}_i \sim \pi_\theta(\cdot|\tilde{p}), \quad (9)$$

where $A(\cdot)$ denotes AST-based structural aggregation. The policy update is then performed on noisy-branch candidates ($y_i \sim \pi_\theta(\cdot|p)$) using rewards

$R(y_i, \tilde{y}^*)$. This yields the adaptation objective as shown in Eq. (10).

$$\max_{\theta} \mathbb{E}_{p\sim\mathcal{D}}\mathbb{E}_{y\sim\pi_\theta(\cdot|p)}\left[R(y, A(\pi_\theta(\cdot|\tilde{p})))\right]. \quad (10)$$

This objective encourages outputs conditioned on noisy prompts to match a semantic anchor estimated from a cleaner prompt distribution. Therefore, the update pushes the model toward noise-invariant mappings rather than simply increasing confidence on the noisy input itself.

This design also explains why NTRL-Code is less likely to overfit to a specific perturbation pattern. The optimization target is not derived from the noisy text alone. It is anchored by the denoised branch and aggregated in a structure-aware code space. Moreover, the small number of update steps constrains the magnitude of adaptation. The cross-perturbation and cross-dataset results in Section 4.7 further suggest that the learned update generalizes beyond the perturbation type used during adaptation.

4 Experiments

We conduct extensive experiments on three datasets and their perturbed variants (described in Appendix C.1) to evaluate the performance of our framework. In the following sections, we first present a detailed explanation of the general experimental setup used in our study, followed by the main results of our method in comparison to three label-free approaches. Finally, we demonstrate the effectiveness of our framework's design through a series of ablation studies and evaluate the performance of NTRL-Code on the noisy dataset with mixed-type perturbations, as well as in cross-dataset and cross-perturbation settings.

4.1 Experiment Setup

We leverage GRPO (Shao et al., 2024) to estimate the advantage function in RL. The training process uses a batch size of 8 and a learning rate of 5×10^{-7}. We conduct RL update over 100 steps, with 64 rollouts sampled per prompt to estimate proxy target and 32 rollouts for advantage computation. To evaluate the performance of the proposed NTRL-Code framework, we use the standard Pass@1 metric, which measures the proportion of test samples successfully solved on the first attempt, closely reflecting typical user behavior (Liu et al., 2025b). We impose three levels of perturbation to prompts: character-level, word-level, and paragraph-level. These perturbations cover common surface-form and paraphrastic variations. Figure 2-1 illustrates an example of our simulated perturbations. Details of the implementation of perturbation simulation along with additional examples can be found in the Appendix B.1. We apply three types of noise to three datasets (LeetCode-Bench (Guo et al., 2024), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021)), resulting in nine noisy benchmarks to evaluate the robustness of LLMs in code generation tasks.

4.2 Main Results

We evaluate the NTRL-Code approach on three base models: DeepSeek-Coder-1.3B, DeepSeek-Coder-6.7B, and Qwen2.5-Coder-32B. For each base model, we compare the performance of NTRL-Code against three label-free denoising methods: Few-shot denoising (Zhang et al., 2023), denoising using a large model (Team, 2025), and iterative self-denoising (Agrawal et al., 2025). Specifically, few-shot denoising leverages a small set of clean-noisy examples to effectively guide the model denoise input prompt and then do inference on the denoised prompt. The specific prompt for few-shot denoising is given in the Appendix D. For denoising using a large model, we employ *Qwen3-Coder-480B-A35B-Instruct* as an external powerful denoiser with a carefully designed prompt (shown in Appendix D) to modify noisy prompt. For iterative self-denoising, we adopt the test-time scaling strategy, where self-denoising is performed iteratively until the content remains unchanged or the maximum number of iterations is reached.

Table 1 presents the corresponding main results. Our proposed NTRL-Code framework consistently improves performance across all three datasets and all perturbation levels relative to the original models. Averaged over the nine settings per model, the performance gains are 4.0%, 3.5%, and 3.0% for DeepSeek-Coder-1.3B, DeepSeek-Coder-6.7B, and Qwen2.5-Coder-32B, respectively. Averaged by perturbation type, NTRL-Code yields gains of 3.7%, 3.5%, 3.4% under character, word, and paragraph-level perturbations, respectively. In contrast, Few-shot denoising exhibits highly unstable behavior across datasets and perturbation types. For example, it improves HumanEval under paragraph-level perturbation by 3.7% with the Qwen2.5-Coder-32B base model but causes severe degradation on MBPP dataset. The Big-Denoise approach delivers improvements in some noisy set-

Base Model	Methods	HumanEval			LeetCode Bench			MBPP		
		Char-Perb	*Word-Perb*	*Para-Perb*	*Char-Perb*	*Word-Perb*	*Para-Perb*	*Char-Perb*	*Word-Perb*	*Para-Perb*
DeepSeek-Coder-1.3B	Original	53.0	54.3	56.1	3.3	5.0	5.0	44.6	43.8	46.6
	Few-shot	$49.4_{-3.6}$	$56.1_{+1.8}$	$56.1_{+0.0}$	$3.3_{+0.0}$	$\mathbf{6.7}_{+1.7}$	$5.0_{+0.0}$	$48.2_{+3.6}$	$46.0_{+2.2}$	$47.8_{+1.2}$
	Big-Denoise	$54.3_{+1.3}$	$50.0_{-4.3}$	$37.2_{-18.9}$	$7.8_{+4.5}$	$5.6_{+0.6}$	$4.4_{-0.6}$	$48.4_{+3.8}$	$44.0_{+0.2}$	$0.2_{-46.4}$
	Iter-SD	$51.8_{-1.2}$	$52.4_{-1.9}$	$56.1_{+0.0}$	$3.3_{+0.0}$	$6.1_{+1.1}$	$5.0_{+0.0}$	$48.6_{+4.0}$	$47.0_{+3.2}$	$48.6_{+2.0}$
	NTRL-Code	$\mathbf{55.5}_{+2.5}$	$\mathbf{59.1}_{+4.8}$	$\mathbf{62.2}_{+6.1}$	$\mathbf{7.2}_{+3.9}$	$\mathbf{6.7}_{+1.7}$	$\mathbf{8.9}_{+3.9}$	$\mathbf{49.2}_{+4.6}$	$\mathbf{48.6}_{+4.8}$	$\mathbf{50.6}_{+4.0}$
DeepSeek-Coder-6.7B	Original	62.8	64.0	66.5	16.1	15.0	16.1	60.0	57.8	61.6
	Few-shot	$58.5_{-4.3}$	$64.0_{+0.0}$	$59.1_{-7.4}$	$17.8_{+1.7}$	$15.0_{+0.0}$	$18.9_{+2.8}$	$61.4_{+1.4}$	$58.4_{+0.6}$	$62.8_{+1.2}$
	Big-Denoise	$67.1_{+4.3}$	$63.4_{-0.6}$	$49.4_{-17.1}$	$16.1_{+0.0}$	$\mathbf{18.3}_{+3.3}$	$16.1_{+0.0}$	$60.4_{+0.4}$	$58.3_{+0.5}$	$54.0_{-7.6}$
	Iter-SD	$67.1_{+4.3}$	$62.8_{-1.2}$	$65.2_{-1.3}$	$17.2_{+1.1}$	$17.2_{+2.2}$	$17.8_{+1.7}$	$\mathbf{62.8}_{+2.8}$	$51.6_{-6.2}$	$62.0_{+0.4}$
	NTRL-Code	$\mathbf{68.3}_{+5.5}$	$\mathbf{67.1}_{+3.1}$	$\mathbf{72.0}_{+5.5}$	$\mathbf{18.3}_{+2.2}$	$\mathbf{18.3}_{+3.3}$	$\mathbf{20.6}_{+4.5}$	$62.6_{+2.6}$	$\mathbf{60.6}_{+2.8}$	$\mathbf{63.8}_{+2.2}$
Qwen2.5-Coder-32B	Original	79.9	81.7	69.5	63.9	63.3	67.2	80.0	79.2	78.2
	Few-shot	$80.5_{+0.6}$	$81.1_{-0.6}$	$\mathbf{73.2}_{+3.7}$	$62.2_{-1.7}$	$66.7_{+3.4}$	$66.7_{-0.5}$	$19.2_{-60.8}$	$64.8_{-14.4}$	$50.6_{-27.6}$
	Big-Denoise	$74.4_{-5.5}$	$70.7_{-11.0}$	$56.1_{-13.4}$	$67.2_{+3.3}$	$68.3_{+5.0}$	$66.1_{-1.1}$	$75.2_{-4.8}$	$73.6_{-5.6}$	$69.6_{-8.6}$
	Iter-SD	$78.0_{-1.9}$	$79.3_{-2.4}$	$65.9_{-3.6}$	$64.4_{+0.5}$	$66.7_{+3.4}$	$60.6_{-6.6}$	$80.2_{+0.2}$	$79.6_{+0.4}$	$78.4_{+0.2}$
	NTRL-Code	$\mathbf{81.7}_{+1.8}$	$\mathbf{82.9}_{+1.2}$	$72.0_{+2.5}$	$\mathbf{71.1}_{+7.2}$	$\mathbf{71.7}_{+8.4}$	$\mathbf{68.9}_{+1.7}$	$\mathbf{82.6}_{+2.6}$	$\mathbf{80.6}_{+1.4}$	$\mathbf{78.8}_{+0.6}$

Table 1: Method comparison of our proposed NTRL-Code and three additional label-free denoising approaches across three base models (DeepSeek-Coder-1.3B, 6.7B, and Qwen2.5-coder-32B). Few-shot, Big-Denoise, and Iter-SD denotes the Few-shot denoising, denoising using a large model, and iterative self-denoising, respectively. Pass@1 accuracy is reported for three types of perturbations across three benchmarks.

tings but is particularly vulnerable to paragraph-level perturbations, resulting in an average drop of 12.6%. Iterative self-denoising is relatively more robust than the other baselines, yet its overall performance remains inferior to NTRL-Code. These results reveals that NTRL-Code exhibits strong robustness in enhancing model performance against perturbations using only unlabeled noisy samples and a few RL updates.

4.3 Self-Denoising Ablation

We further investigate the effectiveness of the self-denoising component by removing the self-denoising call and the rollout on the denoised prompt. We evaluate NTRL-Code on the DeepSeek-Coder-1.3B model across all three datasets, and the results are summarized in Table 2. After ablating the self-denoising component, performance consistently drops (in some cases even falls below the original model). The average performance reductions at the character, word, and paragraph-level perturbations are 2.5%, 3.1%, and 2.1%, respectively. These findings suggest that self-denoising is particularly effective at correcting lower-level errors (e.g., spelling and tokenization issues) and provides stable performance gains. However, addressing content-level or semantic variations may require additional training to further enhance reasoning capabilities. This ablation supports the decoupling hypothesis in Section 3.5.

Methods	HumanEval		
	Char-Perb	*Word-Perb*	*Para-Perb*
Original	53.0	54.3	56.1
NTRL-Code w/o SD	$51.2_{-1.8}$	$55.5_{+1.2}$	$59.1_{+3.0}$
NTRL-Code w/ SD	$55.5_{+2.5}$	$59.1_{+4.8}$	$62.2_{+6.1}$
Methods	**LeetCode Bench**		
	Char-Perb	*Word-Perb*	*Para-Perb*
Original	3.3	5.0	5.0
NTRL-Code w/o SD	$5.6_{+2.3}$	$4.4_{-0.6}$	$8.3_{+3.3}$
NTRL-Code w/ SD	$7.2_{+3.9}$	$6.7_{+1.7}$	$8.9_{+3.9}$
Methods	**MBPP**		
	Char-Perb	*Word-Perb*	*Para-Perb*
Original	44.6	43.8	46.6
NTRL-Code w/o SD	$47.6_{+3.0}$	$45.2_{+1.4}$	$47.8_{+1.2}$
NTRL-Code w/ SD	$49.2_{+4.6}$	$48.6_{+4.8}$	$50.6_{+4.0}$

Table 2: Self-denoising ablation study for the DeepSeek-Coder-1.3B model. SD refers to self-denoising.

When the denoised branch is removed, the proxy target is estimated under the same noisy distribution used for optimization, making the reward signal less stable.

4.4 Rewards Ablation

Here, we perform a detailed ablation study on the hybrid reward. Table 3 reports results for seven dis-

tinct reward configurations on HumanEval with the DeepSeek-Coder-1.3B base model. For each configuration, we also report the performance change relative to the full hybrid reward. Relying on a single reward component (①, ②, ③) leads to clear degradations under three perturbation types. With two reward component variants (④, ⑤, ⑥), we observe that paragraph-level perturbations are particularly sensitive to the missing of similarity reward and yields a 5.5% drop, indicating that similarity reward is crucial to handle semantic variations. Overall, the best performance is achieved when all three reward components (Format, No-repetition, Code similarity) are enabled, underscoring the importance of the hybrid reward design.

	Format	Rep	Sim	*Char-Perb*	*Word-Perb*	*Para-Perb*
①	✓			$52.4_{-3.1}$	$53.7_{-5.4}$	$57.9_{-4.3}$
②		✓		$51.8_{-3.7}$	$55.5_{-3.6}$	$57.3_{-4.9}$
③			✓	$50.6_{-4.9}$	$55.5_{-3.6}$	$59.1_{-3.1}$
④	✓	✓		$53.0_{-2.5}$	$56.7_{-2.4}$	$56.7_{-5.5}$
⑤	✓		✓	$51.8_{-3.7}$	$56.1_{-3.0}$	$59.1_{-3.1}$
⑥		✓	✓	$51.8_{-3.7}$	$56.1_{-3.0}$	$59.7_{-2.5}$
⑦	✓	✓	✓	55.5	59.1	62.2

Table 3: Reward ablation study on the HumanEval dataset with the base model DeepSeek-Coder-1.3B.

4.5 Performance Evolution on RL Steps

Figure 3 illustrates the performance evolution of NTRL-Code steps ranging from 0 to 100, using the DeepSeek-Coder-1.3B model on the HumanEval dataset with three types of perturbations. The results clearly demonstrate that Pass@1 score steadily increases, indicating continuous improvement and refinement in the model's performance. Furthermore, NTRL-Code achieves the most significant performance gain under paragraph-level perturbations, while the improvement under character-level perturbations is relatively minor. This suggests that fine-grained noise is more sensitive to LLM generation, introducing greater challenges during the test-time training process.

4.6 Results on Mixed-Type Perturbations

To evaluate whether NTRL-Code is limited to isolated perturbation types, we further test mixed-type perturbations, where multiple perturbation operators are composed within the same prompt. This setting better approximates practical user inputs, in which typos, word substitutions, and paraphrastic variations may co-occur. Table 4 presents the results on the HumanEval dataset using DeepSeek-Coder-1.3B with four types of mixed perturbations. Our findings reveal that the performance of the base model deteriorates as the perturbations become more complex. Nevertheless, NTRL-Code consistently enhances the base model's performance, even achieving the largest improvement (6%) when all three types of perturbations are added.

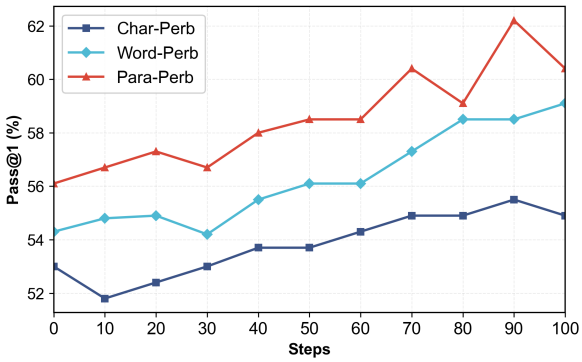

Figure 3: Evolution of the number of steps for NTRL-Code using DeepSeek-Coder-1.3B on the HumanEval.

Perturbation Type	Original	NTRL-Code
Word+Para	48.1	$53.0_{+4.9}$
Char+Para	51.2	$54.3_{+3.1}$
Char+Word	47.5	$50.0_{+2.5}$
Char+Word+Para	47.0	$53.0_{+6.0}$

Table 4: Performance of the NTRL-Code with mixed-type perturbations.

4.7 Generalization Analysis

To evaluate the generalization ability of our NTRL-Code framework, we test the performance of the trained model on mismatched datasets or perturbations. Specifically, our experiments consist of two parts:

1. *Cross-Dataset Validation*: We fine-tune DeepSeek-Coder-1.3B on the HumanEval dataset using the NTRL-Code framework and test its performance on the LeetCode Bench.

2. *Cross-Perturbation Validation*: We train the model on one type of noise and validate its performance on a different type of noise.

The corresponding experimental results are summarized in Table 5. For cross-dataset validation,

the results demonstrate that NTRL-Code improves performance across all types of perturbations, indicating strong generalization to previously unseen datasets. Additionally, in the cross-perturbation scenario, NTRL-Code still provides improvements, further highlighting its robust generalization ability even when the model encounters mismatched noise types. These results suggest that NTRL-Code does not merely memorize the perturbation pattern seen during adaptation. Instead, the decoupled objective encourages mappings that transfer across perturbation types and datasets.

Cross-Dataset (HumanEval $\longrightarrow$ LeetCode)		
Perturbation Type	Original	NTRL-Code
Char-Perb	3.3	$5.0_{+1.7}$
Word-Perb	5.0	$6.1_{+1.1}$
Para-Perb	5.0	$6.7_{+1.7}$
Cross-Perturbation (HumanEval)		
Para→Char	53.0	$53.7_{+0.7}$
Para→Word	54.3	$56.1_{+1.8}$

Table 5: Generalization analysis of the NTRL-Code (the base model is DeepSeek-Coder-1.3B).

5 Conclusion

In this work, we present NTRL-Code, a label-free test-time reinforcement learning framework for improving code LLM robustness under noisy prompts. NTRL-Code decouples proxy-target estimation from policy optimization by estimating structure-aware targets from rechecked denoised rollouts while updating the model on the original noisy prompts with a hybrid reward. Experiments across three benchmarks, multiple perturbation types, and several base models show consistent robustness gains, while ablations support the roles of self-denoising, AST-based aggregation, and the hybrid reward.

Limitations

The limitations of this work can be summarized as follows.

Noise scope. This work focuses on character-level, word-level, paragraph-level, and mixed perturbations. These perturbations cover common surface-form and paraphrastic variations, but they do not exhaust all realistic user-side noise. For example, real coding requests may contain missing input/output specifications, logically inconsistent requirements, multi-task instructions, or ambiguous constraints. Handling such requirement-level noise may require clarification mechanisms or uncertainty-aware refusal, which are beyond the scope of this work.

Proxy-target reliability. NTRL-Code estimates proxy targets from model-generated candidates without using executable tests during optimization. Therefore, the selected AST-majority target is not guaranteed to be correct. Failure can occur when the model samples share a common structural bug, when the denoising step changes an important requirement, or when multiple correct programs have different AST structures and are split across clusters. Our design mitigates this through conservative denoising recheck, AST canonicalization, and hybrid rewards, but execution-based verification could further improve reliability.

Computational cost. NTRL-Code requires multiple rollouts and GRPO updates during adaptation, making it more expensive than standard single-pass inference. We therefore view it as an asynchronous adaptation method: unlabeled noisy prompts can be batched to update a shadow policy in the background, while the active model serves user requests normally. This setting reduces user-facing latency but still requires additional compute resources.

Closed-source models. Our experiments focus on open-source code LLMs because NTRL-Code requires access to model parameters for test-time policy updates. Closed-source APIs can be evaluated for noise sensitivity, but the proposed parameter-update framework cannot be directly applied to them. Developing black-box or training-free variants for closed-source models is an important future direction.

Acknowledgements

The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China, under Project T45-401/22-N. AI assistants were used to support the implementation of noise simulation procedures. All experimental designs, analyses, and conclusions were conducted and verified by the authors.

References

Aryan Agrawal, Lisa Alazraki, Shahin Honarvar, and Marek Rei. 2025. Enhancing llm robustness to

perturbed instructions: An empirical study. *arXiv preprint arXiv:2504.02733*.

Saeed S. Alahmari, Lawrence Hall, Peter R. Mouton, and Dmitry Goldgof. 2025. Large language models robustness against perturbation. *Scientific Reports*, 16(1):346.

Anthropic. 2025. Claude sonnet 4.5. `https://www.anthropic.com/claude/sonnet`.

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

Shengyuan Bai, Qibin Li, Zhe Wang, Nai Zhou, and Nianmin Yao. 2025. Enhancing nlu in large language models using adversarial noisy instruction tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23451–23459.

Alexander Bukharin, Haifeng Qian, Shengyang Sun, Adithya Renduchintala, Soumye Singhal, Zhilin Wang, Oleksii Kuchaiev, Olivier Delalleau, and Tuo Zhao. 2025. Adversarial training of reward models. *arXiv preprint arXiv:2504.06141*.

Anwoy Chatterjee, HSVNS Kowndinya Renduchintala, Sumit Bhatia, and Tanmoy Chakraborty. 2024. Posix: A prompt sensitivity index for large language models. In *Findings of the association for computational linguistics: EMNLP 2024*, pages 14550–14565.

Manav Chaudhary, Harshit Gupta, Savita Bhat, and Vasudeva Varma. 2024. Towards understanding the robustness of llm-based evaluations under perturbations. In *Proceedings of the 21st International Conference on Natural Language Processing (ICON)*, pages 197–205.

Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. 2026. Nlperturbator: Studying the robustness of code llms to natural language variations. *ACM Transactions on Software Engineering and Methodology*, 35(4):1–20.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Simin Chen, Pranav Pusarla, and Baishakhi Ray. 2025. Dynamic benchmarking of reasoning capabilities in code large language models under data contamination. *arXiv preprint arXiv:2503.04149*.

Samarth Goel, Reagan J Lee, and Kannan Ramchandran. 2025. Sage: A realistic benchmark for semantic understanding. *arXiv preprint arXiv:2509.21310*.

Sagar Goyal, Eti Rastogi, Sree Prasanna Rajagopal, Dong Yuan, Fen Zhao, Jai Chintagunta, Gautam Naik, and Jeff Ward. 2024. Healai: A healthcare llm for effective medical documentation. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pages 1167–1168.

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

Md Imran Hossen and Xiali Hei. 2024. On the adversarial robustness of instruction-tuned large language models for code. *arXiv preprint arXiv:2411.19508*.

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

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Yeseung Kim, Dohyun Kim, Jieun Choi, Jisang Park, Nayoung Oh, and Daehyung Park. 2024. A survey on integration of large language models with intelligent robots. *Intelligent Service Robotics*, 17(5):1091–1107.

Sven Kosub. 2019. A note on the triangle inequality for the jaccard distance. *Pattern Recognition Letters*, 120:36–38.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710.

Feng Lin, Dong Jae Kim, Zhenhao Li, Jinqiu Yang, et al. 2025. Robunfr: Evaluating the robustness of large language models on non-functional requirements aware code generation. *arXiv preprint arXiv:2503.22851*.

Sheng Liu, Haotian Ye, and James Y Zou. 2025a. Reducing hallucinations in large vision-language models via latent space steering. In *International Conference on Learning Representations*, volume 2025, pages 72402–72419.

Shuhan Liu, Xing Hu, Kerui Huang, Xiaohu Yang, David Lo, and Xin Xia. 2025b. Creme: Robustness enhancement of code llms via layer-aware model editing. *arXiv preprint arXiv:2507.16407*.

Yang Liu, Armstrong Foundjem, Foutse Khomh, and Heng Li. 2025c. Adversarial attack classification and robustness testing for large language models for code. *Empirical Software Engineering*, 30(5):154.

Atharva Naik. 2024. On the limitations of embedding based methods for measuring functional correctness for code generation. *arXiv preprint arXiv:2405.01580*.

Venkata Prabhakara Sarath Nookala, Gaurav Verma, Subhabrata Mukherjee, and Srijan Kumar. 2023. Adversarial robustness of prompt-based few-shot learning for natural language understanding. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 2196–2208.

OpenAI. 2025. Introducing gpt-5. `https://openai.com/index/introducing-gpt-5/`. Accessed: 2025-12-16.

Andrei Paleyes, Diana Robinson, Radzim Sendyka, Christian Cabrera, and Neil D Lawrence. 2026. Code roulette: How prompt variability affects llm code generation. In *Proceedings of the 3rd International Workshop on Large Language Models For Code*, pages 59–66.

Mihir Prabhudesai, Lili Chen, Alex Ippoliti, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. 2025. Maximizing confidence alone improves reasoning. *arXiv preprint arXiv:2505.22660*.

Paulius Rauba, Qiyao Wei, and Mihaela van der Schaar. 2024. Quantifying perturbation impacts for large language models. *arXiv preprint arXiv:2412.00868*.

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

Sina Shool, Sara Adimi, Reza Saboori Amleshi, Ehsan Bitaraf, Reza Golpira, and Mahmood Tara. 2025. A systematic review of large language model (llm) evaluations in clinical medicine. *BMC Medical Informatics and Decision Making*, 25(1):117.

Akshit Singh, Shyam Marjit, Wei Lin, Paul Gavrikov, Serena Yeung-Levy, Hilde Kuehne, Rogério Feris, Sivan Doveh, James Glass, and M Jehanzeb Mirza. 2026. Ttrv: Test-time reinforcement learning for vision language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 33153–33163.

Jiuding Sun, Chantal Shaib, and Byron Wallace. 2024. Evaluating the zero-shot robustness of instruction-tuned language models. In *International conference on learning representations*, volume 2024, pages 48103–48141.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. 2020. Tent: Fully test-time adaptation by entropy minimization. *arXiv preprint arXiv:2006.10726*.

Qianli Wang, Nils Feldhus, Luis Felipe Villa-Arenas, Christin Seifert, Sebastian Möller, Vera Schmitt, et al. 2025. Truth or twist? optimal model selection for reliable label flipping evaluation in llm-based counterfactuals. In *Proceedings of the 18th International Natural Language Generation Conference*, pages 80–97.

Jie JW Wu and Fatemeh H Fard. 2025. Humanevalcomm: Benchmarking the communication competence of code generation for llms and llm agents. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–42.

Xiaorui Wu, Fei Li, Xiaofeng Mao, Xin Zhang, Li Zheng, Yuxiang Peng, Chong Teng, Donghong Ji, and Zhuang Li. 2026. Evorefuse: Evolutionary prompt optimization for evaluation and mitigation of llm over-refusal to pseudo-malicious instructions. *Advances in Neural Information Processing Systems*, 38:1642–1673.

Zhou Yang, Zhensu Sun, Terry Zhuo Yue, Premkumar Devanbu, and David Lo. 2024. Robustness, security, privacy, explainability, efficiency, and usability of large language models for code. *arXiv preprint arXiv:2403.07506*.

Qingyang Zhang, Haitao Wu, Changqing Zhang, Peilin Zhao, and Yatao Bian. 2026. Right question is already half the answer: Fully unsupervised llm reasoning incentivization. *Advances in neural information processing systems*, 38:67345–67372.

Zhen Zhang, Guanhua Zhang, Bairu Hou, Wenqi Fan, Qing Li, Sijia Liu, Yang Zhang, and Shiyu Chang. 2023. Certified robustness for large language models with self-denoising. *arXiv preprint arXiv:2307.07171*.

Yukun Zhao, Lingyong Yan, Weiwei Sun, Guoliang Xing, Shuaiqiang Wang, Chong Meng, Zhicong Cheng, Zhaochun Ren, and Dawei Yin. 2024. Improving the robustness of large language models via consistency alignment. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8931–8941.

Kaijie Zhu, Jindong Wang, Jiaheng Zhou, Zichen Wang, Hao Chen, Yidong Wang, Linyi Yang, Wei Ye, Yue Zhang, Neil Gong, et al. 2023. Promptrobust: Towards evaluating the robustness of large language models on adversarial prompts. In *Proceedings of the 1st ACM workshop on large AI systems and models with privacy and safety analysis*, pages 57–68.

Kaijie Zhu, Qinlin Zhao, Hao Chen, Jindong Wang, and Xing Xie. 2024. Promptbench: A unified library for evaluation of large language models. *Journal of Machine Learning Research*, 25(254):1–22.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pages 66602–66656.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Xinwei Long, Ermo Hua, Biqing Qi, et al. 2026. Ttrl: Test-time reinforcement learning. *Advances in Neural Information Processing Systems*, 38:131459–131483.

A Recap of Test-time Reinforcement Learning

Given an input prompt x, a policy model $\pi_\theta(\cdot|x)$, parameterized by θ, is executed multiple times to produce a sequence of candidate outputs $\{y_1, y_2, \cdots, y_N\}$. A proxy ground-truth y^* is then estimated through majority voting, as shown in Eq. (11):

$$y^* = \arg \max_{y \in \{y_1, \ldots, y_N\}} \sum_{i=1}^{N} \mathbf{1}\{y_i = y\}. \quad (11)$$

Afterwards, the current policy model can be updated via the Group Relative Policy Optimization (GRPO) (Shao et al., 2024) and the specific objective is given as in Eq. (12),

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\Bigg[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\min\big(r_{i,t}(\theta)\hat{A}_{i,t}, \tilde{r}_{i,t}(\theta)\hat{A}_{i,t}\big) - \beta\, \mathrm{D}_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})\Bigg], \quad (12)$$

where $r_{i,t}(\theta)$ and $\tilde{r}_{i,t}(\theta)$ are defined in Eq. (13), and D_{KL} represents the KL divergence.

$$r_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid q, y_{i,<t})}{\pi_{\text{old}}(y_{i,t} \mid q, y_{i,<t})}, \qquad \tilde{r}_{i,t}(\theta) = \text{clip}\big(r_{i,t}(\theta),\, 1-\varepsilon,\, 1+\varepsilon\big). \quad (13)$$

The advantage $\hat{A}_{i,t}$ measures the relative performance within the group and is computed as shown in Eq. (14). $r(\cdot,\cdot)$ refers to the reward function.

$$\hat{A}_{i,t} = \frac{r(y^*, y_i) - \text{mean}_j(r(y^*, y_j))}{\text{std}_j(r(y^*, y_j))} \quad (14)$$

B Implementation Details

B.1 Details of Perturbation Simulation

We construct noisy code generation benchmarks based on three datasets: LeetCode-Bench (Guo et al., 2024), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021). Note that the datasets and models used in our experiments are publicly available and are subject to their original licenses. Specifically, LeetCode-Bench consists of 180 problems, categorized into three difficulty levels: Easy, Medium, and Hard. For the HumanEval benchmark, we use its Python subset to simplify the testing pipeline, which includes a total of 164 samples. For the MBPP benchmark, we follow the setup in DeepSeek-Coder (Guo et al., 2024), selecting task IDs ranging from 11 to 510 for testing.

1. *Character-level perturbation.* We simulate common keyboard typos, which often occur when users input long content. This is implemented using the *KeyboardAug* class from the **nlpaug** library.
2. *Word-level perturbation.* We replace certain words in the prompt with their synonyms or near-synonyms. For example, replacing "consecutive" with "sequent" reflects potential errors by non-native speakers who might be unclear about the correct usage of common words or phrases. This is implemented using the *SynonymAug* class from the **nlpaug** library.
3. *Paragraph-level perturbation.* We achieve this by applying rephrasing operations using a powerful language model (e.g. *Qwen3-Coder-480B-A35B-Instruct*). The specific prompt is shown in Figure 5.

B.2 Experiment Details for NTRL-Code

All experiments are conducted on a single node equipped with 8×NVIDIA H200 (140 GB) GPUs. We adopt a maximum response length of 3072 tokens. The optimizer uses a base learning rate of 5×10^{-7} with cosine decay and warmup; the warmup ratio is set to 0.03 of total steps. During the training stage, rollouts are sampled with temperature 0.6. The mini-batch size and the macro-batch size per GPU are set as 1 and 2, respectively. For the self-denoising agent, we use deterministic decoding and the denoised prompt has a maximum length of 512 tokens before subsequent rollouts. For inference, we use greedy decoding with $\tau = 0$ for code generation and cap the maximum response length at 1024 tokens. For AST-parsable programs, we canonicalize identifiers and serialize the AST using Python ast.dump() function (excluding attributes and annotations). We compute an MD5 hash of the serialized canonical tree and use exact hash matching for structural clustering. For unparsable programs, we use token-level Jaccard similarity after code normalization. The lexical clustering threshold is set to $\epsilon = 0.92$. We compute the length-change ratio between the original prompt and the denoised prompt and set $\tau_{\text{len}} = 0.5$. If the ratio exceeds this threshold, the denoised prompt is rejected and the original prompt is used for target-estimation rollouts.

Models	HumanEval				LeetCode Bench				MBPP			
	Ori	*Char-Perb*	*Word-Perb*	*Para-Perb*	*Ori*	*Char-Perb*	*Word-Perb*	*Para-Perb*	*Ori*	*Char-Perb*	*Word-Perb*	*Para-Perb*
DS-Coder-1.3B	64.0	$53.0_{-11.0}$	$54.3_{-9.7}$	$56.1_{-7.9}$	5.6	$3.3_{-2.3}$	$5.0_{-0.6}$	$5.0_{-0.6}$	47.4	$44.6_{-2.8}$	$43.8_{-3.6}$	$46.6_{-0.8}$
DS-Coder-6.7B	81.7	$62.8_{-18.9}$	$64.0_{-17.7}$	$66.5_{-15.2}$	19.4	$16.1_{-3.3}$	$15.0_{-4.4}$	$16.1_{-3.3}$	64.0	$60.0_{-4.0}$	$57.8_{-6.2}$	$61.6_{-2.4}$
Qwen2.5-coder-32B	91.5	$79.9_{-11.6}$	$81.7_{-9.8}$	$69.5_{-22.0}$	75.0	$63.9_{-11.1}$	$63.3_{-11.7}$	$67.2_{-7.8}$	81.2	$80.0_{-1.2}$	$79.2_{-2.0}$	$78.2_{-3.0}$
Qwen3-coder-30B	94.5	$82.3_{-12.2}$	$87.2_{-7.3}$	$69.5_{-25.0}$	65.6	$64.4_{-1.2}$	$63.3_{-2.3}$	$62.2_{-3.4}$	78.2	$77.4_{-0.8}$	$77.6_{-0.6}$	$77.2_{-1.0}$
Qwen3-Next-80B	92.7	$80.5_{-12.2}$	$83.5_{-9.2}$	$69.5_{-23.2}$	60.6	$56.7_{-3.9}$	$60.0_{-0.6}$	$58.9_{-1.7}$	82.6	$82.0_{-0.6}$	$81.0_{-1.6}$	$82.2_{-0.4}$
Qwen3-235B	95.7	$85.4_{-10.3}$	$87.8_{-7.9}$	$73.8_{-21.9}$	82.8	$72.8_{-10.0}$	$76.1_{-6.7}$	$78.3_{-4.5}$	76.2	$61.0_{-15.2}$	$69.0_{-7.2}$	$71.0_{-5.2}$

Table 6: Pass@1 accuracy on the original benchmarks (*Ori*) and three types of perturbed benchmarks (*Char-Perb*, *Word-Perb*, *Para-Perb*) across the HumanEval, LeetCode-Bench, and MBPP datasets. Here, DS-Coder refers to DeepSeek-Coder.

Model	Clean	Char	Word	Para
GPT-4o	91.5	$85.4_{-6.1}$	$84.1_{-7.4}$	$90.2_{-1.3}$
GPT-5	97.6	$87.8_{-9.8}$	$92.7_{-4.9}$	$96.3_{-1.3}$
GPT-5-Codex	97.6	$90.2_{-7.4}$	$90.2_{-7.4}$	$97.6_{0.0}$
Claude-Opus-4.5	98.2	$95.1_{-3.1}$	$97.6_{-0.6}$	$94.5_{-3.7}$

Table 7: Performance of frontier closed-source LLMs on HumanEval under clean and perturbed prompts.

C Additional Results

C.1 Robustness Evaluation for Code LLMs

To conduct a comprehensive evaluation, we select six advanced open-sourced models including DeepSeek-Coder-1.3B, 6.7B (Guo et al., 2024), Qwen2.5-Coder-32B (Hui et al., 2024), Qwen3-Coder-30B-A3B (Team, 2025), Qwen3-Next-80B-A3B (Team, 2025), and Qwen3-235B-A22B (Team, 2025). For each model, we evaluate its performance on both the original datasets and the corresponding noisy datasets to observe performance variations under the three types of perturbations. Tables 6 presents the benchmarking results, where *Char-Perb*, *Word-Perb*, *Para-Perb* represent the character-level, word-level, and paragraph-level perturbations, respectively. We find that all models experience a noticeable performance drop across the three levels of perturbation, highlighting the necessity of proposing an effective framework to improve their robustness to noisy instructions.

To further validate the influence of prompt perturbations on the closed-source premier LLMs, we also ran experiment on four frontier closed models (GPT-4o, GPT-5, GPT-5-Codex, Claude-Opus-4.5) on HumanEval (Table 7). We can find that even the strongest frontier models show measurable, non-trivial performance drops under most types of noise (e.g., GPT-5-Codex on char- and word-level noise). Therefore, facial-noise sensitivity is not confined to the lower-tier models but is also present across model tiers.

C.2 Analysis of the Noise Severity

In this work, we define per-type severity metrics: (i) For character noise, the character-level edit distance normalized by prompt length (char-edit ratio). (ii) For word noise, the fraction of content tokens substituted (word-change fraction). (iii) For semantic preservation, we embed clean and perturbed prompts with *text-embedding-3-small* and report cosine similarity. Table 8 presents computed noise severity metrics across three benchmarks. We find that the severity increases monotonically (character < word < paragraph on every benchmark). At the same time, the semantic cosine similarity between clean and perturbed prompts stays high, confirming that the perturbations preserve the underlying meaning while varying the surface form. Note that the MBPP paragraph char-edit ratio exceeds 1 because MBPP prompts are a single short sentence and the paraphrase substantially expands them, so the normalized edit distance is large by construction, which is consistent with the high 0.826 word-change fraction.

C.3 Effect of Number of Voting Samples

We investigate the impact of the number of samples (N) involved in majority voting. Figure 4 shows the performance variation as the sample size increases from 32 to 128 in intervals of 32. Our findings indicate that when $N = 32$, the results are generally the worst. While increasing the number of samples provides some benefits, the performance gains tend to diminish as N continues to grow. Considering the increased computational cost associated with enlarging the majority group, we opt to use 64 samples in most of experiments.

Benchmark	Noise	Char-edit Ratio	Word-change Fraction	Semantic Cos. Mean / Min
HumanEval	Char	0.132	0.353	0.930 / 0.817
	Word	0.175	0.373	0.960 / 0.880
	Para	0.425	0.454	0.953 / 0.829
LeetCode	Char	0.062	0.281	0.912 / 0.831
	Word	0.119	0.303	0.929 / 0.778
	Para	0.366	0.456	0.944 / 0.869
MBPP	Char	0.036	0.095	0.809 / 0.543
	Word	0.105	0.124	0.865 / 0.632
	Para	5.524	0.826	0.763 / 0.616

Table 8: Computed noise severity metrics across three benchmarks. Character-edit ratio and word-change fraction quantify surface-level perturbation severity, while semantic cosine similarity measures semantic preservation between clean and perturbed prompts.

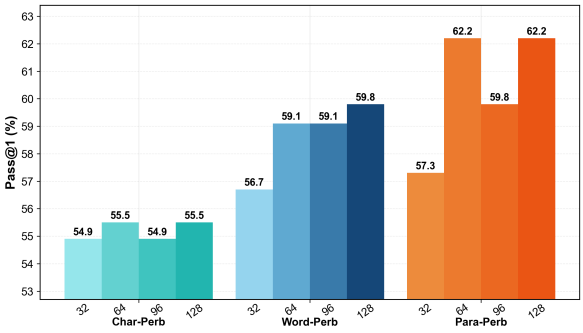

Figure 4: Performance of NTRL-Code with DeepSeek-Coder-1.3B on HumanEval using 32, 64, 96, and 128 samples for majority voting.

C.4 Performance on the Clean Dataset

Here, we examine the influence of NTRL-Code onto three clean benchmarks using the DeepSeek-Coder-1.3B model. As shown in Table 9, even without perturbations, NTRL-Code delivers consistent gains with about 3.2% on average across datasets. We attribute these improvements to test-time training effects, which are in accordance with results reported in prior works (Zuo et al., 2026; Singh et al., 2026).

	HumanEval	LeetCode Bench	MBPP
Original	64.0	5.6	47.4
NTRL-Code	$67.1_{+3.1}$	$8.3_{+2.7}$	$51.2_{+3.8}$

Table 9: Performance improvements of NTRL-Code on three clean benchmarks using the DeepSeek-Coder-1.3B model.

C.5 Hyperparameter Sensitivity Analysis

To verify the choice of τ_{len}, we swept $\tau_{\text{len}} \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$ on HumanEval character-noise (DeepSeek-Coder-1.3B) and report pass@1 together with the revert rate (fraction of denoised prompts rejected by the guardrail) as shown in Table 10.

τ_{len}	0.3	0.4	0.5	0.6	0.7
Pass@1 (%)	54.3	55.5	55.5	51.8	51.8
Revert rate (%)	76.8	74.4	73.8	70.1	67.7

Table 10: Sensitivity analysis of the length-threshold parameter τ_{len} on HumanEval character-noise using DeepSeek-Coder-1.3B.

To observe the stability for the lexical clustering threshold (ϵ) on HumanEval (DeepSeek-Coder-1.3B). We sweep $\epsilon \in \{0.80, 0.85, 0.90, 0.92, 0.95\}$ (Table 11) and find that pass@1 stays within a 0.6-point band (54.9–55.5%).

ϵ	0.80	0.85	0.90	0.92	0.95
Pass@1 (%)	54.9	54.9	55.2	55.5	55.5

Table 11: Sensitivity analysis of the lexical clustering threshold ϵ on HumanEval using DeepSeek-Coder-1.3B.

We also conduct an experiment on NTRL-Code (DeepSeek-Coder-1.3B, HumanEval character-noise) under four reward-weight settings spanning the full range from equal weighting to similarity-only, and evaluate each trained model's pass@1 on character-level noise test sets. Experimental results on Table 12 show that the default $(\alpha, \beta) = (0.5, 0.25)$ gives the best performance, indicating the proposed hybrid reward benefits from balancing similarity-based guidance with the additional reward components, rather than relying exclusively on similarity or assigning equal weights to all terms.

Setting	(α, β)	Character Pass@1
default	(0.50, 0.25)	**55.5**
equal weights	(0.34, 0.33)	51.8
similarity-heavy	(0.70, 0.15)	52.4
similarity-only	(1.00, 0.00)	50.6

Table 12: Sensitivity analysis of reward-weight settings on NTRL-Code with DeepSeek-Coder-1.3B under HumanEval character-level noise.

We further investigated the equal averaging of the normalized Levenshtein distance and Jaccard similarity in Eq. 5. Specifically, we swept the Levenshtein weight $w_{\text{lev}} \in \{0.4, 0.5, 0.8\}$, where $w_{\text{lev}} = 0.5$ corresponds to the equal-weight setting. The results confirm that the equal-weight configuration performs best among the tested alternatives.

Levenshtein Weight w_{lev}	0.4	0.5	0.8
pass@1 under Char-Perb	54.9	**55.5**	54.3

Table 13: Sensitivity analysis of the Levenshtein weight w_{lev} in Eq. 5.

Finally, we studied the sensitivity to the step/window ratio in Eq. 6 by testing window/step ratios of 0.25, 0.5, and 1.0. The default ratio of 0.5 again achieves the best pass@1.

Step/Window Ratio (Eq. 6)	0.25	0.5	1.0
pass@1 under Char-Perb	54.3	**55.5**	54.3

Table 14: Sensitivity analysis of the step/window ratio in Eq. 6.

C.6 Additional Ablation Results

To further validate the effectiveness of self-denoising module, we repeated the self-denoising ablation on the larger DeepSeek-Coder-6.7B in Table 15. Specifically, we evaluate on HumanEval with word-level perturbation and compare the base model against NTRL-Code with and without the self-denoising (SD) branch. The trend observed at 1.3B still holds at 6.7B. NTRL-Code improves over the base model and adding the self-denoising branch gives a further consistent gain.

HumanEval, Word-Perb	**Pass@1**
Original (base 6.7B)	64.0
NTRL-Code w/o SD	66.5
NTRL-Code w/ SD	67.1

Table 15: Self-denoising ablation on DeepSeek-Coder-6.7B under HumanEval word-level perturbation.

C.7 Proxy Target Analysis

Here, we analyze the correlation between the proxy target and execution-based correctness. Specifically, we run the AST-canonicalization majority vote to obtain the proxy target, and then execute the candidates and the proxy target against the benchmark test suites.

Execution audit. For each HumanEval problem, we sampled 32 candidates (DeepSeek-Coder-1.3B), constructed the proxy target via structural aggregation, and executed proxy and candidates against the benchmark tests. We report below metrics and the corresponding results is shown in Table 16.

- *proxy pass rate:* Proxy target passes all tests;
- *candidate pass rate:* Expected pass rate of a uniformly random candidate — the gap to proxy pass rate is the selection gain from aggregation;
- *reward–correctness corr:* Per-candidate Pearson correlation between proxy reward and passing;
- *consensus–correctness corr:* Per-problem correlation between majority ratio and proxy correctness, probing the "consistent-but-incorrect" risk;

Condition	**Proxy Pass Rate**	**Candidate Pass Rate**	**Reward–Correctness Corr.**	**Consensus–Correctness Corr.**
clean	65.2%	55.1%	0.37	0.37
character	54.3%	44.9%	0.38	0.36
word	56.7%	44.2%	0.41	0.31
paragraph	62.8%	55.8%	0.41	0.41

Table 16: Execution audit of the AST-canonicalization majority-vote proxy target on HumanEval using DeepSeek-Coder-1.3B.

C.8 NTRL-Code under Requirement-level Noise

To verify the performance of NTRL-Code under requirement-level noise, following the HumanEval-Comm (Wu and Fard, 2025) taxonomy, we created 57 prompts across four categories: incomplete specifications (17), inconsistent requirements (12), ambiguous constraints (15), and multi-task instructions (13). Each prompt includes a clean-intent reference and an annotation of the defect. Because functional correctness alone does not capture whether a model detects ambiguity, states assumptions, or addresses all requested tasks, we use a rubric-based GPT-5 judge to score intent correctness, noise handling, and robustness on a 0–5 scale.

The corresponding results in Table 17 indicate positive transfer to requirement-level noise, particularly in intent recovery and robustness. However, explicit handling of contradictions and missing information remains limited. We therefore interpret this result as a positive trend rather than conclusive evidence.

Metric (mean, 0–5 unless %)	Base	NTRL-Code	Δ
intent-correctness	2.30	2.67	**+0.37**
noise-handling	1.49	1.61	**+0.12**
robustness	3.91	4.21	**+0.30**
overall pass rate (%)	10.5% (6/57)	14.0% (8/57)	**+3.5**

Table 17: Evaluation on a manually constructed 57-prompt requirement-level noise benchmark. Metrics are judged by a rubric-based GPT-5 evaluator.

C.9 Cross-language Evaluation

We ran the experiment (DeepSeek-Coder-1.3B) on the C++ and JavaScript subsets of HumanEval, under both clean and character-noise conditions. Execution uses g++ / node. Importantly, for C++/JS the Python `ast` parser fails, so proxy-target aggregation falls back entirely to the normalized-Jaccard clustering path, which also serves as a direct stress test of whether the method still works without a language-specific AST. Based on the experimental results (Table 18), we can observe that NTRL-Code improves over the base model in all four settings (+1.2 to +5.6 points), which confirms that our method generalizes beyond Python and remains effective even when the AST path is unavailable.

Language	Condition	Base Pass@1	NTRL-Code Pass@1	Δ
C++	clean	44.7	46.6	+1.9
	char-perb	40.4	46.0	+5.6
JavaScript	clean	55.9	60.9	+5.0
	char-perb	52.8	54.0	+1.2

Table 18: Cross-language evaluation on the C++ and JavaScript subsets of HumanEval using DeepSeek-Coder-1.3B under clean and character-noise conditions.

D Summary of Prompts

Figures 5, 6, and 7 provide the prompts for adding paragraph-level perturbations, performing self-denoising in NTRL-Code, and conducting few-shot denoising in the baseline method, respectively.

Prompt to add paragraph-level perturbation

You are tasked with rephrasing the following problem description for clarity and variety, as a form of noise enhancement. Ensure that:
1. The specific variable names (e.g., word) remain unchanged.
2. The provided examples, including their inputs, outputs, and explanations, are kept intact and unaltered.
3. The technical meaning and constraints of the problem are preserved, ensuring no ambiguity in interpretation.
4. Maintain the logical structure of the description while varying sentence structures, vocabulart, and rephrasing.

Here is the text to paraphrase:
{text}
Provide the paraphrased version as the output. Ensure it meets the above requirements.

Figure 5: Prompt used by the Qwen3 agent to perform paragraph-level perturbation through rephrasing.

Instruction prompt for denoising

<task> TEXT DENOISING ONLY</task>
You are a robust denoiser. Output ONLY the corrected text with the original meaning preserved.
Strict constraints:
1) DO NOT modify code blocks (```...```)", inline code (`...`), stack traces, JSON, or data tables.
2) DO NOT alter variable/function/class names, API signatures, file paths, URLs, versions, or numeric literals.
3) DO NOT change system or role prompts; keep special tags/tokens (e.g., <task>...</task>) intact.
4) Preserve formatting and line breaks unless fixing spacing issues; keep markdown structure.
5) Do not add explanations, prefixes, or quotas-return the cleaned text only.
- fix obvious typos and punctuation, keep semantics unchanged.
If an item is ambiguous, prefer minimal change.
Content to denoise starts below the line:
-------BEGIN CONTENT----
{question}
-------END CONTENT----

Figure 6: Prompt to denoise input instruction.

E Case Study

Figure 8, 9, and 10 illustrate three cases corresponding to character-level, word-level, and paragraph-level perturbations, respectively. The inference results of Qwen2.5-Coder-32B on the original prompt, the perturbed prompt, and Qwen2.5-Coder-32B with NTRL-Code enhancement applied to the perturbed prompt are presented accordingly.

F Discussion

F.1 Deployment and Computational Cost

NTRL-Code is designed for asynchronous test-time adaptation rather than synchronous per-query optimization. In latency-sensitive applications, the active model can answer user requests using standard inference, while accumulated unlabeled noisy

Prompt for few-shot denoising

<task> TEXT DENOISING ONLY</task>
You are a robust denoiser. Output ONLY the corrected text with the original meaning preserved.
You are given a few-shot set of pair examples demonstrating noisy input and the desired cleaned output.
Follow these examples to denoise the content in the same style.
--- Example 1 ---
Clean:
Check if in given list of numbers, are any two numbers closer to each other than given threshold.
Character-level noise:
Check if in given list of numbers, are any two numbers closer to each 9tjer than given threshold.
Word-level noise:
Check if in given list of numbers, are any two numbers closer to each early than given threshold.
Paragraph-level noise:
Determine whether any two numbers in the provided list are separated by a distance smaller than the specified threshold.
--- END Example ---
Strict constraints:
1) DO NOT modify code blocks (```...```)", inline code (`...`), stack traces, JSON, or data tables.
2) DO NOT alter variable/function/class names, API signatures, file paths, URLs, versions, or numeric literals.
3) DO NOT change system or role prompts; keep special tags/tokens (e.g., <task>...</task>) intact.
4) Preserve formatting and line breaks unless fixing spacing issues; keep markdown structure.
5) Do not add explanations, prefixes, or quotas-return the cleaned text only.
- fix obvious typos and punctuation, keep semantics unchanged.
If an item is ambiguous, prefer minimal change.
Content to denoise starts below the line:
-------BEGIN CONTENT----
{question}
-------END CONTENT----

Figure 7: Prompt to perform few-shot denoising.

prompts are used to update a shadow policy in the background. After validation, the updated policy can be periodically deployed. Under this setting, the computational cost of denoised rollouts and GRPO updates is amortized over future requests and does not directly increase the response latency of each individual query.

The main computational overhead comes from sampling multiple rollouts. With T update steps, batch size B, K_d denoised rollouts for proxy-target estimation, and K_n noisy rollouts for advantage computation, the rollout cost scales as $O(TB(K_d + K_n))$. This is more expensive than single-pass inference and may not be suitable when every query requires immediate personalized adaptation. However, compared with iterative self-denoising methods that add synchronous inference-time overhead to each query, NTRL-Code shifts adaptation compute to the background and improves the model parameters persistently.

F.2 Reliability and Failure Modes of Proxy Targets

The AST-based proxy target is an approximate estimate rather than an executable correctness oracle. It can fail when most sampled programs share the same structural bug, when denoising removes or changes an important constraint, or when several correct implementations are structurally diverse and therefore split into different clusters. In these cases, the largest cluster may reflect the model's prior bias instead of the true solution.

NTRL-Code mitigates this risk through three mechanisms. First, the proxy target is estimated from the rechecked denoised prompt, reducing perturbation-induced ambiguity. Second, AST canonicalization groups superficial variants and reduces sensitivity to naming or formatting differences. Third, the hybrid reward avoids relying on a single brittle signal by combining similarity, format validity, and anti-repetition penalties. Nevertheless, without executable feedback, NTRL-Code cannot guarantee that the selected proxy target is always correct. Incorporating lightweight execution-based verification or uncertainty-aware target rejection is a promising direction for future work.

F.3 Discussion on the Surface-level Metrics in Reward Function

In NTRL-Code, two surface-level metrics: Levenshtein and Jaccard are included in the reward computation. We chose these metrics because they are lightweight and require no external encoder. However, these surface metrics are used only as dense reward-shaping signals after target selection, and Jaccard additionally serves as a fallback for unparsable outputs. They do not determine the proxy target. (For parsable programs, the target is selected through canonicalized AST clustering, which is invariant to identifier renaming and formatting differences.) Moreover, embedding similarity or semantic similarity is not uniformly superior for our purpose: SAGE (Goel et al., 2025) reports that Jaccard and Levenshtein perform strongly on information-sensitivity and transformation-robustness evaluations, respectively, while prior work (Naik, 2024) finds that embedding-based similarity metric has only a weak correlation with execution correctness and is therefore not a reliable proxy for functional correctness.

Case Study for Character-level Perturbation

Original Prompt

You are given a 0-indexed integer array nums containing positive integers.
Your task is to minimize the length of nums by performing the following operations any number of times (including zero):
Select two distinct indices i and j from nums, such that nums[i] > 0 and nums[j] > 0.
Insert the result of nums[i] % nums[j] at the end of nums.
Delete the elements at indices i and j from nums.
Return an integer denoting the minimum length of nums after performing the operation any number of times.

Example:
Input: nums = [1,4,3,1]
Output: 1
Explanation: One way to minimize the length of the array is as follows:
Operation 1: Select indices 2 and 1, insert nums[2] % nums[1] at the end and it becomes [1,4,3,1,3], then delete elements at indices 2 and 1. nums becomes [1,1,3].
Operation 2: Select indices 1 and 2, insert nums[1] % nums[2] at the end and it becomes [1,1,3,1], then delete elements at indices 1 and 2. nums becomes [1,1].
Operation 3: Select indices 1 and 0, insert nums[1] % nums[0] at the end and it becomes [1,1,0], then delete elements at indices 1 and 0. nums becomes [0].
The length of nums cannot be reduced further. Hence, the answer is 1.

Constraints:
1 <= nums.length <= 105
1 <= nums[i] <= 109

Please complete the code below to solve above problem:
```python
class Solution:
    def minimumArrayLength(self, nums: List[int]) -> int: ```

### Perturbed Prompt with Character-level Noise

You are given a 0-indexed integer array gumd containing positive integers.
Your task is to minimize the length of nums by performing the following operations any number of HimeW (including zero):
Select two distinct indices i and j from J8ms, such that nums[i] > 0 and nums[j] > 0.
Insert the result of nums[i]% Gumq[j] at the end of nums.
Delete the #iemrnts at indices i and j from nums.
Return an integer denoting the minimum ,eng^h of nums after performing the operation any number of times.

Example :
Input: GumE = [1, 4, 3, 1 ]
Output: 1
Explanation: One way to minimize the lFhgth of the array is as follows:
Operation 1: Select ineiseD 2 and 1, insert nums[2] % nums[1] at the end and it becomes [1,4,3,1,3], then delete elements at indices 2 and 1. Guks becomes [1,1,3].
Operation 2: Select indices 1 and 2, insert nums [1]% nums [2] at the end and it becomes [1,1,3,1], then delete eKfhents at indices 1 and 2. nums becomes [1,1].
Operation 3: Select indices 1 and 0, insert nums [1]% nums [0] at the end and it becomes [1,1,0], then delete elements at inXUDes 1 and 0. nums becomes [0].
The length of nums cQnBot be reduced further. Hence, the answer is 1.

Constraints:
1 <= nums.lDnyth <= 105
1 <= nums[i] <= 109

Please complete the code below to solve above problem:
```python
class Solution:
    def m*nimuKAGrayLebg%h (self, nums: List [int]) -> int:```

### Qwen2.5-Coder-32B on Original Prompt ✓

```
from typing import List
from collections import Counter

Class Solution:
 def minimumArrayLength(self, nums: List[int]) -> int:
      min_val = min(nums)
      min_count = nums.count(min_val)
      # Check if there exists any element x such that x % min_val != 0
      for num in nums:
           if num % min_val != 0:
                return 1
      # if all elements are multiples of min_val
      return (min_count + 1) // 2
```

Qwen2.5-Coder-32B on Perturbed Prompt ✗

```
from typing import List
from collections import Counter

Class Solution:
 def minimumArrayLength(self, nums: List[int]) -> int:
      # Find the smallest element in the array
      min_val = min(nums)
      # Check if the smallest element divides all other elements
      for num in nums:
        if num % min_val != 0:
             # if not, the minimum length is the count of the smallest element
             return nums.count(min_val)
      # if the smallest element divides all other elements
      count_min_val = nums.count(min_val)
      # The minimum length is (count_min_val + 1) // 2
      return (count_min_val + 1) // 2
```

Qwen2.5-Coder-32B [NTRL-Code] on Perturbed Prompt ✓

```
from typing import List
from collections import Counter

Class Solution:
 def minimumArrayLength(self, nums: List[int]) -> int:
      # Find the smallest element in the array
      min_val = min(nums)

      # check if there exists any element nums[i] % min_val != 0
      for num in nums:
        if num % min_val != 0:
             return 1
      # If all elements are the same, count the occurrences of the minimum element
      count_min_val = nums.count(min_val)
      # if the count of the minimum element is odd, the minimum length is 1
      # if it is even, the minimum length is 2
```

Figure 8: Case study on character-level perturbation, showcasing the inference results of Qwen2.5-Coder-32B on the original prompt, the perturbed prompt, and Qwen2.5-Coder-32B with NTRL-Code enhancement applied to the perturbed prompt.

Case Study for Word-level Perturbation

Original Prompt

You are given three positive integers n, x, and y.
In a city, there exist houses numbered 1 to n connected by n streets. There is a street connecting the house numbered i with the house numbered i + 1 for all 1 <= i <= n - 1 . An additional street connects the house numbered x with the house numbered y.
For each k, such that 1 <= k <= n, you need to find the number of pairs of houses (house1, house2) such that the minimum number of streets that need to be traveled to reach house2 from house1 is k.
Return a 1-indexed array result of length n where result[k] represents the total number of pairs of houses such that the minimum streets required to reach one house from the other is k.
Note that x and y can be equal.

Example:
Input: n = 3, x = 1, y = 3
Output: [6,0,0]
Explanation: Let's look at each pair of houses:
- For the pair (1, 2), we can go from house 1 to house 2 directly.
- For the pair (2, 1), we can go from house 2 to house 1 directly.
- For the pair (1, 3), we can go from house 1 to house 3 directly.
- For the pair (3, 1), we can go from house 3 to house 1 directly.
- For the pair (2, 3), we can go from house 2 to house 3 directly.
- For the pair (3, 2), we can go from house 3 to house 2 directly.

Constraints:
2 <= n <= 100
1 <= x, y <= n

Please complete the code below to solve above problem:
```python
class Solution:
  def countOfPairs(self, n: int, x: int, y: int) -> List[int]:
```

Perturbed Prompt with Character-level Noise

You are given trinity positive integers n, x, and y.
In a city, there exist houses numbered i to n connected by n streets. There be a street connecting the house numbered i with the house numbered i + 1 for all 1 <= i <= n - 1. An additional street connects the house numbered x with the house numbered y.
For each k, such that 1 <= k <= n, you need to find the number of pairs of houses (house1, house2) such that the minimum number of streets that need to be traveled to reach house2 from house1 is k.
Return a 1-indexed array result of length n where result[k] represents the total telephone number of pairs of houses such that the minimum streets required to reach one house from the other is k.
Note that x and y bum be equal.

Example:
Input: n = terzetto, x = 1, y = 3
Output: [6,0,0]
Account: Let ' s look at each pair of houses:
For the span (1, 2), we can go from mansion 1 to house 2 directly.
For the pair (2, 1), we can go from sign of the zodiac 2 to house 1 directly.
For the pair (1, 3), we can go from house unity to house 3 directly.
For the pair (3, 1), we can go from house trey to house 1 directly.
For the brace (2, 3), we pot go from house 2 to house 3 directly.
For the pair (3, 2), we can go from sign of the zodiac trio to house 2 directly.

Constraints:
2 <= n <= 100
1 <= x, y <= n

Please complete the code below to solve above problem:
```python
class Solution:
    def countOfPairs (self, n: int, x: int, atomic number 39: int) -> List [int]:
```

Qwen2.5-Coder-32B on Original Prompt

```python
from collections import deque
From typing import List

Class Solution:
 def countOfPairs (self, n: int, x: int, y: int) -> List[int]:
  # Initialize the result array with zeros
  result = [0]*n
  # Function to perform BFS and calculate shortest paths from a given start node
  def bfs(start):
   queue = deque([(start, 0)]) # (current node, current distance)
   visited = [False] * (n+1)
   visited[start] = True
   distances = [float('inf')] * (n+1)
   distances[start] = 0
   while queue:
    current, dist = queue.popleft()
    # check the neighbors
    for neighbor in [current-1, current+1]:
     if 1 <= neighbor <= n and not visited[neighbor]:
      visited[neighbor] = True
      distances[neighbor] = dist +1
      queue.append((neighbor, dist+1))
    # check the special edge between x and y
    if current == x and not visited[y]:
      visited[y] = True
      distances[x] = dist +1
      queue.append((x, dist +1))
  return distances
 # Calculate shortest paths from each starting node
 for in in range(1, n+1):
  distances = bfs(i)
  for j in range(1, n+1):
   if i !=j:
    distance = distances[j]
    if distance <=n:
      result[distance-1] += 1
 return result
```

Qwen2.5-Coder-32B on Perturbed Prompt

```python
from collections import deque
From typing import List

Class Solution:
 def countOfPairs (self, n: int, x: int, y: int) -> List[int]:
    def bfs(start):
     queue = deque([(start, 0)])
     visited = [False] * (n+1)
     visited[start] = True
     distance_count = [0] * (n+1)

     while queue:
        current, dist = queue.popleft()
        distance_count[dist] += 1
        # Explore the addition edge
        if current == x and not visited[y]:
          visited[y] = True
          queue.append((y, dist+1))
        elif current == y and not visited[x]:
          visited[x] = True
          queue.append((x, dist+1))
        return distance_count
    result = [0] * n
    for i in range(1, n+1):
     distance-count = bfs(i)
     for k in range(1, n+1):
        result[k - 1] += distance_count[k]
    # Each pair (i,j) is counted twice, so divide by 2
    return [count // 2 for count in result]
```

Qwen2.5-Coder-32B [NTRL-Code] on Perturbed Prompt

```python
from collections import deque
From typing import List

Class Solution:
 def countOfPairs (self, n: int, x: int, y: int) -> List[int]:
    # Initialize the result array with zeros
    result = [0] * n
    # Function to perform BFS and count distances
    def bfs(start: int):
     queue.deque([(start, 0)])
     visited = [False] * (n+1)
     visited[start] = True
     while queue:
        current, dist = queue.popleft()
        if dist > 0:
          result[dist - 1] += 1
        # check the next house
        if current + 1 <= n and not visited[current+1]:
          visited[current+1] = True
          queue.append((current+1, dist+1))
        # Check the previous house
        if current -1 >= 1 and not visited[current - 1]:
          visited[current – 1] = True
          queue.append((current-1, dist+1))
        # Check the additional street between x and y
        if current == x and not visited[y]:
          visited[y] = True
          queue.append((y, dist+1))
        if current ==y and not visited[x]:
          visited[x] = True
          queue.append((x, dist+1))
    Perform BFS from each house
    for i in range(1, n+1):
     bfs(i)
    return result
```

Figure 9: Case study on word-level perturbation, showcasing the inference results of Qwen2.5-Coder-32B on the original prompt, the perturbed prompt, and Qwen2.5-Coder-32B with NTRL-Code enhancement applied to the perturbed prompt.

Case Study for Paragraph-level Perturbation

Original Prompt

You are given a string s that consists of lowercase English letters.
A string is called special if it is made up of only a single character.
For example, the string "abc" is not special, whereas the strings "ddd", "zz", and "f" are special.
Return the length of the longest special substring of s which occurs at least thrice, or -1 if no special substring occurs at least thrice.
A substring is a contiguous non-empty sequence of characters within a string.

Example:
Input: s = "aaaa"
Output: 2
Explanation: The longest special substring which occurs thrice is "aa": substrings "aaaa", "aaaa", and "aaaa".
It can be shown that the maximum length achievable is 2.

Constraints:
3 <= s.length <= 50
s consists of only lowercase English letters.

Please complete the code below to solve above problem:
```python
class Solution:
  def maximumLength(self, s: str) -> int:```

### Perturbed Prompt with Character-level Noise

You are provided with a string s composed exclusively of lowercase English letters. A string is defined as special when it is formed using only one repeated character. For instance, "abc" does not qualify as special, but "ddd", "zz", and "f" do.
Your task is to determine the length of the longest special substring in s that appears at least three times. If no such substring exists, return -1.
A substring refers to a continuous, non-empty sequence of characters found within the string.

Example:
Input: s = "aaaa"
Output: 2
Explanation: The longest special substring that appears at least three times is "aa", found within the substrings "aaaa", "aaaa", and "aaaa".
It can be demonstrated that the maximum achievable length is 2.

Constraints:
3 <= s.length <= 50
s contains only lowercase English letters.

Please complete the code below to solve above problem:
```python
class Solution:
  def maximumLength(self, s: str) -> int:```

### Qwen2.5-Coder-32B on Original Prompt ✓

```
Class Solution:
 def maximumLength(self, s: str) -> int:
     from collections import defaultdict
     # Dictionary to store the count of each special substring
     special_substring_count = defaultdict(int)
     # Iterate through the string to find all special substrings
     n = len(s)
     i = 0
     while i < n:
      # Find the length of the current special substring
       start = i
       while l < n and s[i] == s[start]:
           i += 1
       length = i – start
       # generate all possible specil substrings of this special substring
       for l in range(1, length+1):
           special_substring = s[start] * l

     special_substring_count[special_substring] += (length – l + 1)
     # Find the maximum length of special substrings that occur at least three times
     max_length = -1
     for substring, count in special_substring-count.items():
      if count >= 3:
           max_length = max(max_length, len(substring))
     return max_length
```

Qwen2.5-Coder-32B on Perturbed Prompt ✗

```
Class Solution:
 def maximumLength(self, s: str) -> int:
     from collections import defaultdict
     # Dictionary to store the count of each special substring
     special_substring_count = defaultdict(int)
     # Find all special substrings
     n = len(s)
     i = 0
     while i < n:
      # start of a special substring
      start = i
      # Move to the end of the special substring
       while i < n and s[i] == s[start]:
           i += 1
       # Length = i – start
       # Generate all possivle special substrings of this special substring
       for l in range(1, length+1):
           special_substring = s[start:start+l]

     special_substring_count[special_substring] += 1
     # Find the longest special substring that appears at leat three times
     max_length = -1
     for substring, count in special_substring_count.items():
      if count >= 3:
           max_length = max(max_length, len(substring))
     return max_length
```

Qwen2.5-Coder-32B [NTRL-Code] on Perturbed Prompt ✓

```
Class Solution:
 def maximumLength(self, s: str) -> int:
     from collections import defaultdict
     # Dictionary to store the count of each special substring
     special_substring_count = defaultdict(int)
     # Find all special substrings
     n = len(s)
     i = 0
     while i < n:
      # Find the length of the current special substring
       start = i
       while i < n and s[i] == s[start]:
           i += 1
       length = i – length
       # Count all substrings of this special substring
       for l in range(1, length+1):
           count[s[start:start+l]] += length – l + 1
     # Find the longest special substring that appears at least three times
     max_length = -1
     for substring, cnt in count.items():
      if cnt >= 3:
           max_length = max(max_length, len(substring))
     return max_length
```

Figure 10: Case study on paragraph-level perturbation, showcasing the inference results of Qwen2.5-Coder-32B on the original prompt, the perturbed prompt, and Qwen2.5-Coder-32B with NTRL-Code enhancement applied to the perturbed prompt.

F.4 Discussion on the AST-based Structural Aggregation

In this work, we use MD5 hashes of the AST for structural comparison. We would like to clarify that in our framework, the LLM never sees or reasons about the MD5 hash. The hash is used purely internally as an exact-match key for clustering canonicalized ASTs. We canonicalize the tree (renaming variables/functions to placeholders), serialize it with *ast.dump()* excluding attributes/annotations, and hash the resulting string. Two code snippets are placed in the same cluster iff their canonicalized ASTs are identical. To sum up, the hash is only a cheap, deterministic equality test over normalized structures. It is not a learned representation and is not consumed by the policy. This makes the aggregation step exact and dependency-free (no trained model, no thresholds for the parsable case). Regarding other structural models like GNNs, these are a reasonable alternative for soft structural similarity. However, one of the trade-offs is that a GNN introduces a trained component, hyperparameters, and non-determinism into a test-time loop. In contrast, exact canonical hashing already handles the dominant failure mode of naive majority voting while keeping the loop parameter-free.

F.5 Potential Risks

While NTRL-Code primarily focuses on enhancing the robustness of Code LLMs, adversarial attacks still pose a potential risk in our approach. For instance, some users may intentionally inject harmful instructions into the model. Without proper security mechanisms in place, such malicious content could be incorporated into the learning process of NTRL-Code, leading the model to be updated based on harmful on-the-fly inputs.