QiMeng-PRepair: Precise Code Repair via Edit-Aware Reward Optimization

Changxin Ke[1,2] Rui Zhang[1,2] Jiaming Guo[1,2] Yuanbo Wen[1,2] Li Ding[2,3] Shuo Wang[1,2]
Xuyuan Zhu[2] Xiong Peng[2] Di Huang[1,2] Zidong Du[1,2] Xing Hu[1,2] Qi Guo[1,2]
Yunji Chen[1,2]*
[1]State Key Lab of Processors, Institute of Computing Technology, CAS
[2]University of Chinese Academy of Sciences
[3]Institute of Microelectronics, CAS
Code Models

Abstract

Large Language Models (LLMs) achieve strong program repair performance but often suffer from *over-editing*, where excessive modifications overwrite correct code and hinder bug localization. We systematically quantify its impact and introduce *precise repair* task, which maximizes reuse of correct code while fixing only buggy parts. Building on this insight, we propose PRepair, a framework that mitigates over-editing and improves repair accuracy. PRepair has two components: *Self-Breaking*, which generates diverse buggy programs via controlled bug injection and min–max sampling, and *Self-Repairing*, which trains models with Edit-Aware Group Relative Policy Optimization (EA-GRPO) using an edit-aware reward to encourage minimal yet correct edits. Experiments show that PRepair improves repair precision by up to 31.4% under fix_1@1, a metric that jointly considers repair correctness and extent, and significantly increases decoding throughput when combined with speculative editing, demonstrating its potential for precise and practical code repair.

1 Introduction

Program repair aims to automatically correct faulty programs while preserving their intended semantics, and has become an important research area in the era of Large Language Models (Hui et al., 2024; Zhang et al., 2025; Guo et al., 2025). Prior works generally follow a structured paradigm, decomposing the task into stages such as error localization, correction, and validation (Xia et al., 2024; Ho et al., 2025; Epperson et al., 2025). With the growing use of coding assistants like Copilot and Cursor, there is an increasing need for fast, end-to-end program repair models. To address this demand, many recent approaches employ supervised fine-tuning (SFT) and reinforcement learning (RL) to train models capable of performing program repair accurately.

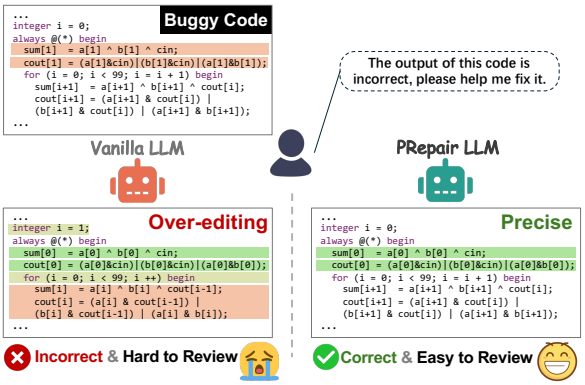


Figure 1: Existing models suffer from *over-editing*, which not only reduces repair accuracy but also significantly increases the review burden for developers. In comparison, PRepair improves both repair accuracy and maintainability in practice.

Most existing training approaches (Muennighoff et al., 2023; Hui et al., 2024; Yang et al., 2025; Fu et al., 2025) optimize repair correctness alone, treating code repair as a correctness-only objective. This formulation ignores how much the model modifies the original program. We observe that these models suffer from an *over-editing* phenomenon (as illustrated in Figure 1), where they tend to regenerate large portions of the code through excessive edits instead of understanding and minimally correcting the original buggy code. Over-editing is harmful for two reasons: (1) it fails to localize the bug, thereby limiting the effectiveness of the repair; and (2) it unnecessarily rewrites the code, breaking the original structure and reducing maintainability in practice. Therefore, for code repair, precise repair is preferred, as it maximizes the reuse of correct logic in the original code while precisely fixing the buggy parts, thereby preserving code logic and reducing developers' review burden. However, while precise repair is crucial for code repair, it remains largely unsolved in existing approaches.

*Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 29114–29129
July 2-7, 2026

Precise repair faces two key challenges: (1) Data scarcity. Effective repair requires models to understand the semantics of buggy programs, reuse their correct components, and precisely localize and fix errors. However, realistic buggy code that simultaneously contains substantial correct logic and localized faults is extremely scarce. (2) Preservation of correct code. During training, it is challenging to make the model aware of how much of the code has been edited, so that it preserves the correct parts while precisely localizing and fixing only the buggy portions.

To address the over-editing issue, we propose the **PRepair** framework, which explicitly guides models to perform precise repairs. Our central insight is that optimizing for minimal yet sufficient edits preserves repair correctness while encouraging faithful reuse of correct program logic. To address the two challenges of precise repair, the PRepair framework consists of two steps: (1) *Self-Breaking*, where we design a precise code repair data generation framework that systematically injects bugs into correct programs to construct large-scale training data, combined with a min–max sampling strategy to maximize the diversity of buggy programs while avoiding over-concentration on similar bug patterns; (2) *Self-Repairing*, where the model is optimized with proposing Edit-Aware Group Relative Policy Optimization (EA-GRPO) to encourage both correct and minimal code repairs. EA-GRPO introduces an edit-aware reward, where edit penalties are applied when the model achieves sufficient repair correctness. This design effectively balances repair correctness and extent, encouraging minimal yet accurate code fixes. Besides, to evaluate both repair correctness and the extent of modifications, we introduce $\text{fix}_p@k$, the first metric specifically designed for assessing precise repair, which jointly considers correctness and the number of edits.

Compared with previous methods that optimize code repair solely for correctness, our method offers two key advantages. First, the model learns to focus its attention on the buggy lines, acquiring an implicit error localization ability that guides precise repair, which not only improves repair accuracy but also enhances cross-domain code repair capability. Second, by following the logic of the buggy code, it reuses correct portions of the original program, alleviating the over-editing problem and improving maintainability in practice, as shown in Figure 1.

Experiments on two models of different sizes and two fundamentally different languages, Python and Verilog, show that PRepair effectively reduces unnecessary edits while improving repair correctness. In addition, when combined with speculative editing, PRepair enables faster inference, demonstrating its practical value and generality across diverse programming languages. The main contributions of this paper are as follows:

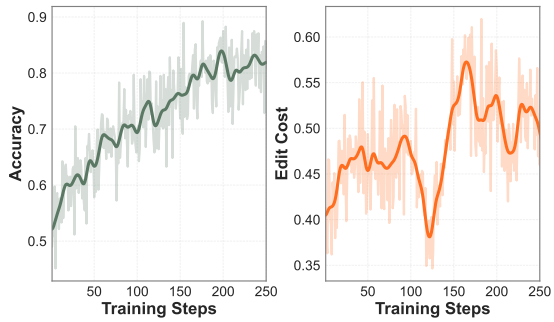
- We identify over-editing as a key issue in LLM-based code repair under GRPO and propose $\text{fix}_p@k$, the first metric for evaluating repair precision.
- We propose the PRepair framework to enhance code repair without labeled data, and introduce EA-GRPO to train models for precise code repair using an edit-aware reward.
- Empirical evaluations on multiple models and benchmarks demonstrate that PRepair achieves superior repair precision and correctness.
- When combined with speculative editing, EA-GRPO significantly increases inference throughput, highlighting the practical value of PRepair as real-world code assistance.

2 Methodology

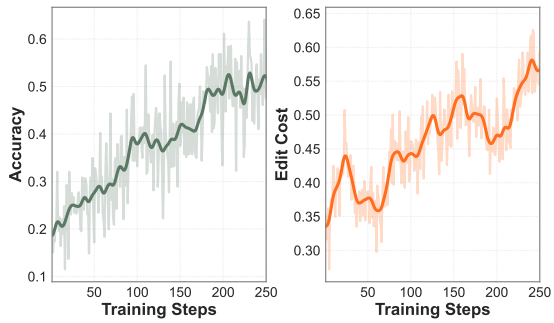
In this section, we first analyze existing models trained with naive GRPO and empirically study the relationship between repair accuracy and extent of modifications. Motivated by these findings, we introduce a novel metric, fix_p, which jointly measures repair accuracy and the number of edited lines. Based on this metric, we then present the proposed **PRepair** framework.

We model code as a sequence of lines $X = \{x_1, x_2, \ldots, x_n\}$. Given a buggy program, the goal of program repair is to perform the necessary line-level insertions, deletions, and substitutions to produce a corrected sequence $Y = \{y_1, y_2, \ldots, y_m\}$ that satisfies the intended functionality. To quantify the distance between the buggy code and the corrected code, we introduce the Edit Cost $\mathbf{D}_{\text{EC}}$, which is based on the Levenshtein distance $\mathbf{D}(X, Y)$ (Levenshtein, 1965). This distance measures the minimum number of insertions, deletions, and substitutions required to transform one code into the other. Let $|X|$ denote the number of lines in the source program. We define Edit Cost as:

$$\mathbf{D}_{\text{EC}}(X, Y) = \frac{\mathbf{D}(X, Y)}{|X|}$$



(a) Python Code Repair



(b) Verilog Code Repair

Figure 2: **GRPO training with correctness-only rewards.** For both Python and Verilog code repair tasks, although performance improves during training, the edit cost increases substantially, leading to a more severe over-editing phenomenon.

Here, dividing $|X|$ normalizes the edit distance by lines of buggy code, allowing fair comparison across programs of different sizes.

2.1 Observations

In this section, we explore the phenomenon of *over-editing* in LLMs and investigate the relationship between code repair accuracy and edit cost.

We conduct experiments on Python and Verilog code repair tasks. The Python dataset is collected from LeetCodeDataset (Xia et al., 2025), and the Verilog dataset is obtained from QiMeng-CodeV-R1 (Zhu et al., 2025). We design a reward that considers only repair correctness, and the model performance and edit cost are shown in the Figure 2. We find that as training progresses, repair correctness improves, but over-editing becomes increasingly severe. The model does not learn to fix errors precisely but instead makes large modifications to "hit" a correct solution. As training continues, the edit cost even exceeds 0.6, indicating that the model introduces extensive redundant changes without understanding the original buggy code and localizing the errors. These findings show that evaluating code repair solely based on correctness is insufficient, which motivates the need to design a metric that explicitly measures repair precision and to incorporate edit cost into training.

2.2 Metric Design

Considering the over-editing phenomenon, to better capture precise code repair capability, we propose $\text{fix}_p@k$, a novel metric that jointly accounts for repair correctness and edit cost. To reduce statistical bias, we adopt an unbiased estimation method by sampling n candidates. The computation of a general metric $(\cdot)@k$ is defined as:

$$(\cdot)@k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}},$$

where c denotes the number of samples that satisfy the corresponding *checking criterion* among the n generated candidates and k represents the number of candidates considered.

Given the golden fixed program Y and the model-generated fix Y', We define $\text{fix}_p@k$, where p denotes the ratio between the acceptable edit cost and the theoretical minimum Edit Cost, representing the tolerance level for repair cost in evaluation. The corresponding checking criterion is:

$$c = \sum_{i=1}^{n} \mathbb{I}\left[\text{correct}_i \ \wedge \ \left(\frac{\mathbf{D}_{\text{EC}}(X, Y')}{\mathbf{D}_{\text{EC}}(X, Y)} \leq p\right)\right].$$

Here, correct_i indicates that the i-th generated code passes all tests. We also report the correctness only metric using pass@k (Chen et al., 2021).

2.3 PRepair framework

Program repair is challenged by the lack of realistic buggy data with localized faults and by the difficulty of preserving correct code during repair. To address this challenge, we propose the PRepair framework, as shown in Figure 3, which consists of two stages: (1) Self-Breaking, where the model generates high-quality buggy code by itself without human annotations. (2) Self-Repairing, where the model is trained with EA-GRPO to improve its ability of precise code repair.

Self-Breaking. Given a program description and its corresponding golden code Y, we prompt the model to inject bugs (detailed prompt is in Appendix B) into Y and sample a set of buggy programs $\mathcal{X} = \{X_1, X_2, \ldots, X_m\}$. To improve computational efficiency while preserving bug diversity, we adopt a min-max sampling strategy.

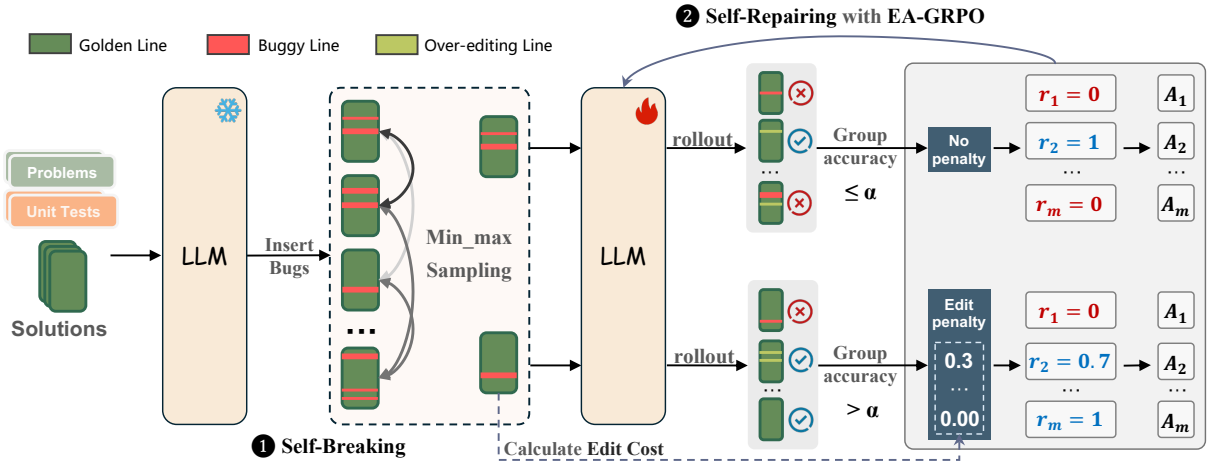


Figure 3: **Overview of the PRepair framework.** It consists of two stages: Self-Breaking, where the model injects diverse bugs into golden programs to construct high-quality buggy inputs, and Self-Repairing, where the model learns to precise repair these buggy programs via EA-GRPO which uses a dynamic edit-aware reward to encourage minimal yet correct edits.

Specifically, we select a subset $\mathcal{X}_s \subset \mathcal{X}$ by minimizing the maximum pairwise similarity among buggy samples, where similarity is defined as $1 - \mathbf{D}_{\text{EC}}(X_i, X_j)$. The selected subset is obtained by solving:

$$\mathcal{X}_s = \min_{\substack{\mathcal{X}' \subset \mathcal{X} \\ |\mathcal{X}'|=k}} \max_{\substack{X_i, X_j \in \mathcal{X}' \\ i \neq j}} \big(1 - \mathbf{D}_{\text{EC}}(X_i, X_j)\big).$$

This strategy encourages the sampled buggy programs to be maximally diverse in terms of edit distance, resulting in a more diverse and informative set of buggy programs for training.

Self-Repairing. Given a program description and its corresponding buggy code set $\mathcal{X}_s$ sampled from the Self-Breaking stage, the objective of this stage is to train the model to repair the buggy programs and improve its repair policy. Specifically, the model generates candidate repairs for each buggy input, and the policy is updated using the proposed *Edit-Aware Group Relative Policy Optimization (EA-GRPO)*. During optimization, rewards are computed with a dynamic edit-aware reward, which jointly considers repair correctness and edit cost to guide the model toward accurate and minimal code fixes.

2.4 EA-GRPO

Program repair differs from code generation. Using a binary reward based solely on correctness is insufficient because it cannot reflect the model's ability to precisely identify errors.

To address this, we design the EA-GRPO mechanism that encourages minimal and precise changes while ensuring correctness. Specifically, to avoid over-penalizing model edits that could harm correctness, the penalty in EA-GRPO is applied dynamically. It is triggered only when the model achieves sufficient group-level accuracy. Compared with naive GRPO (Shao et al., 2024) (details can be found at Appendix E), EA-GRPO introduces a dynamic edit-aware reward, focusing on balancing repair correctness and edit cost.

Group Accuracy Threshold. During training, given a buggy input $X_t \in \mathcal{X}_s$, we compute the average repair accuracy $\text{Acc}_{\mathcal{G}^t}$ of its rollout group $\mathcal{G}^t = \{o_1, o_2, \ldots, o_n\}$, where each o_i denotes a repaired code generated by the model. The edit penalty is activated only when the group-level accuracy exceeds a threshold α, formally defined as

$$\mathcal{T}(\mathcal{G}^t) = \begin{cases} 1, & \text{if } \text{Acc}_{\mathcal{G}^t} \geq \alpha, \\ 0, & \text{otherwise.} \end{cases}$$

Dynamic Edit-Aware Reward Shaping. For correct samples in groups that satisfy the accuracy threshold, we apply a standardized edit penalty to encourage correct repairs with minimal edit cost. Let $\mathcal{G}_c^t \subset \mathcal{G}^t$ denote the set of correct samples. The penalty for each sample $o_i \in \mathcal{G}_c^t$ is defined as

$$\mathcal{P}_i^{\mathcal{G}} = \sigma\left(\frac{\mathbf{D}_{\text{EC}}(X_t, o_i) - \text{mean}(\mathbf{D}_{\text{EC}}(X, \mathcal{G}_c^t))}{\text{std}(\mathbf{D}_{\text{EC}}(X, \mathcal{G}_c^t))}\right),$$

where $\text{mean}(\mathbf{D}_{\text{EC}}(X, \mathcal{G}_c^t))$ and $\text{std}(\mathbf{D}_{\text{EC}}(X, \mathcal{G}_c^t))$ are the mean and standard deviation of the edit cost for correct samples in the group. The outer sigmoid bounds the penalty while preserving the relative ordering of edit costs within the group.

Reward Design. The reward for each sample in the group is then defined as

$$\mathcal{R}_i^{\mathcal{G}} = \begin{cases} 1 - \mathcal{T}(\mathcal{G}) \cdot \beta \cdot \mathcal{P}_i^{\mathcal{G}}, & \text{if } o_i \text{ is correct,} \\ 0, & \text{if } o_i \text{ is incorrect.} \end{cases}$$

where β is a penalty coefficient controlling the strength of the edit penalty. Importantly, the computation of this reward function does not require the golden code, it only uses the edit cost between the buggy input X and the generated samples.

2.5 Speculative Edits

Speculative decoding (Xia et al., 2023) is widely used in code editing scenarios because the original code can be reused across successive edits. We adopt Prompt Lookup Decoding (Saxena, 2023), a speculative decoding method, to accelerate inference. Speculative decoding improves generation efficiency by first producing multiple draft tokens and then verifying them in parallel. Unlike conventional approaches that rely on a separate draft model, Prompt Lookup Decoding directly reuses the input prompt as the draft through n-gram matching, which is particularly well suited for code editing scenarios. For this reason, it is also referred to as *Speculative Edits*. Our work focuses on reducing the edit cost between the input buggy code and the output, which substantially increases the acceptance rate of speculative edits. A detailed theoretical derivation is provided in Appendix D. Given a speculative window of K draft tokens per decoding step, the decoding throughput T (tokens/s) can be expressed as

$$T \propto \frac{1 - (1 - \mathbf{D}_{\mathrm{EC}})^{K+1}}{\mathbf{D}_{\mathrm{EC}}}.$$

It shows that reducing the edit cost leads to a significant increase in throughput. Therefore, when applying speculative edits, a smaller edit cost directly translates to a larger speedup.

3 Experiment

3.1 Experimental Setup

3.1.1 Benchmarks

We form a code repair benchmark that spans multiple programming languages and paradigms and covers diverse real-world errors, enabling a comprehensive evaluation of model code repair capabilities. The statistics of the two benchmarks are shown in Table 5.

Python code repair. We follow HumanEvalFix (Muennighoff et al., 2023), which extends the original HumanEval benchmark. It provides buggy code functions with subtle errors and corresponding unit tests, and models are tasked with generating correct fixes. Bugs are manually introduced to original HumanEval solutions so that the code still runs but fails at least one test. The benchmark covers various types of logical errors, including missing logic, excess logic, and wrong logic such as value, operator, variable, or function misuse, totaling 164 buggy samples.

Verilog code repair. Existing Verilog code repair benchmarks (Tsai et al., 2024) have clear limitations. Most of them mainly target syntax errors and give little attention to functional errors. Our work aims to enable LLMs to reuse correct logic in buggy code and apply precise and minimal fixes. We systematically summarize common logical error patterns in Verilog from Tsai et al. (2024); Yao et al. (2024); Qiu et al. (2025) and prompt models to inject these bugs into correct code from the QiMeng-CodeV-R1 (Zhu et al., 2025) dataset. This process produces a diverse Verilog code repair benchmark with 352 samples.

3.1.2 Base model & Baselines

Models. We conduct experiments on Qwen2.5-Coder-3B and Qwen2.5-Coder-7B (Hui et al., 2024), two models of different scales, to evaluate the generality of our approach across model capacities.

Baselines. We compare our approach with several baselines. (1) **Prompt Engineering** instructs the model to perform code repair with minimal modifications via prompts. Specifically, we append the instruction "Please make sure to make minimal changes to the buggy code." at the end of the prompt. (2) **GRPO** performs reinforcement learning using the same training data, number of training steps, and hyperparameters as EA-GRPO. The only difference is that its reward function considers repair correctness only, without incorporating any edit-aware terms. (3) In addition, we evaluate two widely used commercial code assistant models, GPT4 (OpenAI et al., 2024) and Gemini2.0-flash (Team et al., 2025). For these strong proprietary models, we apply the same prompt engineering strategy to assess how much prompt-based guidance alone can improve repair precision.

Language	Method	pass@k			fix$_1$@k			fix$_{1.5}$@k			fix$_2$@k		
		1	5	10	1	5	10	1	5	10	1	5	10
Python	**GPT4**	84.51	94.68	96.95	52.20	71.50	76.83	53.29	72.81	78.66	60.73	79.85	84.76
	+ Prompt	86.89	95.90	96.95	62.56	80.28	85.98	63.90	81.36	86.59	72.44	87.72	90.24
	Gemini2.0-flash	90.73	91.94	92.07	44.88	47.71	48.78	47.38	49.98	51.22	55.12	58.60	60.37
	+ Prompt	92.20	93.28	93.29	63.78	66.45	67.07	64.70	67.67	68.29	71.71	74.95	75.61
	Qwen2.5-Coder-3B	53.78	82.22	86.48	33.72	55.91	60.93	34.18	56.68	62.16	38.78	63.99	69.74
	+ Prompt	50.91	57.34	63.18	32.13	57.34	63.18	32.32	57.74	63.73	36.83	65.44	71.92
	+ GRPO	**80.52**	**87.26**	**88.43**	34.27	40.37	41.66	36.01	41.67	42.89	45.61	52.71	54.67
	+ EA-GRPO	79.05	80.97	81.09	**67.96**	**70.67**	**71.03**	**67.96**	**70.67**	**71.03**	**74.36**	**77.22**	**77.43**
	Qwen2.5-Coder-7B	86.28	92.91	**93.79**	60.67	72.37	74.24	61.59	72.45	74.25	71.46	81.69	83.27
	+ Prompt	87.23	**92.37**	93.23	66.52	76.44	78.15	66.86	76.51	78.15	77.41	85.58	86.78
	+ GRPO	89.82	91.70	91.93	47.44	48.91	49.09	48.20	50.02	50.29	60.88	62.31	62.50
	+ EA-GRPO	**91.19**	92.90	93.22	**81.62**	**83.51**	**84.01**	**82.13**	**83.76**	**84.08**	**89.54**	**91.61**	**92.00**
Verilog	**GPT4**	69.52	85.44	89.49	2.30	4.69	5.97	3.84	7.76	9.38	5.77	11.65	14.20
	+ Prompt	55.99	79.83	84.38	22.13	44.84	52.56	28.04	53.49	61.65	33.92	58.85	65.34
	Gemini2.0-flash	56.65	69.05	72.44	19.06	23.25	24.43	24.01	30.44	32.39	30.57	38.15	40.06
	+ Prompt	68.01	74.46	76.99	42.33	48.09	50.00	48.44	54.34	56.25	53.49	59.02	61.08
	Qwen2.5-Coder-3B	45.91	59.93	63.57	34.08	50.06	54.57	36.14	52.14	56.50	39.91	55.63	59.66
	+ Prompt	44.53	57.55	60.52	34.20	50.54	54.54	36.18	51.90	55.57	39.52	54.65	57.96
	+ GRPO	47.90	61.08	65.44	18.55	33.64	39.34	21.52	37.41	43.51	28.65	44.52	50.10
	+ EA-GRPO	**52.64**	**66.49**	**69.93**	**37.40**	**54.03**	**58.15**	**40.80**	**56.79**	**60.83**	**45.30**	**60.29**	**63.69**
	Qwen2.5-Coder-7B	57.36	69.31	72.67	36.70	50.29	54.31	42.86	55.85	59.29	48.98	61.47	64.34
	+ Prompt	57.07	68.63	72.10	38.00	51.10	54.74	43.81	55.75	58.62	49.59	61.25	64.14
	+ GRPO	68.37	71.91	**72.89**	8.49	9.95	10.21	12.93	15.69	16.24	23.85	27.29	28.04
	+ EA-GRPO	**68.66**	**72.02**	72.75	**68.11**	**71.38**	**72.07**	**68.11**	**71.38**	**72.07**	**68.59**	**71.85**	**72.61**

Table 1: **Main results.** We report pass@k and fix$_p$@k results with $k \in \{1, 5, 10\}$ and $p \in \{1, 1.5, 2\}$. We evaluate GPT4 and Gemini2.0-flash with prompt engineering, as well as Qwen2.5-Coder-3B and Qwen2.5-Coder-7B under prompt engineering, GRPO, and our EA-GRPO. **Bold** indicates the best result, and underline indicates the second best in the same model.

3.1.3 Implementation Details

For Python code repair, we use the training data from (Xia et al., 2025), which contains 2,869 Python programming tasks crawled from LeetCode, each equipped with comprehensive test suites. In the Self-Breaking stage, we first prompt the model to sample $|\mathcal{X}| = 32$ buggy variants for each task, and then apply a min-max sampling strategy to reduce the number of samples to $|\mathcal{X}'| = 4$. We further filter out false buggy cases that still pass all test cases. This process results in a final dataset of 10,242 <program description, buggy code> pairs. For Verilog code repair, we use the training data from QiMeng-CodeV-R1 (Zhu et al., 2025), which contains 3,033 Verilog programming tasks, each provided with golden reference code and rule-based verification tools to validate the correctness of generated programs. Using the same parameters as in Python code repair, the Self-Breaking step yields 11,200 buggy code samples.

We conduct reinforcement learning training using the VeRL framework (Sheng et al., 2024). More details and training hyperparameters are provided in Appendix B.

3.2 Results and Analysis

Our main results and comparisons with the baselines are presented in Table 1 and Figure 4.

3.2.1 Main Results

Training is Necessary. As shown in Table 1, we report the results of applying prompt engineering to GPT4, Gemini2.0-Flash, Qwen2.5-Coder-3B, and Qwen2.5-Coder-7B. Our results reveal that prompt engineering introduces substantial uncertainty in model behavior. For Python code repair, this strategy has little impact on pass@k and yields only limited improvements in fix$_p$@k. In contrast, for Verilog code repair, prompt engineering significantly degrades performance of GPT4, reducing pass@1 by 13.53%. These observations indicate that prompt engineering is far less effective than EA-GRPO. GPT-4 and Gemini 2.0 Flash achieve substantially lower fix$_1$@1 than Qwen2.5-Coder-7B trained with EA-GRPO. On Python, their fix$_1$@1 is lower by 19.10% and 17.84%, respectively. On Verilog, the gap is even larger, with drops of 45.98% and 25.78%. These results show that training with EA-GRPO is necessary.

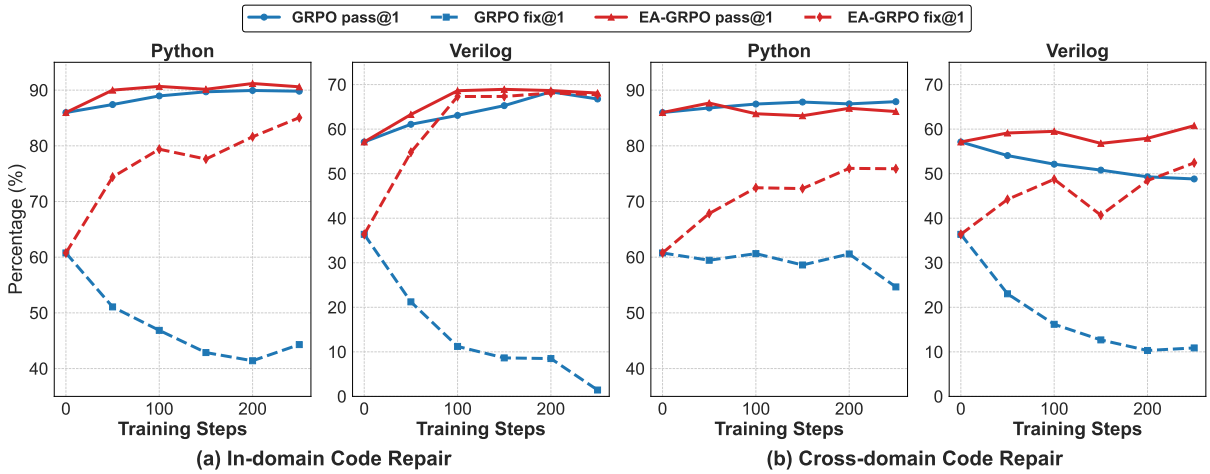


Figure 4: **Code repair performance of in-domain and cross-domain.** We plot the changes of pass@1 and fix_1@1 across training steps, reporting both in-domain and cross-domain performance. (a) In-domain: models are trained on Python data and evaluated on Python code repair; similarly for Verilog. (b) Cross-domain: models trained on Python data are evaluated on Verilog code repair, and vice versa.

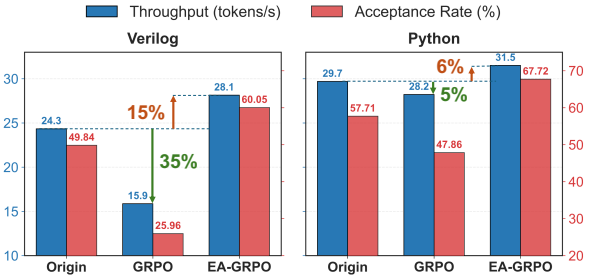


Figure 5: **Decoding Performance with Speculative Edits.** Throughput and acceptance rates of Origin (before training), GRPO, and EA-GRPO on Python and Verilog benchmarks, using buggy code as draft.

α	β	pass@1	pass@5	fix_1@1	$\text{fix}_{1.5}$@1	fix_2@1
0	0.05	90.73	92.66	**81.74**	**82.35**	88.93
0.5	0.2	85.06	85.36	80.27	80.27	84.09
0.8	0.05	**91.19**	**92.90**	81.62	82.13	**89.54**
0.8	0.25	88.78	91.22	77.93	78.54	86.62
1.1	/	89.82	91.70	47.44	48.20	60.88

Table 2: Ablation results of EA-GRPO on Qwen2.5-Coder-7B for Python code repair with varying α and β, reporting pass@1, pass@5, and fix_p@1 metrics.

Fewer Edits, More Correct Repairs. As shown in Table 1, under the fix_p metric, EA-GRPO substantially improves repair precision on both languages. Specifically, fix_1@1 increases by 20.95% on Python and by 31.41% on Verilog compared to the original model, significantly alleviating the over-editing phenomenon. In contrast, models trained with GRPO exhibit a severe degradation in fix_p. On Verilog, fix_1@1 drops sharply from 36.70% to 8.49%, and fix_2@1 decreases from 48.98% to 23.85%, reflecting pronounced over-editing behavior that substantially increases the code review burden for developers.

Notably, EA-GRPO also yields consistent gains in repair correctness in most settings. Compared with GRPO, Qwen2.5-Coder-7B trained with EA-GRPO improves pass@1 by 1.37% on Python and by 0.29% on Verilog, while Qwen2.5-Coder-3B achieves a 4.74% improvement in pass@1 on Verilog. These results indicate that explicitly encouraging fewer edits does not harm repair correctness; instead, it helps the model better understand the original program logic and more accurately localize bugs, leading to more effective repairs. In a small number of cases, such as Qwen2.5-Coder-3B on Python, pass@1 is slightly lower than that of GRPO (by 1.47%). However, this minor drop is accompanied by a substantial improvement in fix_p@k, demonstrating that EA-GRPO successfully balances repair correctness and edit cost.

We further present a case study in Appendix A and Figure 6. The results show that models trained with EA-GRPO generate fixes that better follow the logic of the buggy code, while placing substantially higher attention on the buggy lines. This indicates that the model learns to reuse the correct parts of the original program and precisely localize and repair the buggy components.

Better Cross-domain Generalization. We evaluate cross-domain generalization by assessing the Verilog code repair performance of models trained on Python and, conversely, the Python code repair performance of models trained on Verilog. As presented in Figure 4. We observe that in cross-domain settings, the models trained with EA-GRPO main-

tain stable repair correctness while significantly improving $\text{fix}_1@1$. In contrast, the models trained with GRPO exhibits a notable drop in $\text{fix}_1@1$, indicating increased edit cost, and its $\text{pass}@1$ is also unstable. For instance, when trained on Python data and evaluated on Verilog code repair, $\text{pass}@1$ of GRPO decreases from 57.12% to 48.81% (a drop of 8.31%), and $\text{fix}_1@1$ drops from 36.38% to 10.88% (a drop of 26.50%). This demonstrates that optimizing solely for correctness does not enable the model to generalize its understanding of code or to accurately localize bugs. By contrast, EA-GRPO encourages the model to reuse correct portions of the buggy code while precisely localizing errors, achieving better cross-domain generalization.

Faster Repair via Speculative Edits As shown by the throughput and acceptance rate in Figure 5 and Table 7, EA-GRPO substantially increases the draft token acceptance rate due to its significantly reduced edit cost, resulting in up to a 15% improvement in decoding throughput. In contrast, GRPO exacerbates over-editing, leading to a throughput degradation of up to 35%. These results demonstrate the practical significance of our method: when deployed in real-world code assistants, EA-GRPO can markedly improve online serving efficiency while maintaining high repair quality.

3.3 Ablation Study

To investigate the effectiveness of EA-GRPO, we conduct an ablation study on Qwen2.5-Coder-7B as shown in Table 2. Specifically, we vary the Group Accuracy Threshold α, which controls when the edit penalty is applied: $\alpha = 0$ applies the penalty to all correct samples, whereas $\alpha = 1.1$ uses only the correctness reward. We also experiment with different values of the penalty coefficient β. These ablations illustrate the impact of EA-GRPO on balancing repair correctness and minimal edits. In particular, increasing β may reduce $\text{pass}@1$, which in turn lowers $\text{fix}@k$. On the other hand, setting α too low penalizes all samples, causing the model to neglect correctness, while setting α too high prevents the model from learning precise repairs. Both extremes can degrade performance.

4 Related Work

Buggy Data Construction. In software, benchmarks for function-level code repair differ mainly in how buggy programs are generated. QuixBugs (Prenner and Robbes, 2021) contains only 40 programs, limiting coverage. DebugBench (Tian et al., 2024) injects bugs using LLMs and relies on online evaluation, which may not reflect realistic software defects. HumanEvalFix (Muennighoff et al., 2023) contains 164 tasks with human-injected bugs, better capturing real-world error patterns. We therefore adopt HumanEvalFix as our primary benchmark for Python code repair. In hardware, HLSdebugger (Wang et al., 2025) generates bugs with LLMs, but its data is not publicly available. RTLFixer (Tsai et al., 2024) collects buggy Verilog programs from LLM-generated incorrect solutions, but these often fail to retain substantial correct logic, limiting the study of precise repairs. We thus build our Verilog benchmark on QiMeng-CodeV-R1 (Zhu et al., 2025), which provides high-quality reference implementations and systematic verification.

LLMs for Code Repair. Prior LLM-based code repair approaches either use multi-stage pipelines, including error localization, correction, and validation (Xia et al., 2024; Epperson et al., 2025), or agent systems with RAG and external tools (Ho et al., 2025; Tsai et al., 2024). These methods are effective but often slow and costly. Another line of work trains LLMs end-to-end (Muennighoff et al., 2023; Hui et al., 2024; Yang et al., 2025; Fu et al., 2025; Xu et al., 2025), focusing primarily on functional correctness. In contrast, our approach explicitly targets both correctness and repair precision, which is crucial for realistic code repair.

5 Conclusion

In this work, we identify *over-editing* as a fundamental limitation of existing LLM-based code repair approaches that optimize correctness alone. We show that this issue not only increases review burden and harms maintainability, but also weakens error localization and degrades inference efficiency in practical settings. To address this, we propose PRepair, which explicitly encourages minimal yet sufficient edits through self-breaking data generation and the EA-GRPO training objective. Extensive experiments on Python and Verilog Benchmarks demonstrate that PRepair substantially improves repair precision, with $\text{fix}_1@1$ increasing by up to 34.24% while maintaining stable correctness, and when combined with Speculative Edits, it also accelerates inference, achieving up to 15% higher decoding throughput highlighting the practical as real-world code assistance.

Limitations

Although PRepair demonstrates effective precise repair performance across multiple programming languages, it still has the following limitations:

Metric of Repair Precision. We adopt fix_p as the metric of repair precision, which measures edit cost as the ratio of modified lines to the total number of lines in the buggy program. While this relative formulation ensures fairness across programs of different lengths, it implicitly assumes that the perceived cost of modification scales linearly with program size. In practice, this assumption may not always hold: modifying 1 line out of 3 and modifying 10 lines out of 30 yield the same edit ratio, yet the latter arguably imposes a heavier burden on code maintainers due to the larger absolute number of changes. Designing edit cost metrics that better reflect the non-linear relationship between program length and perceived repair precision is left for future work.

Automatic Hyperparameter Tuning. Although the ablation study demonstrates the effectiveness of the accuracy threshold and penalty coefficient in EA-GRPO, the optimal settings vary across datasets with different difficulty levels. We will explore automatic tuning methods under limited computational budgets in future work.

Application Scope. PRepair focuses on function-level code repair, where LLMs are used as coding assistants to perform precise fixes. In real-world software development, bugs may appear at the file level or even the project level, where high repair precision is also required. In such scenarios, models typically generate patches directly rather than rewriting entire files or projects, which naturally aligns with the objective of minimizing edit cost. Therefore, our method remains applicable in these broader settings, and we leave a systematic investigation of its effectiveness at the file and project level as future work.

Acknowledgments

This work is partially supported by the NSF of China (Grants No.U22A2028, 62525203, 62302483, 62341411, 6240073476), Science and Technology Major Special Program of Jiangsu (Grant No. BG2024028), Strategic Priority Research Program of the Chinese Academy of Sciences (Grants No.XDB0660300, XDB0660301, XDB0660302), CAS Project for Young Scientists in Basic Research (YSBR-029) and Youth Innovation Promotion Association CAS.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang (Eric) Zhu, and Saleema Amershi. 2025. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, page 1–15. ACM.

David Jiahao Fu, Aryan Gupta, Aaron Councilman, David Grove, Yu-Xiong Wang, and Vikram Adve. 2025. Slmfix: Leveraging small language models for error fixing with reinforcement learning. *Preprint*, arXiv:2511.19422.

Jiale Guo, Suizhi Huang, Mei Li, Dong Huang, Xingsheng Chen, Regina Zhang, Zhijiang Guo, Han Yu, Siu-Ming Yiu, Pietro Lio, and Kwok-Yan Lam. 2025. A comprehensive survey on benchmarks and solutions in software engineering of llm-empowered agentic system. *Preprint*, arXiv:2510.09721.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. 2025. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. *Preprint*, arXiv:2408.08927.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Vladimir I. Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin,

Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Julian Aron Prenner and Romain Robbes. 2021. Automatic program repair with openai's codex: Evaluating quixbugs. *Preprint*, arXiv:2111.03922.

Siyu Qiu, Muzhi Wang, Raheel Afsharmazayejani, Mohammad Moradi Shahmiri, Benjamin Tan, and Hammond Pearce. 2025. Towards llm-based root cause analysis of hardware design failures. *Preprint*, arXiv:2507.06512.

Apoorv Saxena. 2023. Prompt lookup decoding.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

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

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, and 1332 others. 2025. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *Preprint*, arXiv:2401.04621.

Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. 2024. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *Preprint*, arXiv:2311.16543.

Jing Wang, Shang Liu, Yao Lu, and Zhiyao Xie. 2025. Hlsdebugger: Identification and correction of logic bugs in hls code with llm solutions. *Preprint*, arXiv:2507.21485.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Heming Xia, Tao Ge, Peiyi Wang, Si-Qing Chen, Furu Wei, and Zhifang Sui. 2023. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. *Preprint*, arXiv:2203.16487.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. 2025. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. *Preprint*, arXiv:2504.14655.

Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the objective of llm-based program repair. *Preprint*, arXiv:2404.08877.

Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawende Bissyande, Claire Le Goues, and Shunfu Jin. 2025. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *ACM Transactions on Software Engineering and Methodology*.

Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. 2024. Hdldebugger: Streamlining hdl debugging with large language models. *Preprint*, arXiv:2403.11671.

Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2025. A systematic literature review on large language models for automated program repair. *Preprint*, arXiv:2405.01466.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. 2025. Qimeng-codev-r1: Reasoning-enhanced verilog generation. *Preprint*, arXiv:2505.24183.

A Case Study

A.1 Repair Cases

This task requires creating a function that takes a numeric value as a string and returns the nearest integer. When the number is exactly halfway between two integers, the function rounds it away from zero. For example, 14.5 rounds to 15, while -14.5 rounds to -15. The function must correctly handle both positive and negative numbers, as well as numbers with or without decimal points.

The buggy implementation carefully considers string inputs, removes trailing zeros, distinguishes positive and negative .5 values, and applies standard rounding for other numbers. However, it mistakenly rounds positive .5 down and negative .5 up, which is opposite to the intended "round away from zero" behavior.

The baseline GRPO method failed to understand the correct handling in the buggy code and instead rewrote the entire logic, introducing additional errors. Compared to the original buggy implementation, it ignores the careful string-based handling. It mishandles negative .5 values by rounding toward zero, relies on unstable floating-point comparisons, and does not account for empty-string inputs.

In contrast, our method correctly understood the proper handling in the buggy code. It precisely identified the issue of rounding positive .5 down and negative .5 up and made minimal modifications by replacing lines 12 and 14, achieving an accurate and efficient fix.

Buggy Code:

```
1  def closest_integer(value):
2      from math import floor, ceil
3
4      if value.count('.') == 1:
5          # remove trailing zeros
6          while (value[-1] == '0'):
7              value = value[:-1]
8
9      num = float(value)
10     if value[-2:] == '.5':
11         if num > 0:
12             res = floor(num)
13         else:
14             res = ceil(num)
15     elif len(value) > 0:
16         res = int(round(num))
17     else:
18         res = 0
19
20     return res
```

Baseline GRPO Results:

```
1  import math
2
3  def closest_integer(value):
4      num = float(value)
5      if num - math.floor(num) ==
           0.5:
6          return math.ceil(num)
7      elif math.ceil(num) - num ==
           0.5:
8          return math.floor(num)
9      else:
10         return round(num)
```

EA-GRPO Results:

```
1  def closest_integer(value):
2      from math import floor, ceil
3
4      if value.count('.') == 1:
5          # remove trailing zeros
6          while value[-1] == '0':
7              value = value[:-1]
8
9      num = float(value)
10     if value[-2:] == '.5':
11         if num > 0:
12             res = ceil(num)
13         else:
14             res = floor(num)
15     elif len(value) > 0:
16         res = int(round(num))
17     else:
18         res = 0
19
20     return res
```

A.2 Comparison of Attention Score Heat Map

To analyze how models specifically attend to repairing the input buggy code, we compute a word-level attention score matrix from the model's token-level attention, using the example in Appendix A.1.

Let the input prompt tokens be $x = \{x_1, \ldots, x_n\}$ and the generated output tokens be $y = \{y_1, \ldots, y_m\}$. Denote the model's token-level attention from output to input at layer l as $A \in \mathbb{R}^{m \times n}$, where A_{ij} represents how much output token y_i attends to input token x_j.

Since tokens may correspond to multiple subword pieces, we first group tokens into words. Let $M^{\text{in}} \in \mathbb{R}^{n \times N}$ be the input token-to-word mapping, where N is the number of input words, and $M^{\text{out}} \in \mathbb{R}^{m \times M}$ the output token-to-word mapping for M output words. Each entry is normalized by the number of tokens in the corresponding word. Then the word-level attention matrix $W \in \mathbb{R}^{M \times N}$ is computed as:

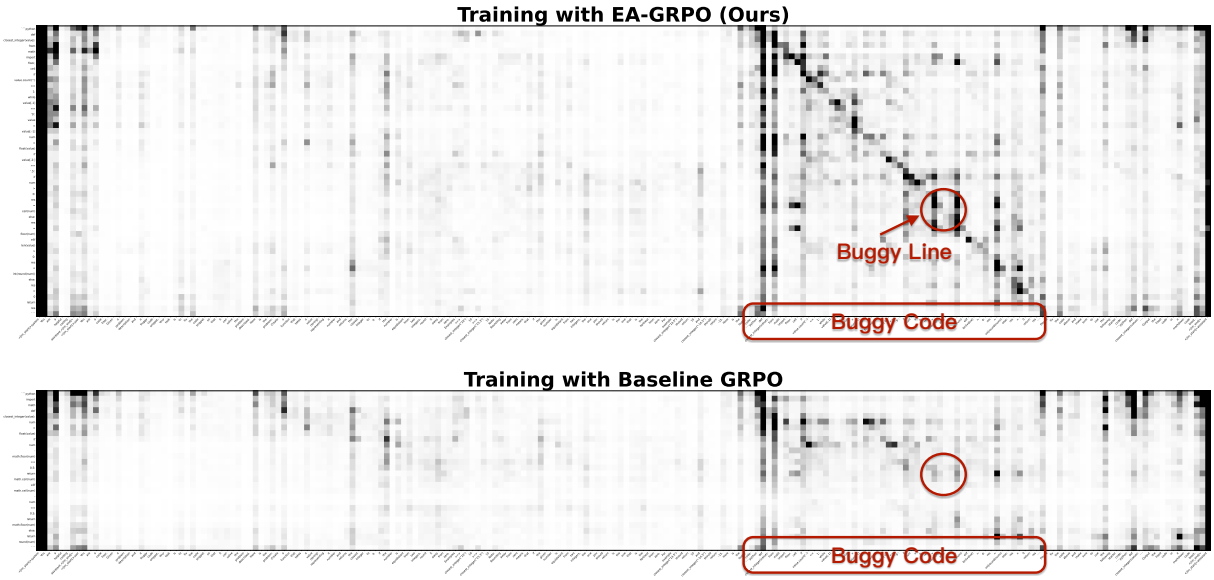


Figure 6: **Comparison of attention scores in code repair.** The top figure shows the PRepair model trained with EA-GRPO, and the bottom figure shows the model trained with GRPO using correctness-only rewards. The vertical axis corresponds to output tokens, the horizontal axis corresponds to input tokens, and the color intensity indicates the relative magnitude of the attention score.

$$W = (M^{\text{out}})^\top \, A \, M^{\text{in}}$$

Here, W_{ij} represents how strongly output word i attends to input word j. Extreme values are clipped at the 98th percentile to improve visualization contrast.

Using this method, we compute and visualize attention matrices for two models:

1. **Ours**: trained with EA-GRPO.
2. **Baseline**: trained with GRPO.

For comparison, the heatmaps in Figure 6 are plotted vertically, with output words on the vertical axis, input words on the horizontal axis, and color intensity representing the attention scores.

B Implementation Details

B.1 Training Setup

All RL training experiments are conducted on 8 A100-80GB SXM GPUs for the 7B model and on 8 L40S-48GB GPUs for the 3B model. The training hyperparameters are summarized in Table 3.

B.2 Inference & Evaluation

To reduce statistical bias, we adopt the unbiased estimation method described in Section 2.2. We set $n = 20$ during evaluation to compute $(\cdot)$@1, $(\cdot)$@5, and $(\cdot)$@10.

B.2.1 Inference parameters

For local models, we perform inference using vLLM, with the inference hyperparameters summarized in Table 4.

Max tokens	Temperature	Top-p	Top-k	Min-p
2048	0.7	0.8	20	0

Table 4: Inference sampling parameters used for local models.

B.2.2 Robust Evaluation for Edit Cost

Some models may introduce additional comments or reformat the code during repair, which can significantly inflate the measured edit cost and lead to unstable and unfair evaluation. To mitigate this issue, for Python code, we parse the programs into AST and remove all comments as well as redundant whitespace and line breaks before computing the edit cost. Similarly, for Verilog, we use iverilog[1] to obtain an AST-based representation and eliminate non-semantic characters. This preprocessing ensures that the edit cost reflects only semantic code changes, leading to a fair and consistent evaluation across models.

B.3 Statistics of Benchmarks

We summarize the bug types in the two benchmarks in Table 5. The results show that the bench-

[1]https://github.com/steveicarus/iverilog

Category	Parameter	Value	Parameter	Value
Algorithm	Advantage Estimator	GRPO	Normalize Advantage	True
	Use KL in Reward	False	KL Penalty Type	fixed
	KL Coefficient	0.001	Target KL	0.1
Policy Optimization	Learning Rate	1×10^{-6}	PPO Epochs	1
	Clip Ratio	0.2	Loss Aggregation	token-mean
	Entropy Coefficient	0.0	Use KL Loss	True
	KL Loss Coefficient	0.001	KL Loss Type	low_var_kl
Batch & Token Control	Train Batch Size	64	PPO Mini-batch Size	64
	PPO Micro-batch / GPU	2	Max Tokens / GPU	16384
Rollout Configuration	Rollout Engine	vLLM	Rollout Samples (N)	8
	Temperature	1.0	Top-p	1.0
	Top-k	-1	Prompt Length	2048
	Response Length	1024	Sampling Mode	stochastic
Length Control	Filter Overlong Prompts	True	Truncation Strategy	error
Distributed Training	Number of Nodes	1	GPUs per Node	8

Table 3: **RL Parameter Setting.** For both the correctness-only reward setting and our PRepair method, we use the same RL hyperparameters to ensure a fair comparison.

marks cover a wide range of bug categories and subtypes, including diverse logical errors commonly observed in real-world programs.

Language	Bug Category	Subtype	Count
Python	Missing Logic	Missing logic	33
		Excess logic	31
	O/V Misuse	Value misuse	44
		Operator misuse	25
	Wrong Logic	Variable misuse	23
		Function misuse	8
		Total	**164**
Verilog	Data-related	Bitwise error	54
		Value error	73
		Width error	137
		Arithmetic error	51
		Data error	5
	Control-related	Comparison error	12
		Assignment error	9
		Sensitivity list error	3
		State error	4
		Condition error	4
		Total	**352**

Table 5: Statistics of bug types in the Python and Verilog code repair benchmarks.

C Token-Level vs. Line-Level Edit Distance

Our $\text{fix}_p@k$ metric is built on line-level edit distance. This choice is deliberate and task-aligned, and we further explore a token-level variant to verify that our conclusions are robust to the granularity of the edit cost.

Semantic consistency. Compared with the commonly used token-level edit distance, line-based edit distance better preserves consistency in semantic importance. Token-level edit distance is often too fine-grained and can underestimate semantic changes. For example, replacing `a = b` with `a = c` changes only one token out of three at the token level, yet this modification completely alters the assignment semantics. At the line level, the edit cost is one full line, which more faithfully reflects the actual impact of the change.

Alignment with real-world development. A central motivation of our work is to reduce developers' review burden. In real workflows, code changes are inspected at the line level: tools such as `git diff` and Unix `diff` report modifications line by line, and code review is conducted line by line. Developers do not review code at the token or AST level. Line-based edit cost is therefore more consistent with practical usage scenarios.

Empirical comparison. We additionally evaluate Verilog baselines under a token-level version of $\text{fix}_p@1$. As shown in Table 6, the performance trends under token-level and line-level metrics

Method	line-level			token-level		
	fix_1@1	$\text{fix}_{1.5}$@1	fix_2@1	fix_1@1	$\text{fix}_{1.5}$@1	fix_2@1
GPT4	2.30	3.84	5.77	4.83	7.95	13.18
+ Prompt	22.13	28.04	33.92	24.94	33.15	38.01
Gemini2.0-flash	19.06	24.01	30.57	19.72	31.28	35.34
+ Prompt	42.33	48.44	53.49	45.14	54.26	57.27
Qwen2.5-Coder-7B	36.70	42.86	48.98	43.49	47.23	49.69
+ Prompt	<u>38.00</u>	<u>43.81</u>	<u>49.59</u>	<u>44.19</u>	<u>48.14</u>	<u>50.30</u>
+ GRPO	8.49	12.93	23.85	16.34	30.80	38.42
+ EA-GRPO	**68.11**	**68.11**	**68.59**	**67.66**	**68.39**	**68.59**

Table 6: **Line-level vs. token-level edit distance on Verilog.** We report fix_p@1 with $p \in \{1, 1.5, 2\}$ under both granularities. **Bold** indicates the best result, and <u>underline</u> indicates the second best in the same model. The ranking of methods is fully consistent across the two granularities.

Lang	Method	TPS	Draft	Acc.	AR
Verilog	Origin	24.34	229,650	114,467	0.498
	EA-GRPO	**28.15**	214,404	128,752	**0.601**
	GRPO	15.87	295,744	76,786	0.260
Python	Origin	29.70	13,404	7,735	0.577
	EA-GRPO	**31.51**	13,065	8,847	**0.677**
	GRPO	28.22	14,105	6,751	0.479

Table 7: Decoding performance with N-gram speculative decoding. TPS denotes throughput (tokens/s), Acc. denotes accepted tokens, and AR denotes acceptance rate.

are fully consistent: EA-GRPO remains the best method by a large margin under both granularities, while vanilla GRPO remains the weakest on the fix metric.

D Speculative Edits

To analyze the acceleration benefits of our method, we provide an analytical approximation that relates the program repair objective to the efficiency of Prompt Lookup Decoding under conservative assumptions. Prompt Lookup Decoding retrieves N-gram matches from the prompt at each decoding step and use them as draft tokens.

D.1 Acceptance Derivation

Let a buggy program be represented as a sequence of lines $X = \{x_1, x_2, \ldots, x_n\}$, where $n = |X|$ denotes the total number of lines. The repair process produces a corrected program $Y = \{y_1, y_2, \ldots, y_m\}$. Our EA-GRPO objective explicitly minimizes the normalized Distance Edit Cost, denoted as $\mathbf{D}_{\text{EC}}(X, Y)$, which measures the fraction of modified lines between X and Y.

For a given program X, the expected number of modified lines M is approximated as:

$$M = |X| \cdot \mathbf{D}_{\text{EC}}(X, Y). \tag{1}$$

In N-gram speculative decoding, draft tokens are obtained by performing an N-gram lookup over the input prompt (the buggy code X). For analytical tractability, we adopt a conservative approximation where draft tokens are aligned and verified at the line level: a line contributes to successful speculative acceptance only if it remains unchanged in the repaired output. Under this assumption, the probability that a randomly selected line is accepted, denoted as R_{line}, is given by:

$$R_{\text{line}} = \frac{|X| - M}{|X|} = 1 - \mathbf{D}_{\text{EC}}(X, Y). \tag{2}$$

Although speculative decoding operates at the token level, this approximation captures the dominant behavior in code repair, where edits typically disrupt token continuity within modified lines. Therefore, we approximate the token-level acceptance rate R by the line-level acceptance ratio:

$$R \approx R_{\text{line}} = 1 - \mathbf{D}_{\text{EC}}(X, Y). \tag{3}$$

This relation indicates that the speculative acceptance rate is inversely correlated with the edit cost. By explicitly minimizing $\mathbf{D}_{\text{EC}}(X, Y)$, EA-GRPO effectively increases R, transforming the input buggy program into a high-fidelity implicit draft for speculative decoding.

D.2 Throughput Derivation

Given a speculative window of K draft tokens, we analyze the expected number of tokens generated per target model verification step. Let the random variable X denote the number of tokens accepted before the first mismatch, where $X \in \{1, 2, \ldots, K+1\}$. Specifically, $X = i+1$ if the first i draft tokens are accepted and the $(i+1)$-th token is rejected, except for the case where all K draft tokens are accepted.

Under the assumption that each draft token is independently accepted with probability R, the probability mass function is:

$$P(X = i+1) = \begin{cases} R^i(1-R), & 0 \le i < K, \\ R^K, & i = K. \end{cases} \tag{4}$$

The expected number of tokens produced per verification step is:

$$\begin{aligned}
&\mathbb{E}[X] \\
&= \sum_{i=0}^{K-1} (i+1)\, R^i(1-R) + (K+1)R^K \\
&= (1-R) \sum_{i=0}^{K-1} (i+1) R^i + (K+1)R^K \\
&= (1-R) \sum_{j=1}^{K} j R^{j-1} + (K+1)R^K \\
&= (1-R)\, \frac{d}{dR} \left(\sum_{j=0}^{K} R^j \right) + (K+1)R^K \\
&= (1-R)\, \frac{d}{dR} \left(\frac{1-R^{K+1}}{1-R} \right) + (K+1)R^K \\
&= (1-R)\, \frac{-(K+1)R^K(1-R) + (1-R^{K+1})}{(1-R)^2} \\
&+ (K+1)R^K \\
&= \frac{1-(K+1)R^K + KR^{K+1}}{1-R} + (K+1)R^K \\
&= \frac{1-R^{K+1}}{1-R}.
\end{aligned}$$

Substituting the approximation $R \approx 1 - \mathbf{D}_{\mathrm{EC}}(X, Y)$, we obtain:

$$E \approx \frac{1-(1-\mathbf{D}_{\mathrm{EC}}(X,Y))^{K+1}}{\mathbf{D}_{\mathrm{EC}}(X,Y)}. \tag{5}$$

Since the N-gram lookup latency is negligible compared to the target model verification cost, the system throughput (measured as tokens per second) scales proportionally with E. Relative to the baseline decoding scheme where $E = 1$, the throughput improvement factor is therefore approximately:

$$T \propto \frac{1-(1-\mathbf{D}_{\mathrm{EC}})^{K+1}}{\mathbf{D}_{\mathrm{EC}}}. \tag{6}$$

We consider the throughput function

$$T \propto f(D) = \frac{1-(1-D)^{K+1}}{D}, \quad D \in (0,1). \tag{7}$$

Taking the derivative with respect to D gives

$$\begin{aligned}
&f'(D) = \\
&\frac{D(K+1)(1-D)^K - \big(1-(1-D)^{K+1}\big)}{D^2}.
\end{aligned}$$

The numerator can be simplified as

$$\begin{aligned}
g(D) = (K+1)D(1-D)^K - 1 + (1-D)^{K+1} \\
< 0, \quad \forall D \in (0,1),
\end{aligned}$$

which implies $f'(D) < 0$. Therefore, $f(D)$ is strictly decreasing with D, i.e.,

$$\text{as } \mathbf{D}_{\mathrm{EC}} \text{ decreases, } T \text{ increases.} \tag{8}$$

This analysis shows that as EA-GRPO reduces the edit cost, the system transitions into a high-efficiency regime where the expected token yield grows non-linearly with decreasing $\mathbf{D}_{\mathrm{EC}}$. This theoretical trend is consistent with our empirical observations in Figure 5 and Table 7, where EA-GRPO significantly improves both acceptance rate and end-to-end decoding throughput.

E Preliminary of GRPO

Group Relative Policy Optimization (GRPO) is an on-policy reinforcement learning algorithm built upon the Proximal Policy Optimization (PPO) framework. GRPO removes the value model to significantly reduce inference cost, while introducing group relative advantage estimation to more accurately assess the quality of model outputs. Furthermore, a KL-divergence penalty is incorporated to stabilize policy updates and prevent the policy from deviating excessively against the reference model.

Given a group $\mathcal{G}$ with rewards $\{\mathcal{R}_i^{\mathcal{G}}\}_{i \in \mathcal{G}}$, the group-normalized advantage is computed as

$$\mathcal{A}_i^{\mathcal{G}} = \frac{\mathcal{R}_i^{\mathcal{G}} - \mathrm{mean}\big(\mathcal{R}_j^{\mathcal{G}}\big)}{\mathrm{std}\big(\mathcal{R}_j^{\mathcal{G}}\big)}$$

The computed advantage is broadcast to all tokens of the corresponding output. Model parameters are updated using the GRPO objective with a KL divergence constraint:

$$\mathcal{J}(\theta) = \mathbb{E}\Bigg[\frac{1}{|\mathcal{G}|}\sum_{i\in\mathcal{G}}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min\Big(r_{i,t}(\theta)\mathcal{A}_i^{\mathcal{G}},$$

$$\mathrm{clip}\big(r_{i,t}(\theta),1-\epsilon,1+\epsilon\big)\mathcal{A}_i^{\mathcal{G}}\Big) - \gamma\mathrm{KL}(\pi_\theta\|\pi_{\theta_{\mathrm{old}}})\Bigg]$$

where

$$r_{i,t}(\theta) = \frac{\pi_\theta(o_{i,t} \mid x, o_{i,<t})}{\pi_{\theta_{\mathrm{old}}}(o_{i,t} \mid x, o_{i,<t})}$$

is the importance sampling ratio at token t, and γ controls the strength of the KL regularization.

F Prompts

In this section, we detail the prompt utilized in the process of Self-Breaking and Self-Repairing.

The following is the prompt we use for Self-Breaking.

> **Prompt:** You are a code breaker. Your task is to subtly introduce bugs into the provided code while preserving its syntactic correctness and overall structure.
> You may modify, replace, or delete lines to insert bugs that cause execution failures or incorrect results. Make sure your bugs are not obvious and are challenging to detect and fix. First, briefly explain the bugs you introduced. Then, output only the modified code, wrapped in a code block, without any additional comments.
> **Please keep the code format unchanged and only insert the necessary modifications.**
> Here is the golden code to break:
> ```{language}
> {code}
> ```
> Here is the problem associated with the code:
> {problem}

The following is the prompt we use for Self-Repairing.

> **Prompt:** You are a code fixer. Given a problem description and a buggy code snippet, Your task is to fix the code snippet so that it can solve the problem described.
>
> Here is the problem:
> {problem}
>
> Here is the buggy code:
> ```{language}
> {buggy_code}
> ```
> Output the fixed code in a markdown code block.