

# RePair: Automated Program Repair with Process-based Feedback

Yuze Zhao<sup>1</sup>, Zhenya Huang<sup>1,2\*</sup>, Yixiao Ma<sup>1</sup>, Rui Li<sup>1</sup>, Kai Zhang<sup>1</sup>,  
Hao Jiang<sup>1</sup>, Qi Liu<sup>1,2</sup>, Linbo Zhu<sup>1,2</sup>, Yu Su<sup>2,3</sup>

<sup>1</sup>State Key Laboratory of Cognitive Intelligence, University of Science and Technology of China

<sup>2</sup>Institute of Artificial Intelligence Comprehensive National Science Center

<sup>3</sup>School of Computer Science and Artificial Intelligence, Hefei Normal University, China

{yuzezhao, mayx, ruili2000, kkzhang0808, jianghao728}@mail.ustc.edu.cn

{huangzhy, kkzhang08, qiliuql}@ustc.edu.cn

lbzhu@iai.ustc.edu.cn

yusu@hfnu.edu.cn

## Abstract

The gap between the trepidation of program reliability and the expense of repairs underscores the indispensability of Automated Program Repair (APR). APR is instrumental in transforming vulnerable programs into more robust ones, bolstering program reliability while simultaneously diminishing the financial burden of manual repairs. Commercial-scale language models (LM) have taken APR to unprecedented levels. However, the emergence reveals that for models fewer than 100B parameters, making single-step modifications may be difficult to achieve the desired effect. Moreover, humans interact with the LM through explicit prompts, which hinders the LM from receiving feedback from compiler and test cases to automatically optimize its repair policies. In this literature, we explore how small-scale LM (less than 20B) achieve excellent performance through process supervision and feedback. We start by constructing a dataset named CodeNet4Repair, replete with multiple repair records, which supervises the fine-tuning of a foundational model. Building upon the encouraging outcomes of reinforcement learning, we develop a reward model that serves as a critic, providing feedback for the fine-tuned LM’s action, progressively optimizing its policy. During inference, we require the LM to generate solutions iteratively until the repair effect no longer improves or hits the maximum step limit. The results show that process-based not only outperforms larger outcome-based generation methods, but also nearly matches the performance of closed-source commercial large-scale LMs.<sup>1</sup>

## 1 Introduction

Since the birth of the program, its reliability has been a primary concern. The capacity of large language models to auto-generate code has further

\*Corresponding author

<sup>1</sup>Code and data are publicly available at <https://github.com/TnTWoW/RePair>

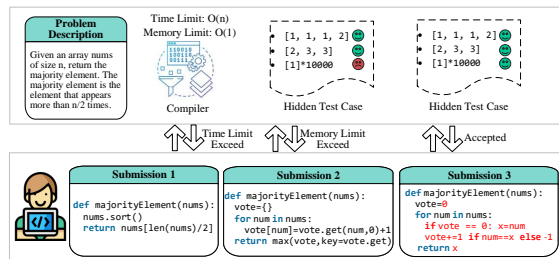


Figure 1: The general procedure for competitors to refine a solution on programming contest platform. Initially, they draft a solution based on the problem description and additional constraints such as time and memory limits. They then progressively improve their solution using feedback from the platform, like exceeding time or memory limits, until they achieve an accepted result.

intensified these concerns (Khoury et al., 2023), sparking discussions on possible solutions. Program repair accepts vulnerable programs as input, enhancing them through locating, correcting, and testing - a process that is often challenging within software development and programming competition. The confrontation arises when these defective programs encounter errors only when they receive rare and unexpected inputs. Moreover, it requires experienced programmers to invest substantial time in identifying and rectifying errors. A sophisticated Automated Program Repair (APR) system can reduce the expertise threshold and time commitment required, significantly enhancing the robustness of the program.

Prior works viewed APR as a sequence-to-sequence task, then LLMs, with their massive model parameters and training data, can address this issue through zero- or few-shot learning (Jiang et al., 2021; OpenAI, 2023; Ouyang et al., 2022; Chen et al., 2021; Zeng et al., 2022; Touvron et al., 2023; Hoffmann et al., 2022; Anil et al., 2023; Chen et al., 2022; Le et al., 2022; Rozière et al., 2024; Guo et al., 2024; Wei et al., 2023; Zhang et al., 2023). Whether it’s traditional methods or

LLMs, they achieve the repair by making a single modification. These outcome-based supervision methods pose a stringent challenge to the model due to the substantial edit distance between input and output. Simultaneously, the one-step modification approach does not align with human behavioral patterns: To be specific, when facing complex tasks, programmers usually repair the buggy program step by step in a cycle of modification, testing, and feedback. This inspires us to pay attention to the exploration of modification processes in APR tasks. Unlike *outcome-based* supervision, this *process-based* method requires guidance and supervision at each modification step.

Figure 1 depicts the typical program repair process in programming competitions, which is guided by compilers and test cases: Competitors initially construct a sketch of the program based on the problem’s intent and other constraints. They then complete the repair through a continuous process of submission, feedback, and interaction. Specifically, they initially attempt to solve “Find the majority element” problem by sorting and selecting the middle element. However, when they encounter time constraints, they shift to calculating each element’s frequency. But this method still requires  $O(n)$  space complexity. After receiving “Memory Limit Exceed” status, they ultimately use the Boyer-Moore majority vote algorithm with  $O(1)$  space complexity.

This example illustrates the characteristics of *process-based* program repair, which necessitates ongoing interaction with the compiler and test cases for feedback-guided repair. However, applying this type of process-based feedback on LMs appears unfeasible. First and foremost, the primary obstacle hindering related research is the absence of process-based datasets in practical scenarios. Second, the supervision method for intermediate processes in APR tasks are still under exploration. Previous work explored process-based supervision in mathematical problem solving and reasoning tasks (Lightman et al., 2023; Liu et al., 2023a), which required the intermediate steps to be correct. However, the intermediate steps in APR tasks serve as incorrect supervision signals and cannot be directly used for training. Last, interaction with LMs can be achieved through explicit prompt engineering. Attempts to use prompt engineering instead of compiler and test cases for feedback hinder the precise construction of human intentions by LMs.

In this work, to the best of our knowledge, we

	DeepFix	Review4Repair	Bug2Fix	CodeNet4Repair
Language	C	Java	Java	Python
Test Cases	×	×	×	✓
Repair Form	single step	single step	single step	multi step
Problem Description	×	×	×	✓
Test Size	6,971	2,961	58,356,545	10,144

Table 1: A comparison between the CodeNet4Repair dataset and existing datasets for program repair. The advantages of CodeNet4Repair stem from its comprehensive inclusion of test cases, problem descriptions, and detailed repair steps. CodeNet4Repair’s test set contains 10,144 complex program repair processes at competition level.

conduct the first few comprehensive exploration of process-based feedback with LMs in APR task. For this, we first establish a multi-step program repair dataset called CodeNet4Repair. Following that, we introduce a process-based feedback APR framework called **RePair**. RePair includes two models: a reward model and a repair model. We start by training a reward model to mimic the compiler as a virtual tool. It takes program text as input and gives assertions about the program’s status. The repair model works on buggy programs, completing one repair in a step. It then offloads the assessment of the program’s status to the virtual tool and waits for feedback to adjust the strategy for the next modification. Finally, we use  $\text{pass}@k$  as a metric to objectively assess the quality of program repair.

## 2 Data Collection

As far as we know, there has not been a dataset established for program repair tasks that includes processes. Thus we attempt to develop a procedural dataset specifically for these tasks. Overall, the dataset includes problem description, memory and time limit of the problem, repair process (a series of programs from error to correctness) and resource usage during execution (memory usage, CPU time, and code size). We derive our data from the large-scale programming competition dataset CodeNet (Puri et al., 2021). We make every effort to cleanse and filter this data to guarantee its quality. We clarify problem description, clean up the program from preliminary to fine, and collect additional high-quality test cases. Finally, we organize the program into a procedural format. In Table 1, we compare other excellent datasets and showcased the unique aspects of CodeNet4Repair (Gupta et al., 2017; Huq et al., 2020; Tufano et al., 2019).

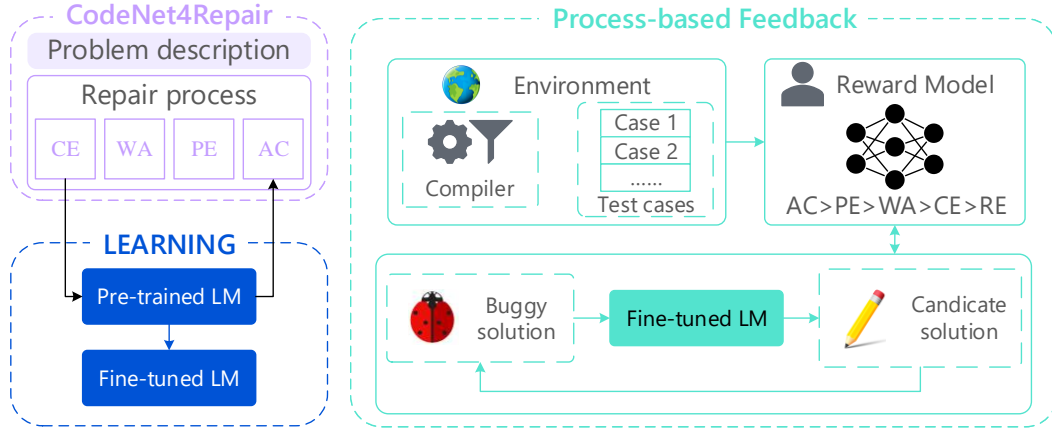


Figure 2: An illustration of process-based Automated Program Repair with compiler and test case feedback: (1) The introduction of a clean, privacy-protected dataset called CodeNet4Repair. (2) The application of Supervised Fine-Tuning (SFT) on pre-trained language models. (3) The incorporation of process-based feedback via reinforcement learning (RL). This process includes: establishing a reward model as a critic, and LM adjusts its repair policies based on the feedback from the critic. SFT and RL are both trained on CodeNet4Repair training set.

## 2.1 Problem Description Collection

Even professional programmers need to understand the problem that the program is intended to solve when fixing it. CodeNet provides unprocessed HTML for storing the description of each question. We utilize regular matching to extract these descriptions from the CodeNet’s HTML files. Finally, we manually supplement any descriptions that were challenging to recognize with Optical Character Recognition.

## 2.2 Program Preliminary Filtering

We chose Python as our benchmark language. We design the following rules for preliminary filtering.

- (1) No duplicate submissions. Casual or copy submissions do not aid in program repair process.
- (2) No malicious submissions, which include automated disruptive attempts, invalid code aimed at attacking the platform’s codebase, and destructive codes designed to interfere with system files.
- (3) No privacy breach submissions. Some IDEs automatically generate comments that could reveal the author’s identity.
- (4) Ensure each repair process includes at least one acceptable commit. The processes without any accepted outcomes will be excluded.

After this preliminary filtering, we obtain 1,227,259 submission programs. All non-English characters from the code comments are removed prior to fine filtering.

## 2.3 Program Fine Filtering

Although the original CodeNet dataset provides the program execution status (e.g., Wrong Answer or Accepted), these may vary due to differences in Python versions and environments. We set up a standard Python 3.11.3 environment and are meticulously testing each of the 1,227,259 execution results using test cases to ensure consistent status. Ignoring extra expenses such as process switching and assuming an average execution time of 4 seconds per solution, we would require a total of  $\frac{4 \times 1227259}{60 \times 60} = 1364$  (Core · hours) CPU time. Finally, we obtain 278,408 consistent programs. We arrange the answer records of each user for each question in chronological order, ensuring that the last program is “Accepted”.

To enhance evaluations precision, we expand the number of high-quality test cases collected online and annotated by hand. We organize these programs with high-quality test cases into a procedure-based dataset called CodeNet4Repair. CodeNet4Repair is an information-leak-free program repair dataset with Apache-2.0 License.

## 3 Method

Our methodology follows that of Ouyang et al., which aligned GPT-3 with human. We start from StarCoderBase (Li et al., 2023), a 15.5B code foundation pre-trained LM with 8K context length. It is trained on 1 trillion tokens from The Stack (Kocetkov et al., 2023). Figure 2 illustrates the techniques used to train RePair.

### 3.1 Supervised Fine Tuning on APR Task

To ensure that the LM can understand program repair tasks, we use the prompt templates in Appendix A.1 for Supervised Fine-Tuning (SFT) (Zhang et al., 2022; Huang et al., 2024). Given a vulnerable program sequence  $\mathbf{x} = \{x_1, x_2, \dots, x_N\}$ , LM is expected to output a robust program  $\mathbf{y} = \{y_1, y_2, \dots, y_{N'}\}$ ,  $y_t \in \mathcal{V}$  that can pass all test cases, where  $\mathcal{V}$  is vocabulary. During the training phase, the model parameters  $\theta$  are learned by maximizing the likelihood of the output and the ground-truth (Zhang et al., 2021). The training objective is to minimize the following loss:

$$\begin{aligned} \mathcal{L}_{ce}(\theta) &= \mathbb{E}[-\log P(\mathbf{y} | \mathbf{x}; \theta)] \\ &= -\sum_{t=1}^{N'} \log P(y_t | \mathbf{y}_{<t}, \mathbf{x}; \theta), \end{aligned} \quad (1)$$

where  $\mathbb{E}$  is the expectation over entire dataset, and  $\mathbf{y}_{<t}$  is a partial sequence before time-step  $t$ .

### 3.2 Process-based Feedback

After supervised fine-tuning, we generally obtain a fine-tuned LM suitable for the APR tasks. However, this model only understands how to provide a possible solution under the condition of given  $\mathbf{x}$ , lacking process supervision and feedback from environment like compilers and test cases. To ensure that the LM can gradually refine the program through interaction, we introduce reinforcement learning (RL). In this context, we treat the fine-tuned LM as an *actor*. Given a *state*  $\mathbf{x}$ , its output  $\hat{\mathbf{y}}$  is considered an *action*. Guided by feedback from the reward model, it iteratively refines program towards a final possible result.

#### 3.2.1 Reward Modeling

Instead of using direct feedback from the compiler and test cases, we train a reward model (RM) to assess program quality. The reward model serves as a *virtual tool*, while the LM optimizes repair strategies by interacting with it. The main reasons boil down to two points: (1) During training, direct interaction with the environment significantly blocks the training pipeline. The batch-processed tensors in cuda must first be transferred to memory for decoding, undergoing syntax check and case testing sequentially; when providing feedback, these results are re-encoded and sent back to cuda. This process drastically reduces the throughput - a cost that is unbearable. (2) During the inference phase,

---

#### Algorithm 1 Generate Repaired Program with Process-based Feedback, Actor-Critic Style

---

**Input:** Trained critic  $r_\phi(\cdot)$ ; Trained actor’s policy  $\pi_\theta$ ; Vulnerable program  $x_0$ ; Max iterations  $T$ ; Max Patience  $P$

**Output:** Repaired Program

```

1:  $t = 0$  ▷ Counter of timestep
2:  $p = 0$  ▷ Counter of unimproved patience
3:  $r_0 = r_\phi(x_0)$  ▷ Initialize the reward
4: while  $t < T$  &  $p < P$  do
5:    $x_{t+1} = \pi_\theta(x_t)$  ▷ Actor generate
6:    $r_{t+1} = r_\phi(x_{t+1})$  ▷ Critic review
7:    $\Delta = r_{t+1} - r_t$  ▷ Evaluate the boost
8:   if  $\Delta \leq 0$  then ▷ If no boost
9:      $p = p + 1$ 
10:  else
11:     $p = 0$  ▷ Counter reset
12:  end if
13:   $t = t + 1$ 
14: end while
15: return  $x_{t-p}$  ▷ Roll back to  $p$  steps ago

```

---

the inability to access test cases for vulnerable programs compromises generalization performance.

Unlike the methods used by (Christiano et al.; Ziegler et al.; Stiennon et al.; Ouyang et al.), our approach doesn’t require extensive investment in collecting human preferences to train a reward model. We can substantially reduce costs by sourcing program execution preferences from automated compiler and test cases. Specifically, we first empirically define a non-strict partial order based on program quality from high to low:

$$AC > PE > WA = TLE = MLE > CE > RE.$$

AC signifies that the program was “Accepted” by all test cases. PE stands for “Presentation Error”, which indicates that the output data is correct but not properly formatted. WA, TLE, MLE, CE, RE are abbreviations for “Wrong Answer”, “Time Limit Exceed”, “Memory Limit Exceed”, “Compile Error” and “Runtime Error” respectively. This partial order defines the severity of program errors in an ascending order, from AC (lowest) to RE (highest). We prioritize fixing RE over CE because RE typically indicates more serious issues such as logical errors which pose a greater risk than CE.

Then, we optimize the reward model parameters  $\phi$  using pairwise ranking based on the above partial order. Given  $K$  programs with status, all aimed



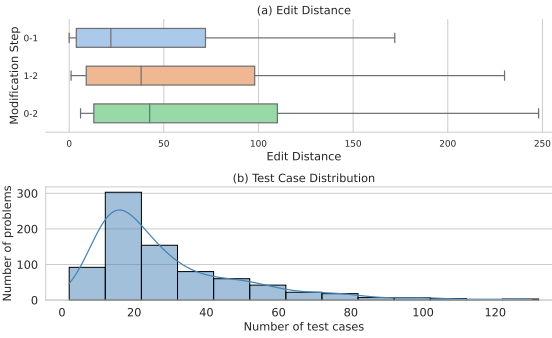


Figure 3: (a) The edit distances between three program after 2-step modifications. 0-1: The edit distance after the first refinement; 1-2: The edit distance after the second refinement; 0-2: The edit distance of a single-step refinement. (b) The distribution of test cases. Most of the test cases are concentrated between 10-20.

at solving an identical problem, there is a partial order set  $D$  containing  $\binom{K}{2}$  pairs of partial orders. After shuffling  $D$ , we train the reward model by minimizing the following pairwise ranking loss:

$$\mathcal{L}_{pr}(\phi) = -\frac{1}{\binom{K}{2}} \mathbb{E}_{(y_w, y_l) \sim D} [\log(\sigma(r_\phi(y_w) - r_\phi(y_l)))], \quad (2)$$

where  $r_\phi(\cdot)$  is a reward model, which takes a program as input and outputs a scalar reward. In practice, we use another smaller pre-trained autoregressive model, and replace the original model’s non-embedding layers with a projection layer to output a scalar value. The pair  $(y_w, y_l)$  is a partial order where  $y_w$  is the preferred program (for instance, the status of  $y_w$  is AC while  $y_l$  is WA).

### 3.2.2 Reinforcement Learning

In the previous two sections, we fine-tuned a LM and developed a reward model capable of realistically evaluating program states by optimizing loss  $\mathcal{L}_{ce}(\theta)$  and loss  $\mathcal{L}_{pr}(\phi)$  respectively. In this section, we will utilize reinforcement learning (RL) to finalize program repair through multiple interactions with the fine-tuned LM and well-trained reward model (Qian and Yu, 2021). We adopt Proximal Policy Optimization (PPO) algorithm (Stiennon et al., 2020; Ouyang et al., 2022; Schulman et al., 2017) to fine-tune our LM. The LM functions as an actor, generating a repaired program (*action*) based on the input program (*state*) and the *policy*  $\pi$  of the LM (i.e., LM parameters  $\theta$ ). The trained reward model plays as a critic, assessing the effectiveness of repairs and rewarding the LM accordingly. Our objective is to optimize the LM parameters  $\theta$  by maximizing these rewards.

	#problems	#records	#codes
Train set	563	94062	252031
Test set	61	10144	26377
	record per problem	code per record	size per code
Train set	167.07	2.68	270.42
Test set	166.3	2.60	246.88

Table 2: Dataset Information

To be more specific, we supervise the process by maximizing the the following objective function:

$$\begin{aligned} \mathcal{L}_{rl}(\theta) &= \mathbb{E}_{(x_i, x_{i+1}) \sim D \pi_{\theta'}} [r_\phi(x_{i+1}) - r_\phi(x_i) - \beta \text{KL}(\theta, \theta')] \\ &= \mathbb{E}_{(x_i, x_{i+1}) \sim D \pi_{\theta'}} \left[ \Delta_\phi^r(x_{i+1}, x_i) - \beta \log \frac{\pi_\theta(x_{i+1} | x_i)}{\pi_{\theta'}(x_{i+1} | x_i)} \right], \end{aligned} \quad (3)$$

where  $x_{i+1}$  is the refined output based on the input  $x_i$  and learned LM’s policy  $\pi_{\theta'}$ ,  $r_\phi(\cdot)$  is calculated as the learned reward model, the KL reward coefficient  $\beta$  control the strength of the KL penalty,  $\Delta_\phi^r(x_{i+1}, x_i) = r_\phi(x_{i+1}) - r_\phi(x_i)$  means the evaluation of the degree of improvement.

### 3.3 Multi-step Generation Under RM Supervision

We achieved process supervision with feedback by maximizing reward function. In this section, we will demonstrate how the language model interacts with virtual tools during the generation phase to achieve alignment during both training and generation stages. We continuously request the LM to refine solutions until either of two conditions is met: (1) there is no improvement ( $\Delta \leq 0$ ) in  $P$  consecutive steps; (2) the maximum number of iterations has been reached. The Algorithm 1 elucidates the specific process of generation.

## 4 Experiments

### 4.1 Data Preparation

We sample repair records that corrected after two-step modifications from CodeNet4Repair, resulting in three versions, and calculate the edit distance between these versions. In Figure 3 (a), the findings align with our hypothesis: The edit distance required to complete the repair in one step is longer than that required in multiple steps. Despite an increase in overall cost, step-by-step repairing can reduce the complexity of tasks.

We ensure evaluation quality by using only questions with high-quality test cases for our training and testing sets. We manually annotate and collect additional high-quality test cases from the internet to serve as hidden tests for each problem. This process resulted in test cases for 794 questions.

Model		Size	pass@1				pass@3				pass@5			
			Easy	Medium	Hard	All	Easy	Medium	Hard	All	Easy	Medium	Hard	All
Close-source	PaLM	540B	16.83	15.94	18.57	17.13	22.07	29.38	27.86	26.11	24.39	37.50	31.43	30.56
	GPT-3.5	-	53.41	45.32	39.14	46.39	65.24	62.03	58.57	62.13	68.29	65.63	65.71	66.67
	Claude2	-	50.73	43.75	36.29	43.98	65.12	63.75	55.29	61.53	71.95	73.44	62.86	69.44
	ChatGLM-Pro	-	20.49	20.94	17.43	19.63	23.17	27.19	24.43	24.77	24.39	29.69	28.57	27.31
Open-source	StarCoderBase	15B	0.49	0.31	0.00	0.28	1.46	0.94	0.00	0.83	2.44	1.56	0.00	1.39
	StarCoderChat	15B	0.00	0.00	0.57	0.46	0.00	0.00	1.29	0.42	0.00	0.00	1.43	0.46
	CodeGen2	16B	4.15	1.87	0.29	2.22	10.73	5.16	0.86	5.88	15.85	7.81	1.43	8.80
	CodeGeeX2	6B	7.80	7.19	1.43	5.56	19.27	17.03	3.86	13.61	26.83	23.44	5.71	18.98
	LLaMA2	70B	10.24	5.94	6.00	7.59	20.37	14.84	10.86	15.65	24.39	21.88	12.86	19.91
	LLaMA2-Chat	70B	30.73	23.13	18.86	24.63	39.02	33.13	25.71	32.96	41.46	39.06	30.00	37.04
	<b>Our Model</b>	15B	<b>51.61</b>	<b>44.13</b>	<b>40.57</b>	<b>44.34</b>	<b>62.47</b>	<b>59.98</b>	<b>52.34</b>	<b>60.01</b>	<b>67.75</b>	<b>64.14</b>	<b>60.32</b>	<b>65.66</b>

Table 3: Results on CodeNet4Repair: compared to open-source models, our process-based feedback method has demonstrated superior performance. It also remains competitive when compared with commercial LLMs.

Figure 3 (b) shows the distribution of these test cases. To prevent data leakage, we divided CodeNet4Repair based on the problem ID in a ratio of 9:1. The training set consists of 94,062 repair processes, while the testing set comprises 10,144 repair processes. Table 2 provides the statistical information about filtered dataset.

## 4.2 Experimental Setup

In the fine-tuning stage, we train our model using mixed precision training. We use AdamW optimizer with  $2e-5$  learning rate. To prevent training instability caused by an excessively high learning rate, we utilize a cosine LR schedule down to 10% of the original learning rate with learning rate warmup. We use ZeRO++ to distribute model tensors across accelerators (Wang et al., 2024).

In reward modeling, we adopt the LR of  $9.6e-6$  and cosine learning rate schedule. We randomly select  $K = 9$  programs in same problem and rank them based on their execution status. Each batch contains 64 units, thus a single gradient backpropagation involves  $64 \times \binom{9}{2} = 2304$  comparisons.

In the process-based feedback stage, we initialize RL policies from the fine-tuned LM. We train the LM for 32k episodes with 512 batch size. We assign a KL penalty factor,  $\beta$ , of 0.02 and establish a learning rate of  $9e-6$ . We perform nucleus sampling with  $\text{top}_p=0.95$  and  $\text{top}_k=50$ , and set the temperature to 0.2.

## 4.3 Baselines

We compare the latest state-of-the-art LM that possess few-shot learning and code comprehension capabilities. These models include PaLM (chat-bison-001) (Anil et al., 2023), GPT-3.5 (gpt-3.5-turbo-0613), Claude2 and ChatGLM-Pro (Zeng et al., 2022). We also compare some open-

sourced models such as StarCoderBase/Chat (Li et al., 2023), CodeGen2, CodeGeeX2 (Zheng et al., 2023), LLaMA2/-chat (Touvron et al., 2023).

## 4.4 Evaluation

Previous program repair datasets lacked annotated test cases and stable test environment (Dinella et al., 2020; Hendrycks et al., 2021; Li et al., 2022; Wang et al., 2023), leading to a shortage of automatic evaluation methods for program repair based on execution results. We contend this issue and advocate for the evaluation using execution outcomes.

Following previous code generation’s evaluation method, we use  $\text{pass}@k$  as our evaluation metric (Hendrycks et al., 2021; Chen et al., 2021; Nijkamp et al., 2023). Here we use the unbiased estimator proposed in (Chen et al., 2021), which is defined as follows:

$$\text{pass @ } k = \mathbb{E} \left[ 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]. \quad (4)$$

In this work, we report  $k \in \{1, 3, 5\}$  as the final result. The reason is that in  $\text{pass}@k$ , a larger  $k$  provides a more comprehensive assessment of LM’s ability to generate code. However, in practice, users won’t request multiple candidates repeatedly; therefore, reporting a small number of  $k$  suffices for program repair evaluation.

## 4.5 Main Results

We use the pass rate of all problems in 14 billion submission records as a standard to measure the difficulty of the problems. These problems are categorized into three levels - easy, medium, and hard - based on their pass rates divided into tertiles. The model’s performance at various difficulties is detailed in Table 3. The results of the experiment

Model			pass@1	pass@3	pass@5
Process	Feedback	Reward			
×	×	-	36.32	47.63	52.63
×	✓	Pair	37.26	54.81	62.27
✓	×	-	20.43	35.14	44.72
✓	✓	Pair	<b>44.34</b>	<b>60.01</b>	<b>65.66</b>
✓	✓	Point	40.11	58.89	64.27
✓	✓	List	39.45	57.33	63.56

Table 4: The results of ablation study. We conduct an in-depth study on the design of process supervision, feedback, and reward functions. The experimental results confirm our model’s effectiveness. Point: Point-wise loss; Pair: Pair-wise loss; List: List-wise loss.

indicate that our model outperforms all other open-source models. Even when compared to sophisticated commercial LLMs, we achieve competitive results. Starting with an analysis of the open-source models: We acknowledge that some performance gains come from supervised fine-tuning. However, as evidenced by StarCoderBase/Chat results, our foundation model lacks zero- or few-shot program correction capabilities and struggles with repair tasks without supervised fine-tuning. In comparison to LLMs designed for code downstream tasks, our repair model generates programs that better meet task requirements and outperforms them at every difficulty level. Moreover, in LLaMA2-Chat we found that multiple rounds of dialogue-based supervised fine-tuning do not degrade but rather enhance its understanding of human instructions and repair abilities. While our model’s performance is not yet on par with commercial LLMs such as ChatGPT and Claude2, considering the resources they’ve invested - including compute, data collection, and manual annotation among others - which are significantly greater than ours, we have still achieved competitive results. This validates the effectiveness of our process-based feedback method and provides guidance for future research.

#### 4.6 Model Analysis

We explore the rationality of the model including process supervision, feedback, and reward function design. We design different variants from those three perspectives. First, we arrange and combine process supervision and feedback to create four different approaches. Then, we discuss the design of reward functions within the process-based feedback framework.

**Impacts of Process Supervision.** We attempt to eliminate process supervision signals, enabling

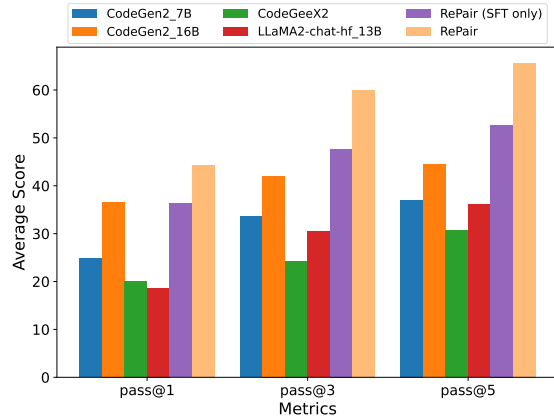


Figure 4: Performance comparison on other fine-tuned open-sourced model.

the model to depend solely on single-step supervision signals. Without feedback, the model degenerates into a fine-tuned single-step program repair model. While with the feedback, the fine-tuned model’s adjustment of its repair policies is limited to the final signal.

**Impacts of Feedback.** We remove feedback from the reward function during process supervision, which prevents the language model from using reinforcement learning for policy adjustment. When given a repair sequence, we force the language model to use the subsequent repair step as a supervision signal for learning.

**Impacts of Reward Function.** We introduce two variants to verify the effectiveness of pairwise ranking in reward modeling. We use two ranking methods: point-wise, and list-wise. Point-wise ranking scores individual program states without considering their order relationship; while list-wise ranking compares multiple programs together (beyond pairs) to determine their order relationship.

The results of the experiment are shown in Table 4. We can draw the following conclusions from the results: (1) Regardless of the supervision method used, introducing feedback to adjust LM’s repair policies is necessary. Performance may significantly decrease due to additional noise if feedback is lacking in the process-based supervision method. (2) Pair-wise ranking proves more effective as a reward model compared to point-wise or list-wise ranking. Using point-wise ranking solely requires the model to score each program based on compiler and test case, without comprehending the program’s preferences and modification direction. Conversely, providing  $K$  rankings simultaneously

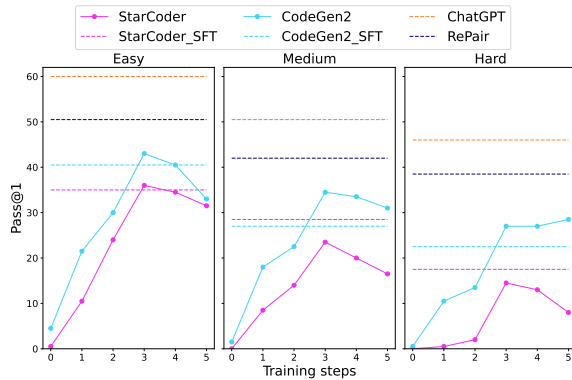


Figure 5: Performance of explicit prompts at different training steps. At three difficulty levels, the best performance was achieved at three training steps. ChatGPT can effectively understand feedback from the compiler and test cases. For small-scale models, explicit prompts are still difficult to understand.

in a list-wise task can be overly complicated and potentially hinder performance.

#### 4.7 Process-based Feedback Can Bring Performance Benefits

Someone might question the superiority of RePair over current open-source models due to our use of SFT on the training set, while other models employed a few-shot setting. It is important to clarify that SFT was utilized to enhance the model’s comprehension of the repair task, rather than being the primary factor for performance improvement. To substantiate this, we also fine-tune several open-source models with similar parameter counts using the same dataset. The results, presented in Figure 4, demonstrate a performance enhancement following SFT. However, this improvement is largely due to the model’s increased understanding of the task and the standardization of output formats. Furthermore, when combined with SFT, process-based feedback method significantly amplifies the model’s potential, leading to even greater performance gains.

#### 4.8 Smaller Models May Struggle to Receive Effective Explicit Feedback

As mentioned in the abstract and introduction, understanding explicit prompts for small-scale LMs remains challenging. We directly interact with LMs using the prompt template provided in Appendix A.1. We fine-tune two open-source models, StarCoder and CodeGen2, within 5 steps and present the experimental results in Figure 5. The experimental results indicate that: (1) The performance of the open-source model peaked after three

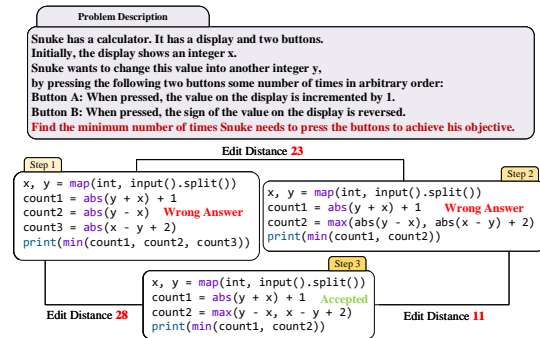


Figure 6: An example of repair process for “Count the number of integer transitions”.

steps of fine-tuning. (2) ChatGPT’s performance was further enhanced after receiving feedback from compilers and test cases, demonstrating better results than one-step repair. (3) Small-scale models struggle to comprehend feedback information through explicit prompts.

#### 4.9 Qualitative Results

As shown in Figure 6, our model through step-by-step modification, have effectively simplified the complexity of each repair step and achieved satisfactory “Accepted” results. For more examples, please refer to the Appendix A.3.

### 5 Related Works

#### 5.1 Large Language Models

The recent years have witnessed a substantial evolution in the field of large language models. language modeling is the prediction of the probability distribution of the next token given a context. The scaling law makes it possible to enhance performance by increasing the size of the model and data (Kaplan et al., 2020). The emergence of large-scale language models with over 100B parameters has consistently resulted in record-breaking performance (Srivastava et al., 2023; Hendrycks et al., 2021; Chen et al., 2021). Larger models begin to solve problems that smaller models cannot handle, a phenomenon known as *emergence*. Typical phenomena of emergence include In Context Learning (ICL) and Chains of Thought (CoT) (Brown et al., 2020; Wei et al., 2022). The core of ICL is to draw knowledge from analogies. After providing demonstrations, the query problem is presented alongside prompts to form the input. A CoT is a prompt formed through a series of logical thinking steps, guiding the LLMs to answer in a thoughtful manner.



This method can significantly improve performance on reasoning tasks (Liu et al., 2023b). Our work is dedicated to using small-scaled language models to enhance program repair performance.

## 5.2 Outcome- and Process- based Approaches

The outcome-based approaches supervise the final results, while the process-based approach focuses on each step leading to the result. Both have their strengths and weaknesses: outcome-based requires fewer supervision signals, but process-based approaches align more with human thinking patterns. In (Uesato et al., 2022), researchers conducted experiments on math dataset and found that both approaches produced similar error rates, but due to cost considerations, the process-based method performed better. However, recent research (Lightman et al., 2023) has proposed a different view; after building larger datasets and annotating more processes, the process-based method showed clear advantages. Process-based can help machine learning systems accurately understand humans’ way of thinking and regulate potential hallucination.

## 5.3 Learning from Feedback

Through learning from feedback, language models can effectively receive signals beyond labels (Gou et al., 2024). Some NLP tasks rely on automated evaluation metrics such as BLEU, ROUGE to get feedback from ground truth. However, this kind of automatic feedback that solely depends on exact matching is not an effective assessment criterion (Colombo et al., 2022). Therefore, researchers are attempting to train language models using human preferences as supervisory signals. In some open text generation tasks, learning from human feedback has significantly improved performance (Jaques et al., 2019; Bahdanau et al., 2016; Lawrence and Riezler, 2018; Zhou and Xu, 2020; Stiennon et al., 2020). Following this line, (Ouyang et al., 2022) applies human feedback to GPT-3 to achieve alignment with humans, thereby mitigating the issue of generating untruthful or toxic text. Our work focuses on obtaining feedback from the compiler as a *virtual tool*, integrating signals from the tools into the training process of language models.

## 6 Conclusion

In this work, we proposed an automated program repair method that utilizes process-based feedback. This approach is inspired by strategies used in programming competitions and incorporates process

supervision into the repair process. Due to the unavailability of suitable datasets, we first created a multi-step repair dataset called CodeNet4Repair. This dataset not only includes the initial faulty programs and the final accepted ones but also records the intermediate repair process. On this basis, we implemented supervision over the process with feedback. Specifically, we fine-tuned a pre-trained model StarcoderBase through to make it understand program repair tasks. Then, we empirically defined a partial order relationship for program states and trained reward models using pairwise ranking. This reward model would serve as a critic, assessing each attempt by the LM to polish the program. Meanwhile, the LM plays the role of an actor, constantly adjusting its policies for repairing the program based on received rewards. During the inference stage, LM iteratively repaired the program until the exit conditions are met. The experimental results indicated that the process-based feedback method can demonstrate excellent performance. With smaller model sizes, **RePair** still can rival or even surpass closed-source commercial large models. We believe that process-based feedback is currently underexplored, and will continue to explore more universal methods.

## 7 Limitation

We contributed a dataset called CodeNet4Repair, which focuses on program repair in real competition scenarios. We acknowledge that CodeNet4Repair still has limitations in evaluating model repair capabilities. Due to time and resource constraints, we were unable to collect all repair processes across different languages and software engineering domains. But there are significant similarities between competition and engineering scenarios. This implies that our model can achieve a level of usability for engineering project purposes, and our dataset can partially evaluate its repair performance in an engineering context.

## 8 Acknowledgement

This research was partially supported by grants from the National Natural Science Foundation of China (No. 62106244), the Anhui Provincial Natural Science Foundation (No. 2308085QF229), the Fundamental Research Funds for the Central Universities (No. WK2150110034), and the CIPSC-SMP-Zhipu.AI Large Model Cross-Disciplinary Fund.

## References

- Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.
- Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. 2016. An actor-critic algorithm for sequence prediction. In *ICLR*.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests.
- Mark Chen, Jerry Tworek, et al. 2021. Evaluating large language models trained on code.
- Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.
- Pierre Jean A. Colombo, Chloé Clavel, and Pablo Piñanida. 2022. Infoml: A new metric to evaluate summarization and data2text generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10):10554–10562.
- Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujia Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*.
- Daya Guo, Qihao Zhu, et al. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.
- Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1).
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.
- Zhenya Huang, Yuting Ning, Longhu Qin, Shiwei Tong, Shangzi Xue, Tong Xiao, Xin Lin, Jiayu Liu, Qi Liu, Enhong Chen, and Shijing Wang. 2024. Edunlp: Towards a unified and modularized library for educational resources.
- Faria Huq, Masum Hasan, Mahim Anzum Haque Pantho, Sazan Mahub, Anindya Iqbal, and Toufique Ahmed. 2020. Review4repair: Code review aided automatic program repairing.
- Natasha Jaques, Asma Ghandeharioun, Judy Hanwen Shen, Craig Ferguson, Agata Lapedriza, Noah Jones, Shixiang Gu, and Rosalind Picard. 2019. Way off-policy batch deep reinforcement learning of implicit human preferences in dialog.
- Nan Jiang et al. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.
- Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.
- Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt?
- Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. 2023. The stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*.
- Carolin Lawrence and Stefan Riezler. 2018. Improving a neural semantic parser by counterfactual learning from human bandit feedback. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1820–1830, Melbourne, Australia. Association for Computational Linguistics.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*.
- Raymond Li, Loubna Ben, et al. 2023. Starcoder: may the source be with you!
- Yujia Li, David Choi, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

- Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. [Let's verify step by step](#).
- Jiayu Liu, Zhenya Huang, Zhiyuan Ma, Qi Liu, Enhong Chen, Tianhuang Su, and Haifeng Liu. 2023a. Guiding mathematical reasoning via mastering commonsense formula knowledge. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1477–1488.
- Jiayu Liu, Zhenya Huang, Chengxiang Zhai, and Qi Liu. 2023b. Learning by applying: A general framework for mathematical reasoning via enhancing explicit knowledge learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4497–4506.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.
- OpenAI. 2023. [Gpt-4 technical report](#).
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.
- Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 1035.
- Hong Qian and Yang Yu. 2021. Derivative-free reinforcement learning: A review. *Frontiers of Computer Science*, 15(6):156336.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, et al. 2024. [Code llama: Open foundation models for code](#).
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. [Proximal policy optimization algorithms](#).
- Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, et al. 2023. [Beyond the imitation game: Quantifying and extrapolating the capabilities of language models](#). *Transactions on Machine Learning Research*.
- Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE.
- Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. [Solving math word problems with process- and outcome-based feedback](#).
- Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2024. [ZeRO++: Extremely efficient collective communication for large model training](#). In *The Twelfth International Conference on Learning Representations*.
- Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2023. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1271–1290.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.
- Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. [Copiloting the copilots: Fusing large language models with completion engines for automated program repair](#). In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '23*. ACM.
- Aohan Zeng, Xiao Liu, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*.
- Kai Zhang, Qi Liu, Hao Qian, Biao Xiang, Qing Cui, Jun Zhou, and Enhong Chen. 2021. Eatn: An efficient adaptive transfer network for aspect-level sentiment analysis. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):377–389.
- Kai Zhang, Kun Zhang, Mengdi Zhang, Hongke Zhao, Qi Liu, Wei Wu, and Enhong Chen. 2022. Incorporating dynamic semantics into pre-trained language model for aspect-based sentiment analysis. *arXiv preprint arXiv:2203.16369*.
- Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma:

Revisiting template-based automated program repair via mask prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 535–547. IEEE.

Qinkai Zheng, Xiao Xia, et al. 2023. *Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x*.

Wangchunshu Zhou and Ke Xu. 2020. *Learning to compare for better training and evaluation of open domain natural language generation models*.

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2020. *Fine-tuning language models from human preferences*.

## A Appendix

### A.1 Prompts Used in Experiments

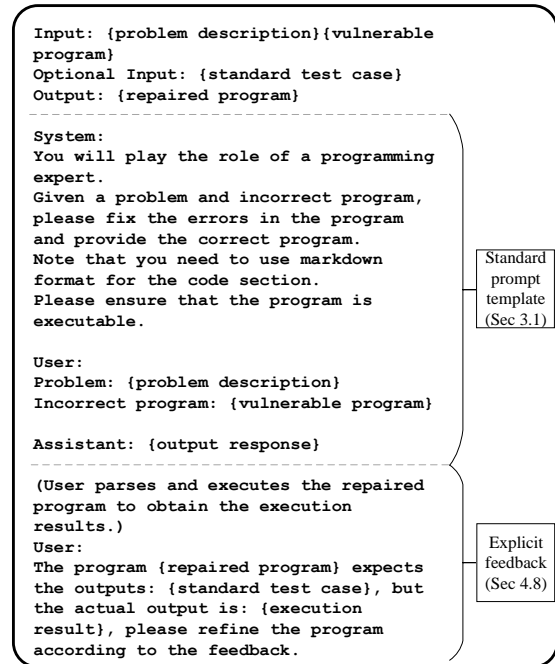


Figure 7: Prompt Template for Program Repair.

We performed supervised fine-tuning and explicit feedback experiments following the prompt template shown in Figure 7. For explicit feedback, we also input optional standard test cases.

### A.2 Ethics Statement

As creators and users of LLMs, we recognize the profound ethical responsibilities that come with their development and deployment. We will strive to avoid creating or perpetuating harm, including the reinforcement of bias, misinformation, and harmful stereotypes. We advocate for the use of open-source LLMs in potential ethical risk research. We believe that a public LLM can reveal its internal state, thereby avoiding possible risks.

### A.3 Detailed Case Study

In the next page, we provide six examples of repairs. Through the examples shown, we found that the edit distance for single-step modifications (directly from Step 1 to Step 3) is very high, resulting in tasks being too complex (e.g., having both compilation errors and semantic errors) to be completed in one step. However, process-based modifications can decompose complex tasks into simple ones and solve each problem gradually, thereby producing good repair effects.



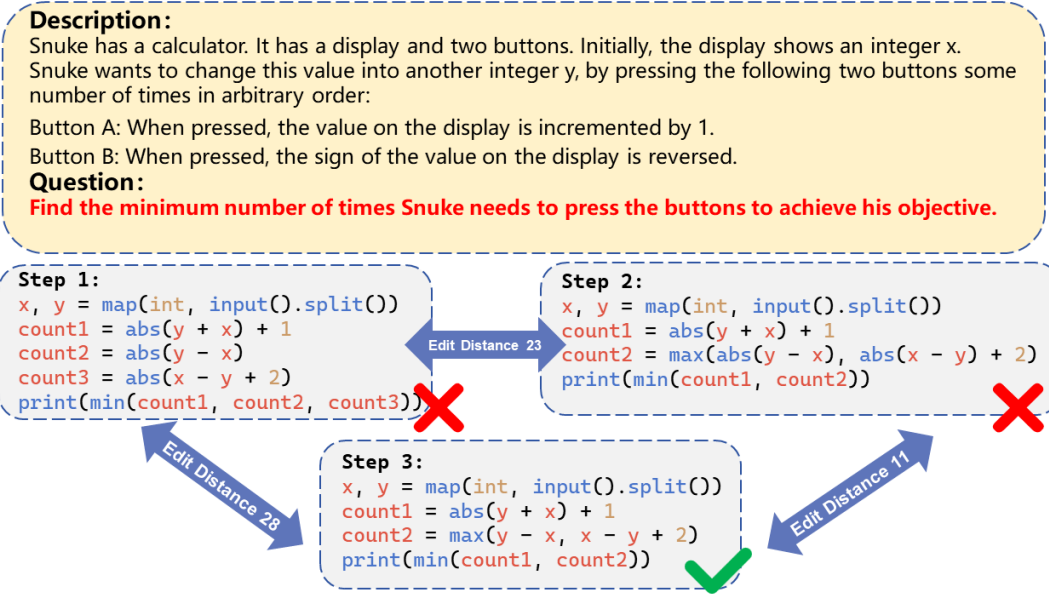


Figure 8: Case 1, an example of repair process for “Count the number of integer transition”.

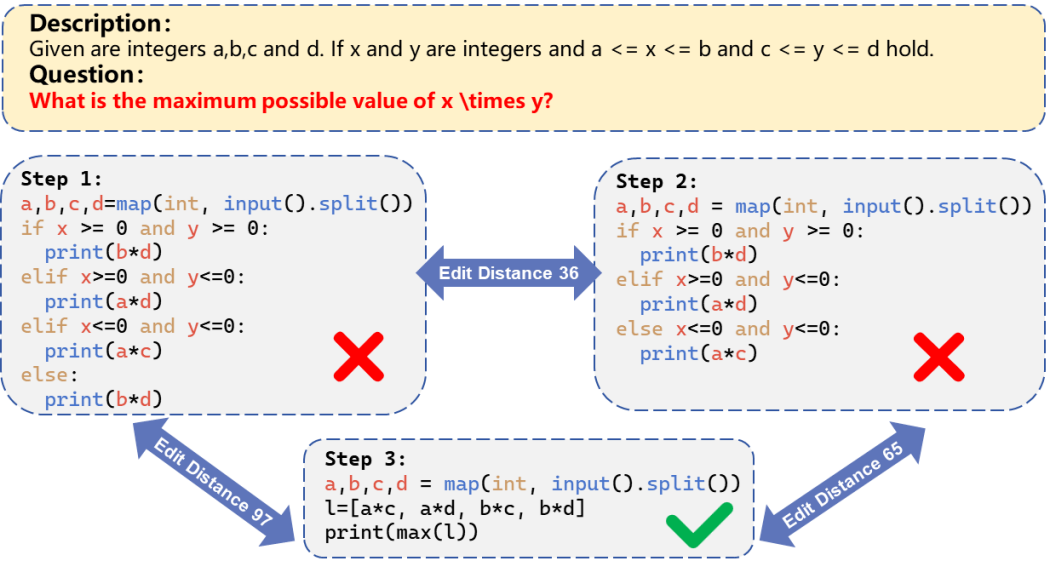


Figure 9: Case 2, an example of repair process for “Maximum possible value of  $x \times y$ ”.

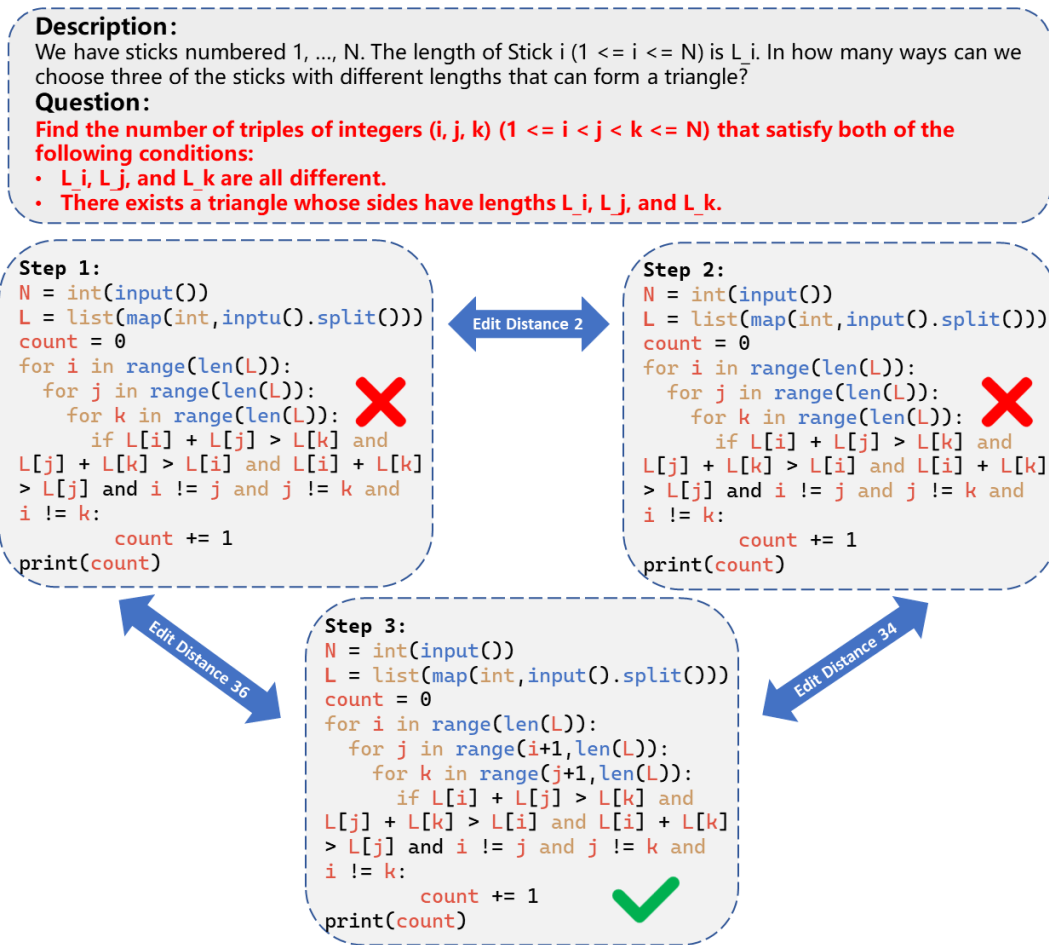


Figure 10: Case 3, an example of repair process for “The number of combinations of sticks that can form a triangle”.

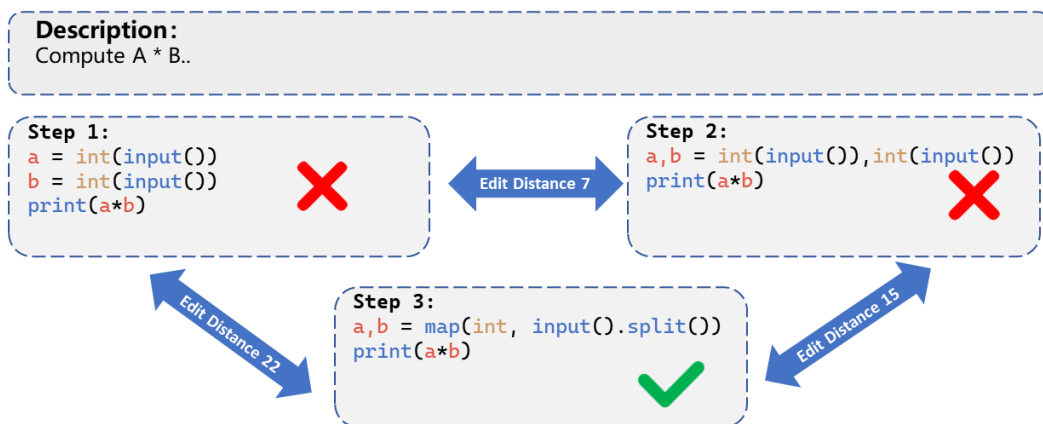


Figure 11: Case 4, an example of repair process for “ $A \times B$ ”.

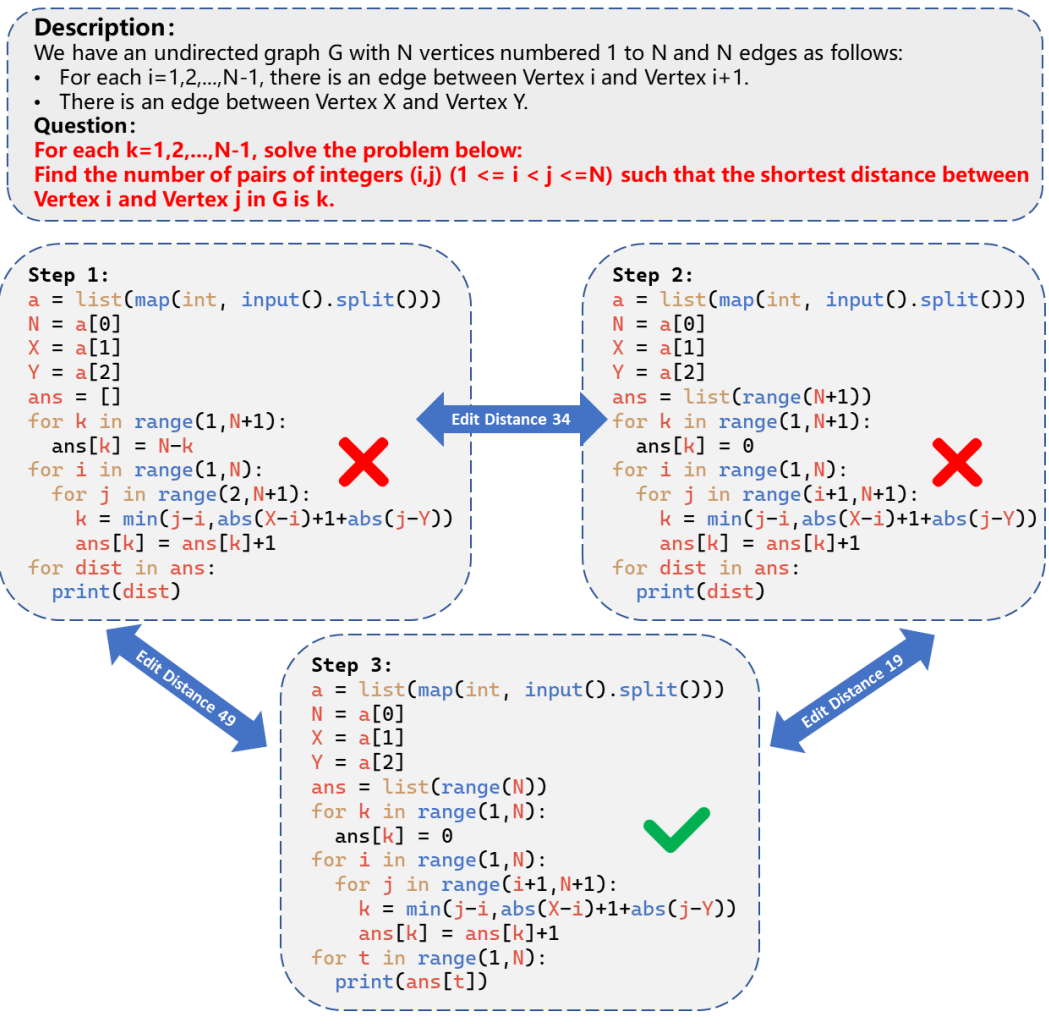


Figure 12: Case 5, an example of repair process for “Distance calculation in an undirected graph”.

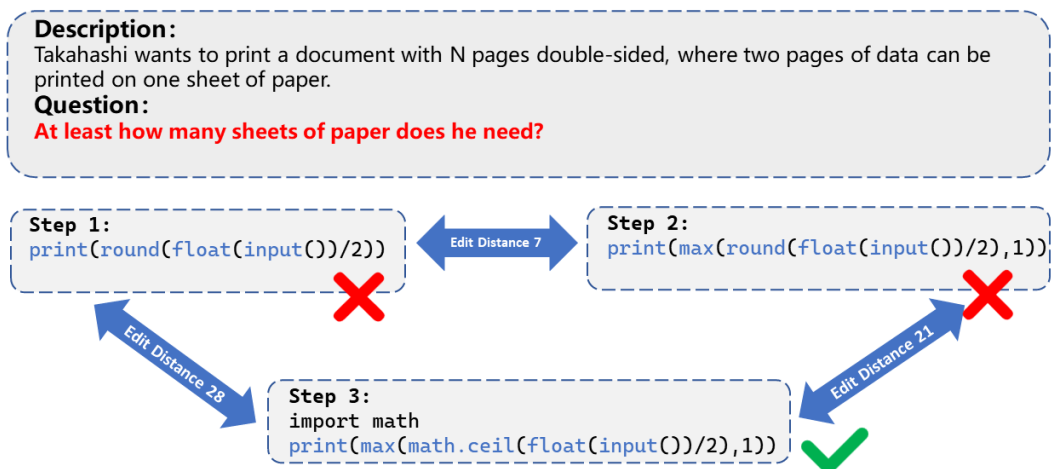


Figure 13: Case 6, an example of repair process for “Calculate the minimum number of sheets of paper required”.