Less is More: Adaptive Program Repair with Bug Localization and Preference Learning

Zhenlong Dai[1,4], Bingrui Chen[2], Zhuoluo Zhao[1], Xiu Tang[1,4], Sai Wu[1,4], Chang Yao[1,4], Zhipeng Gao[1*], Jingyuan Chen[1*]

[1]Zhejiang University
[2]Hohai University
[3]Guizhou University
[4]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
{zhenlongdai,tangxiu,wusai,changy,zhipeng.gao,jingyuanchen}@zju.edu.cn,
ChenBingrui@hhu.edu.cn, ie.zlzhao21@gzu.edu.cn

Abstract

Automated Program Repair (APR) is a task to automatically generate patches for the buggy code. However, most research focuses on generating correct patches while ignoring the consistency between the fixed code and the original buggy code. How to conduct adaptive bug fixing and generate patches with minimal modifications have seldom been investigated. To bridge this gap, we first introduce a novel task, namely **AdaPR** (**Ada**ptive **P**rogram **R**epair). We then propose a two-stage approach ***AdaPatcher*** (**Ada**ptive **Patch** G**e**nerato**r**) to enhance program repair while maintaining the consistency. In the first stage, we utilize a Bug Locator with self-debug learning to accurately pinpoint bug locations. In the second stage, we train a Program Modifier to ensure consistency between the post-modified fixed code and the pre-modified buggy code. The Program Modifier is enhanced with a location-aware repair learning strategy to generate patches based on identified buggy lines, a hybrid training strategy for selective reference and an adaptive preference learning to prioritize fewer changes. The experimental results show that our approach outperforms a set of baselines by a large margin, validating the effectiveness of our two-stage framework for the newly proposed AdaPR task.

Code — https://github.com/zhenlongDai/AdaPatcher

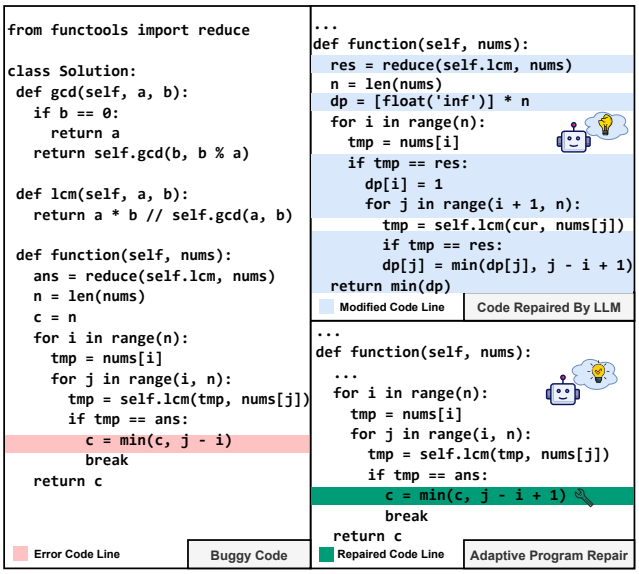

Figure 1: Example of AdaPR. The adaptive repaired code is correct and minimizes code modifications.

Introduction

As software systems become more and more prevalent in everyday life, software bugs also become inevitable. These software bugs can potentially cause security issues or even financial losses (Shahriar and Zulkernine 2012; Dissanayake et al. 2022; Krasner 2021). Usually, developers need to fix these buggy codes manually by spending a significant amount of time and effort. To alleviate developers' burden for bug fixing, Automated Program Repair (APR) has been introduced to automatically generate patches given the original buggy code. APR techniques take a buggy code and a correct specification as input, aiming to generate the fixed program satisfying the given specifications.

Nowadays, inspired by the promising performance of Large Language Models (LLMs) in understanding (Li et al. 2022; Chen et al. 2021; Wu et al. 2024; Lin et al. 2024) and code generation (Jiang et al. 2024; Li et al. 2023; Dai et al. 2024), researchers have applied LLMs to perform the APR task (Ye et al. 2022; Fan et al. 2023; Jin et al. 2023) and demonstrated remarkable results. However, most studies focus on generating *correct* patches, the *consistency* between the fixed code and the buggy code is often ignored and cannot be guaranteed, which greatly hinders the practical use of LLM for bug fixing. Consider the practical scenario in Fig. 1 as an example, Alice is a developer, she implemented the `Solution` class to achieve her goal. Nonetheless, her program failed to pass the tests which indicates potential bugs within her written code. The error message suggested "AssertionError: Expected output is 4, but the received output is 3". Alice tried to use LLM to help her fix this bug by feeding LLMs with the original buggy code and error

*Co-corresponding authors.

message. However, the patch generated by LLMs overwrote most code lines in `function` (*e.g.*, colored in blue). It is difficult for Alice to accept this patch because the generated code is too far from her original written one. The extensive modifications made by LLMs make the fixed code hard to trace and understand. As a result, Alice refused to integrate this patch into her codebase.

To address this gap, we propose a new task in this paper, namely **Adaptive Program Repair**, denoted as **AdaPR**. Different from APR, AdaPR not only aims to generate "*correct*" patches for the buggy code, but also aims to generate "*consistent*" patches with minimal modifications. More formally, given the buggy code and the correct specifications (*e.g.*, failed test cases), AdaPR adaptively fixes the buggy program with the least possible changes to satisfy the given specification. For example, in Fig. 1, Alice can fix this bug by only changing one line of code, *i.e.*, from `c = min(c, j - i)` to `c = min(c, j - i + 1)`. This newly generated patch aligns with her design intentions and existing code structures, the consistency between the fixed code and the original buggy code makes Alice easy to understand the code changes and increases her confidence of this fix pattern. Consequently, Alice accepted this patch without a doubt and incorporated it into her codebase.

So far, the existing studies focus on generating *correct* patches, there is no research investigating how to adaptively fix buggy code with minimal modifications. AdaPR is a non-trivial task regarding the following key challenges: (i) ***Where to fix:*** Identifying the precise location(s) where the bug has been introduced is challenging. When a bug occurs, different parts of the program may exhibit abnormal behaviors according to the bug. To fix the bug adaptively, AdaPR first requires locating the root cause of the problem and pinpointing the exact buggy line(s) that need modifications. (ii) ***How to fix:*** Generating patches with minimal modifications is challenging. Because LLMs are typically trained on general programming corpus (*e.g.*, comment-code pairs, question-solution pairs), LLMs' primary goal is to generate correct and functional code. Regarding program repair, LLMs tend to repair a program by rewriting it from scratch without considering the existing code structure or semantics. AdaPR requires the patches to be both correct and consistent. In other words, the generated patches should involve as few modifications as possible while still addressing bugs effectively. How to fix the program incrementally and adaptively is another challenge in this work.

To tackle the above challenges, we propose a novel two-stage approach named $\boldsymbol{AdaPatcher}$, which is designed to patch a buggy program correctly and consistently. To address the first *where to fix* challenge, we propose a diff-based component, namely Bug Locator, to pinpoint the exact bug locations within the buggy code. Specifically, for a passed program and a failed program written by the same developer, we first record the run-time values of different variables respectively. Following that, we teach LLM to do self-debug learning to identify bug locations, i.e., the LLM is guided to debug and analyze the differences (*e.g.*, code deletions, modifications) between the passed program and the failed program and finally determine which code line causes the test failures. To address the second *how to fix* challenge, we design a Program Modifier component for our second stage. The Program Modifier is enhanced with three techniques to ensure the consistency and correctness of the fixed program and the original buggy program. Particularly, to avoid LLMs repairing the program from scratch, we leverage location-aware repair learning to generate patches based on the identified buggy lines. To reduce the negative effects of the incorrect bug locations, we propose a hybrid training strategy that enables the Program Modifier to selectively reference bug locations instead of blindly modifying them. Moreover, to make code changes as small as possible, the Program Modifier is trained to generate fewer modifications by adaptive preference learning.

In summary, our paper makes the following contributions: (1) We first propose a novel task, namely AdaPR, to fix buggy code with minimal modifications. This newly proposed task aims to produce both correct and consistent code patches, which is more practical in real-world software development; (2) We build a dataset with over 50K(3) We present a novel model, named $\boldsymbol{AdaPatcher}$, to perform the AdaPR task. $\boldsymbol{AdaPatcher}$ is based on LLMs and introduces several customized improvements to effectively handle *where to fix* and *how to fix* challenges. The experimental results show the effectiveness of our model over a set of baselines, showing its potential to enhance automated program repair while reducing modifications at the same time. We hope our study can lay the foundations for this research topic.

Related Work

Recent advancements in LLMs have spurred their integration into automated program repair (Sobania et al. 2023; Xia, Wei, and Zhang 2023; Jiang et al. 2023; Paul et al. 2023). Enhancing code LLMs with feedback mechanisms has demonstrated potential (Miceli-Barone et al. 2023), particularly through feedback from tools like compilers (Bouzenia et al. 2023; Xia and Zhang 2022) such as traces or test diagnostics. CoT reasoning loop (Yao et al. 2022) has been used to predict repair actions based on interactive feedback from debuggers. NExT (Ni et al. 2024) focuses on tuning LLMs to reason with pre-existing execution information. Additionally, LLMs can generate natural language explanations for errors (Chen et al. 2023; Zhang et al. 2022), offering another valuable form of feedback. Self-improvement methods iteratively refine code generated by LLMs using CoT reasoning over self-provided feedback (Madaan et al. 2024; Zhang et al. 2023). CoFFEE (Moon et al. 2023) uses LLMs to generate natural language explanations for errors. Existing approaches in automated program repair focus on accuracy, while our research aims to enhance program repair with minimal modifications.

Methodology

In this section, we introduce a novel two-stage framework $\boldsymbol{AdaPatcher}$, aimed at enhancing program repair while maintaining the consistency. The first stage employs a Bug Locator to identify the root cause of bugs and pinpoint the

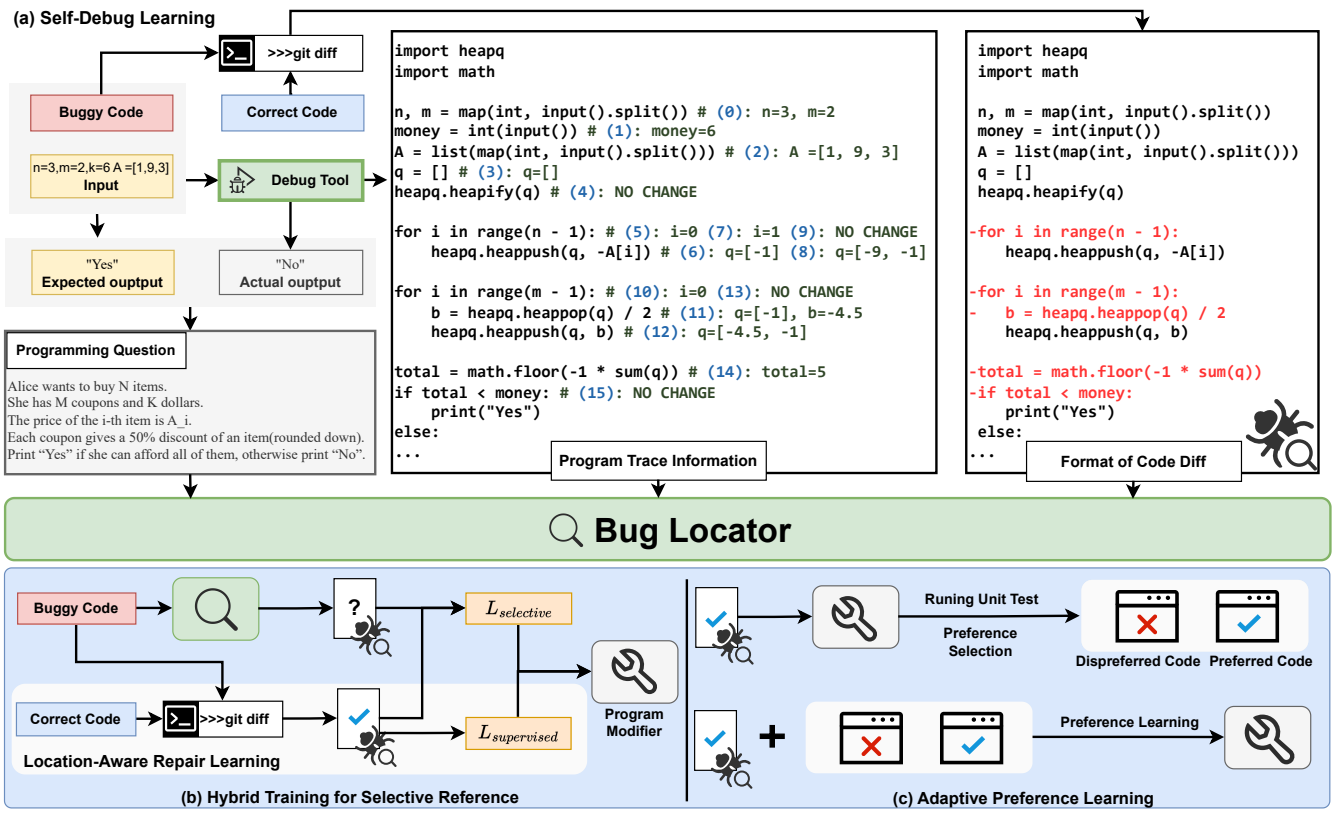

Figure 2: Overview of ***AdaPatcher***. (a) Illustration of the Self-Debug Learning process. (b) Illustration of the Hybrid Training for Selective Reference process. (c) Illustration of the Adaptive Preference Learning process.

buggy code lines in the form of `Code Diff`[1]. The second stage utilizes a Program Modifier to adaptively propose fixes for the identified buggy code lines. This approach prioritizes patches that require minimal changes, thus preserving the cleanliness and maintainability of the codebase.

Task Definition

Given a specific programming task q, a buggy code c, and a correct specification s, the objective is to generate a fixed version of the buggy code, denoted as y, which satisfies the specification s. The fixed version y should maintain consistency with the surrounding code and require minimal modifications to the original code c.

Stage I: Bug Locator

LLMs demonstrate strong code comprehension capabilities (Nam et al. 2024); however, they often struggle with accurately identifying and describing code bugs (Olausson et al. 2023), particularly in pinpointing buggy lines. To address this challenge, we propose a diff-based approach that simplifies and clarifies bug locations for LLMs. As shown in Fig. 2, bug locations are aligned with the corresponding buggy lines both semantically and structurally, with a '-' symbol prefix indicating the need for deletion or correction. The form of `Code Diff` simplifies and clearly pinpoints bug locations by providing a structured and explicit indication of where changes are needed. This makes it easier for the Bug Locator to focus their attention on the relevant buggy portions of the code.

Additionally, since LLMs often lack an understanding of program execution at runtime, identifying and locating runtime bugs is challenging. To address this, we propose a novel self-debug learning to enhance the Bug Locator's ability to identify and locate runtime errors.

Format of `Code Diff`. Given a buggy code $c = \{c_1, c_2, ..., c_n\}$ and a corrected code y, a diff file is generated using Git[2] by comparing c and y. Lines marked with a '-' symbol in the diff file are identified as buggy lines L. The diff file $d = \{d_1, d_2, ..., d_n\}$ is created by prefixing buggy lines in c with a '-' symbol:

$$d_i = \begin{cases} \text{<space>} \cdot c_i, & c_i \notin L \\ \text{'-'} \quad \cdot c_i, & c_i \in L \end{cases} \tag{1}$$

where d_i is the i-th line of the diff file, and both d and c contain n lines. The symbol $\cdot$ represents string concatenation, and <space> denotes a whitespace character.

[1] `Code Diff` refers to the differences between the buggy and correct code.

[2] https://git-scm.com/

Self-Debug Learning. Certain bugs manifest only during runtime, necessitating an understanding of program execution. LLMs often struggle with these bugs due to their training on the static textual form of code. Drawing inspiration from the practice of rubber duck debugging (Parreira, Gillet, and Leite 2023; Ni et al. 2024), we introduce self-debug learning to enhance the Bug Locator θ's ability to identify and localize runtime bugs.

Specifically, given a buggy code c and a corresponding failed test case t from the correct specification s, the code is executed with t to capture the actual output. The program's I/O data, denoted as D_t, includes both input and expected/actual output. Additionally, using the Python '`traceback`' module[3], as shown in Fig. 2, we capture the variable states at each executed line and record the execution order (*e.g.*, colored in blue) to create program trace information R_t.

To facilitate LLM comprehension of program trace information, R_t is formatted as compact inline code comments (*e.g.*, colored in green) that do not disrupt the code structure: 1) Comments display only variables that change after each line's execution, marking each execution step; and 2) Loop trace information is compressed using ellipses for large iteration counts. As shown in Fig. 2, R_t is structurally aligned with diff-based file d, providing a coherent format for the Bug Locator to identify and localize errors. The self-debug prompt is then constructed as:

• **Instruction**: Given a programming question and a corresponding piece of buggy code written in <language>, please provide a program repair proposal for the buggy code. Use '-' to represent the line that may need to be deleted or modified.

• **Programming Task**: q
• **Buggy Code**: c
• **Execution Information of Failed Test Case**:
I/O data: D_t
Program Trace Information: R_t

The objective of self-debug learning is to minimize the negative log-likelihood of the `Code Diff` file d by utilizing the prompt:

$$\mathcal{L}_{\text{BL}} = -\sum_{(q,c,t,d)\sim\mathcal{D}} \log P_\theta(d|q,c,D_t,R_t), \tag{2}$$

where all repair instances (q, c, t, d, y) form the dataset $\mathcal{D}$, and P_θ represents the probability distribution over the LLM's vocabulary.

Stage II: Program Modifier

Repairing code with few modifications requires understanding the modification process of buggy code. However, LLMs typically struggle with this process since they may not have the knowledge to make informed decisions about which changes to make in order to repair the code effectively. To address this challenge, we introduce location-aware repair learning, which directs the Program Modifier to focus on buggy areas identified in the first stage. Recognizing the possibility of incorrect bug locations produced by the Bug Locator, we propose a hybrid training strategy to prevent the Program Modifier from making unnecessary modifications, thereby improving repair accuracy. Additionally, to further reduce the extent of modifications, we train the Program Modifier to align with the preference for fewer changes through adaptive preference learning.

Location-Aware Repair Learning. To explicitly capture the modification process, we propose Location-Aware Repair Learning, which trains the Program Modifier to make precise fixes by focusing on identified buggy areas. Given the bug locations and correct code, we guide the Program Modifier to make corrections without altering unrelated code.

Specifically, the Program Modifier ϕ is trained using supervised learning to predict the correct code y as follows:

$$\mathcal{L}_{\text{supervised}} = -\sum_{(q,c,d,y)\sim\mathcal{D}} \log P_\phi(y|q,c,d), \tag{3}$$

where $P_\phi \in \mathbb{R}^{|\mathcal{V}|}$ is the probability distribution on the LLM's vocabulary.

Hybrid Training for Selective Reference. The Bug Locator may generate incorrect bug locations, potentially leading the Program Modifier to fail in fixing the bugs. To address this issue, we propose a hybrid training strategy for selective reference, which further trains the Program Modifier to repair code based on bug locations that may be incorrect. The training strategy enhances the Program Modifier's selective reference to bug locations provided by the Bug Locator instead of blindly modifying them.

Specifically, the dataset $\mathcal{D}$ is split into $\mathcal{D}_1$ and $\mathcal{D}_2$, with the data volumes satisfying $|\mathcal{D}_1| : |\mathcal{D}_2| = 1 : k$, where k is ratio parameter. Each instance $(q, c, d, y) \in \mathcal{D}_2$ is processed by the Bug Locator θ to generate new labels $\hat{d}$:

$$\hat{d} = \text{LLM}_\theta(q, c, D_t, R_t). \tag{4}$$

Then we construct a new dataset $\mathcal{D}_2'$:

$$\mathcal{D}_2' = \{(q, c, d, \hat{d}, y) \mid (q, c, d, y) \in \mathcal{D}_2\}. \tag{5}$$

The loss function of selective reference for the Program Modifier is:

$$\mathcal{L}_{\text{selective}} = -\sum_{(q,c,d,\hat{d},y)\sim\mathcal{D}_2'} (\log P_\phi(y|q,c,d) + \log P_\phi(y|q,c,\hat{d})). \tag{6}$$

We jointly train the Program Modifier using supervised learning data and selective learning data to maintain its repair capability and enhance its selective reference ability:

$$\mathcal{L}_{\text{Hybrid}} = \mathcal{L}_{\text{supervised}}(\mathcal{D}_1) + \mathcal{L}_{\text{selective}}(\mathcal{D}_2'), \tag{7}$$

where $\mathcal{L}(\cdot)$ denotes the loss function applied to the respective dataset during training.

[3]https://docs.python.org/3/library/traceback.html

Adaptive Preference Learning. Even when fixing the same bug, different methods can result in varying extents of code changes. To prioritize fewer modifications during the repair process, we draw inspiration from Direct Preference Optimization (DPO) (Rafailov et al. 2024), which steers LLMs to match specific preferences. Building on DPO, we propose an adaptive preference learning mechanism that guides LLMs to reduce the extent of code modifications further. Given two codes, differing in the extent of modifications, we utilize preference learning to guide the Program Modifier in learning preference for fewer modifications, as shown in Fig. 2.

Specifically, we obtain preference pairs (y^+, y^-), representing the preferred (*i.e.*, the correct version with fewer modifications) and dispreferred (*i.e.*, the incorrect version with more extensive modifications) codes generated by the Program Modifier ϕ after running the unit test.

The preference set $\mathcal{D}_p$ consists of preference pairs (q, c, d, y^+, y^-).

Based on the preference set $\mathcal{D}_p$, we apply DPO-Positive learning (Pal et al. 2024) to enhance the Program Modifier ϕ, iterating on its training to derive ϕ^* that prioritize repairs requiring fewer modifications. Formally, the training objective of Program Modifier ϕ^* is defined as:

$$\mathcal{L}_{\text{repair}}(\phi^*; \phi) = -\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}_p} \log \sigma[r(x, y^+) - r(x, y^-) - g(x, y^+)], \tag{8}$$

where σ denotes the logistic function, (q, c, d) is simplified as x, and r is the reward function on the generated code implicitly defined by ϕ^* and ϕ, with a hyperparameter β to control the deviation from ϕ as:

$$r(x, y) = \beta \log \frac{P_{\phi^*}(y|x)}{P_{\phi}(y|x)}. \tag{9}$$

And g denotes the penalty term within the log-sigmoid to encourage maintaining a high log-likelihood of the preferred code:

$$g(x, y^+) = \lambda \cdot \max(0, \log \frac{P_{\phi}(y^+|x)}{P_{\phi^*}(y^+|x)}), \tag{10}$$

where λ is a hyperparameter. After training, the Program Modifier is optimized to increase the probability of generating the preferred code, thereby achieving effective repairs with fewer changes.

Experiments

Experimental Setups

Dataset. We construct the first dataset, named **ACPR** (Accuracy-Consistency Program Repair) for our AdaPR task, which aims to evaluate the generated patches from accuracy (*i.e.*, fixing bugs correctly) and consistency (*i.e.*, minimizing modifications). Specifically, our dataset is collected from CodeNet (Puri et al. 2021), which contains submissions of programming problems from different users. For a given buggy program, we pair it with a randomly selected failed test case (a test case includes a test input and an expected output) as well as a passed program from the same user's submission for the same programming problem, making a $\langle buggy\ code, failed\ test\ case, passed\ code\rangle$ triplet sample. The whole dataset contains 52,168 triplet data samples. We then split our dataset into train/validation/test sets by the ratio of 8:1:1, ensuring that any particular programming problem appears in only one of them to avoid data leakage problems. To prevent overfitting code data from the same programming problem and to ensure fairness in evaluation, we balance the dataset by capping the maximum number of pairs per problem at 150/10/20 in the train/validation/test sets. The overall statistics of the dataset are given in Table 1. Further details can be found in the Appendix.

Sets	triplet	problems	users	information per problem		
				triplet	code lines	test cases
train	50023	688	13701	73	28	80
val	927	110	783	8	34	87
test	1668	110	1247	15	40	84

Table 1: Dataset Statistics.

Evaluation Metrics. To thoroughly evaluate a model's performance regarding our AdaPR task, we adopted the following evaluation metrics: (1) **Code Accuracy Rate (Acc)**: It represents the percentage of code that successfully passes all test cases of the programming problem (Muennighoff et al. 2023). (2) **Code Improvement Rate (Improve)**: This metrical measures the average improvement rate for each piece of buggy code. It calculates the proportion of additional test cases passed after the buggy code is modified. The calculation equation for the improvement rate of the i-th fixed code follows:

$$I_i = \frac{\chi(\mathcal{A}) \times n}{m}, \tag{11}$$

where $\chi(\cdot)$ is an indicator function that returns 1 if the condition inside the parentheses is true, and 0 otherwise. $\mathcal{A}$ is true if all previously passing test cases still pass after the code modification, and false otherwise. n denotes the number of cases that additional pass after repair and m represents the number of test cases that failed previously. The value of i-th fixed code is I_i if the code passes all test cases passed by the buggy code, and 0 otherwise. (3) **Failed Repair Rate (FR)**: It counts the proportion of the generated code that fails to pass the previously passed cases, which is calculated as follows:

$$FR = \frac{\sum_{i=1}^{|D|} \chi(\mathcal{B}_i)}{|D|}, \tag{12}$$

where $|D|$ is the number of pieces of code, $\chi(\cdot)$ is an indicator function. $\mathcal{B}_i$ is true if the i-th piece of code causes the previously passed cases to fail, and false otherwise. (4) **Code Consistency Rate (CCR)**: It calculates the proportion of lines of code that are preserved after modification. It is defined as follows:

$$Consistency = \frac{r}{k}, \tag{13}$$

where k indicates the total number of code lines in the fixed code, and r indicates the number of code lines preserved in the after-modification code.

Repair	Framework Structure	Methods	Bug Localization	Acc	Improve	CCR	FR
CodeLlama	End-to-end	Instruction	-	10.67	11.67	54.96	30.70
		CoT	-	10.43	11.78	57.06	33.33
		Few-shot Learning	-	9.35	10.50	40.20	31.18
		Fine-Tuning	-	28.12	30.34	51.16	14.21
	Two-stage	$AdaPatcher_{CC}$	CodeLlama	31.95	33.74	57.18	16.97
		$AdaPatcher_{GC}$	GPT-4o	**33.81**	**35.80**	**62.22**	17.74
GPT-4o	End-to-end	Instruction	-	62.77	64.11	27.40	34.65
		CoT	-	44.90	45.87	26.95	53.12
		Few-shot Learning	-	48.74	40.36	30.50	49.04
	Two-stage	$AdaPatcher_{GG}$	GPT-4o	63.31	65.47	**50.08**	14.07
		$AdaPatcher_{CG}$	CodeLlama	**67.57**	**69.81**	48.69	**12.83**

Table 2: Evaluation results on the ACPR dataset. All results in the table are reported in percentage (%).

Baselines. To evaluate the effectiveness of our model on the AdaPR task, we build $\boldsymbol{AdaPatcher}$ based on popular LLMs, including both closed-source and open-source models. For the closed-source baseline, one high-performance model **GPT-4o** (OpenAI 2024; Achiam et al. 2023) is considered. For the open-source baseline, we utilize the **CodeLlama-Instruct-7B** (Roziere et al. 2023), which is a popular foundation model for code-related tasks. We use GPT-4o and CodeLlama for stage I (*i.e.*, bug localization) and stage II (*i.e.*, program repair) respectively, denoted as $\boldsymbol{AdaPatcher_{GC}}$, $\boldsymbol{AdaPatcher_{GG}}$ and $\boldsymbol{AdaPatcher_{CG}}$, $\boldsymbol{AdaPatcher_{CC}}$ respectively. All the baselines adopt an end-to-end framework to perform the program repair task. Additionally, we incorporate baselines with three widely used LLM-based optimization methods: (1) **Chain-of-Thought (CoT)**: CoT prompting (Kojima et al. 2023) elicits complex multi-step reasoning to enhance the model's cognitive capabilities on program repair tasks. (2) **Few-Shot Learning**: Few-shot prompting (Brown et al. 2020) utilizes LLMs' in-context learning abilities to achieve high performance with input-output pairs as extra context. (3) **Fine-Tuning**: LoRA (Hu et al. 2021) injects trainable rank decomposition matrices into LLMs, updating weights based on supervised labels for the program repair task.

Experimental Results

RQ1. Effectiveness Evaluation. In this research question, we want to evaluate the effectiveness of our approach on the AdaPR task. Table 2 shows the experimental results of our approach and baselines on our test set. It is obvious that: (1) Regarding program repair accuracy, our approach (*e.g.*, $AdaPatcher_{GC}$, $AdaPatcher_{CG}$) outperforms other baselines (*e.g.*, CoT, Few-shot Learning and Fine-tuning) by a large margin. The superior performance is due to our Bug Locator's capability to precisely identify the bug locations in the first stage. During the first stage, $\boldsymbol{AdaPatcher}$ pinpointed the exact buggy line(s) by utilizing the self-debug learning from code-diff samples, enhancing our approach's ability to effectively handle *where to fix* challenge. (2) Regarding the program repair consistency, the advantages of our approach over other baselines are also obvious. For example, the best consistency score achieved by baseline (*i.e.*, CodeLlama Fine-Tuning) is 51.16%, our $AdaPatcher_{GC}$ achieved a consistency ratio of 62.22%, significantly outperforming other baseline models. We attribute this improved consistency to the effectiveness of the Program Modifier in the second stage. During the second stage, we design three key techniques (*i.e.*, Location-Aware Repair learning, Hybrid Training for Selective Reference and Adaptive Preference Learning) to ensure the consistency between the fixed code and buggy code. **Overall, our two-stage framework shows stable and substantial improvements compared to the end-to-end program repair framework, validating the effectiveness of our two-stage approach for the AdaPR task.** (3) Additionally, $\boldsymbol{AdaPatcher}$ achieves better program repair accuracy when we use GPT-4o in the second stage (*i.e.*, program repair), and achieves better program repair consistency when we use CodeLlama in the same stage. This may be because GPT-4o has an advantage over CodeLlama regarding accuracy due to its significantly larger model parameters. At the same time, CodeLlama has its own strength in identifying bug locations after fine-tuning. Therefore, when we choose CodeLlama for the first stage and GPT-4o for the second stage, the optimal performance is obtained.

Model	Acc	Improve	CCR
$\boldsymbol{AdaPatcher_{CC}}$	31.95	33.74	**57.18**
w/o Self-Debug Learning	31.89	33.27	57.06
w/o Location-Aware Repair Learning	28.05	29.61	54.39
w/o Hybrid Training	29.38	31.03	54.35
w/o Adaptive Preference Learning	32.07	33.85	56.78
$\boldsymbol{AdaPatcher_{CG}}$	**67.57**	**69.81**	**48.69**
w/o Self-Debug Learning	65.71	68.11	48.09

Table 3: Ablation study.

RQ2. Ablation Study. In this RQ, we conduct an ablation study to assess the contribution of different techniques by systematically removing each component from our approach. In particular, for $\text{AdaPatcher}_{\text{CC}}$, we remove key components (*i.e.*, Self-Debug Learning, Hybrid Training, and Preference Learning) separately. For $\text{AdaPatcher}_{\text{CG}}$, we remove the sole component (*i.e.*, Self-Debug Learning) since GPT-4o is close-sourced. The experimental results are illustrated in Table 3, we can see that: (1) Removing Self-Debug Learning, Location-Aware Learning, or Hybrid Training, results in a decline in the performance of accuracy

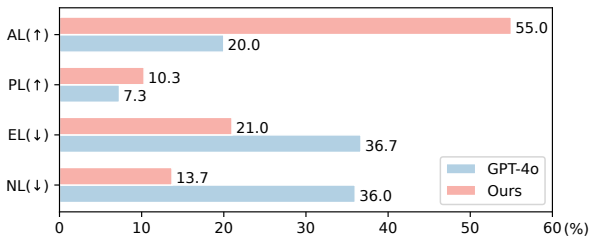

Figure 3: The statistical result of the human study.

and consistency, which signals the importance and effectiveness of these components. (2) Although Preference Learning causes a slight decrease in the metric of code accuracy, it further enhances the consistency, indicating this component can effectively reduce modifications during code repair.

RQ3. Why Our Approach Works/Fails. We manually inspected test cases where our approach worked and failed. Specific analysis and examples are provided in the Appendix. We demonstrate a buggy code fixed by our approach. Our Bug Locator first precisely identifies the buggy code line and then our program modifier fixes this bug by slightly changing this buggy line. The effectiveness of two-stage framework (Bug Locator + Program Modifier) ensures the correctness and consistency of our generated code patches. We also inspected a number of cases where ***AdaPatcher*** failed to handle. We summarize two common failed situations. One common failed situation is that the failed test case does not provide sufficient information to precisely identify the bug locations. Another bad situation is that the buggy code is too complicated or subtle for ***AdaPatcher*** to learn. For example, complicated bugs may require developers to make code changes across different sub-modules.

RQ4. Human Study for Bug Localization. The bug localization in stage one plays an important role for guiding the subsequent program repair process. Therefore, in this RQ, we conduct a human study to manually evaluate stage one's performance. We compare the bug localization capability of GPT-4o and CodeLlama trained with our framework with human evaluation. Specifically, 900 samples are provided to 2 experienced evaluators, each evaluator is asked to determine the bug locations independently, the first author is then involved in leading a discussion when they have disagreements. Following that, we compare the model-predicted bug locations with human-identified locations in terms of the following aspects: (1) **Accurate Localization** (AL) refers to that model-predicted locations match human-identified locations precisely. (2) **Partial Localization** (PL) indicates that only part of model-predicted locations match human-identified locations. (3) **Erroneous Localization** (EL) indicates model-generated locations do not match human-identified locations at all. (4) **No Localization** (NL) denotes that no bugs are identified by models. Fig. 3 illustrated the human study results. The CodeLlama trained with our framework performs better than GPT-4o in bug localization, with 35.0% and 3.0% more examples of AL and PL respectively, while having 15.7% and 22.3% fewer EL and NL examples. The results show that even GPT-4o fails to identify bug locations of a buggy code effectively, verifying the challenge of this task. Our Bug Locator, enables a small-scale LLM (*i.e.*, CodeLlama) to achieve a much superior performance than GPT-4o, validating the effectiveness of our self-debug learning with code diff samples.

Methods	Acc	Improve	FR
Hybrid Training	**32.07**	**33.85**	**16.73**
Supervised Training	27.88	29.76	26.92
Weakly Supervised Training	31.60	33.61	18.94
Supervised and Weakly Supervised	31.06	32.23	17.33

Table 4: Evaluation results of different training methods.

RQ5. Hybrid Training Analysis. To verify the effectiveness of our hybrid training method for selective reference, in this RQ, we conducted a comparative analysis against other common training methods. Particularly, we further designed our experiments to combine both weakly supervised data and supervised data in the training process. As illustrated in Table 4, the experimental results show that: (1) Our hybrid training is superior to other training methods in various metrics of the correctness of program repair. (2) Compared to supervised training, the correctness of our method has significantly improved, demonstrating the effectiveness of Hybrid Training in avoiding blind modification. (3) Compared to supervised and weakly supervised training, the experimental results demonstrate the effectiveness of our hybrid training regardless of the amount of training data.

Future Work

Several limitations are concerned with our work. Firstly, our study is based on Python, which is one of the most popular programming languages used by developers. However, our approach is language-independent, we believe our approach can be easily adapted to other programming languages Secondly, the correctness of the generated code is affected when our model is applied by using adaptive preference learning. Exploring effective ways to generate repaired code with reduced modifications while further improving its correctness is an interesting research topic for our future work.

Conclusion

This research aims to generate fixed code while requiring minimal modifications. To perform this novel task, we propose an approach ***AdaPatcher*** that utilizes self-debug learning to train a Bug Locator to accurately identify bugs and fix code through bug locations. For program repair, we train a Program Modifier through location-aware repair learning. Then we propose hybrid training to effectively avoid blindly modifying incorrect bug locations. Additionally, adaptive preference learning is used to learn fewer modifications. The experimental results show the effectiveness of our approach for this task. We hope our study lays the foundations for this new research and provide valuable insights into the potential for bug location and adaptive program repair capabilities of Open-source and closed-source LLMs.

Acknowledgements

This work was sponsored by the National Natural Science Foundation of China (No.62307032, No.62037001), the Key Research and Development Program of Zhejiang Province (No.2024C03270) and CCF-Zhipu Large Model Innovation Fund (No.CCF-Zhipu202409).

References

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Bouzenia, I.; Ding, Y.; Pei, K.; Ray, B.; and Pradel, M. 2023. TraceFixer: Execution trace-driven program repair. *arXiv preprint arXiv:2304.12743*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *NeurIPS*.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. D. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Dai, Z.; Yao, C.; Han, W.; Yuanying, Y.; Gao, Z.; and Chen, J. 2024. MPCoder: Multi-user Personalized Code Generator with Explicit and Implicit Style Representation Learning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 3765–3780.

Dissanayake, N.; Jayatilaka, A.; Zahedi, M.; and Babar, M. A. 2022. Software security patch management-A systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology*, 144: 106771.

Fan, Z.; Gao, X.; Mirchev, M.; Roychoudhury, A.; and Tan, S. H. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1469–1481. IEEE.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515*.

Jiang, N.; Liu, K.; Lutellier, T.; and Tan, L. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1430–1442. IEEE.

Jin, M.; Shahriar, S.; Tufano, M.; Shi, X.; Lu, S.; Sundaresan, N.; and Svyatkovskiy, A. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1646–1656.

Kojima, T.; Shixiang, S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2023. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*.

Krasner, H. 2021. The cost of poor software quality in the US: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Lin, W.; Feng, Y.; Han, W.; Jin, T.; Zhao, Z.; Wu, F.; Yao, C.; and Chen, J. 2024. E^3: Exploring Embodied Emotion Through A Large-Scale Egocentric Video Dataset. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Miceli-Barone, A. V.; Barez, F.; Konstas, I.; and Cohen, S. B. 2023. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python. *arXiv preprint arXiv:2305.15507*.

Moon, S.; Song, Y.; Chae, H.; Kang, D.; Kwon, T.; Ong, K. T.-i.; Hwang, S.-w.; and Yeo, J. 2023. Coffee: Boost your code llms by fixing bugs with feedback. *arXiv preprint arXiv:2311.07215*.

Muennighoff, N.; Liu, Q.; Zebaze, A.; Zheng, Q.; Hui, B.; Zhuo, T. Y.; Singh, S.; Tang, X.; Von Werra, L.; and Longpre, S. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Nam, D.; Macvean, A.; Hellendoorn, V.; Vasilescu, B.; and Myers, B. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

Ni, A.; Allamanis, M.; Cohan, A.; Deng, Y.; Shi, K.; Sutton, C.; and Yin, P. 2024. NExT: Teaching Large Language Models to Reason about Code Execution. *arXiv preprint arXiv:2404.14662*.

Olausson, T. X.; Inala, J. P.; Wang, C.; Gao, J.; and Solar-Lezama, A. 2023. Is Self-Repair a Silver Bullet for Code Generation? In *The Twelfth International Conference on Learning Representations*.

OpenAI. 2024. ChatGPT-4o. https://openai.com/index/hello-gpt-4o.

Pal, A.; Karkhanis, D.; Dooley, S.; Roberts, M.; Naidu, S.; and White, C. 2024. Smaug: Fixing failure modes of preference optimisation with dpo-positive. *arXiv preprint arXiv:2402.13228*.

Parreira, M. T.; Gillet, S.; and Leite, I. 2023. Robot Duck Debugging: Can Attentive Listening Improve Problem Solving? In *Proceedings of the 25th International Conference on Multimodal Interaction*, 527–536.

Paul, R.; Hossain, M. M.; Siddiq, M. L.; Hasan, M.; Iqbal, A.; and Santos, J. 2023. Enhancing automated program repair through fine-tuning and prompt engineering. *arXiv preprint arXiv:2304.07840*.

Puri, R.; Kung, D. S.; Janssen, G.; Zhang, W.; Domeniconi, G.; Zolotov, V.; Dolby, J.; Chen, J.; Choudhury, M.; Decker, L.; et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Rafailov, R.; Sharma, A.; Mitchell, E.; Manning, C. D.; Ermon, S.; and Finn, C. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Shahriar, H.; and Zulkernine, M. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44(3): 1–46.

Sobania, D.; Briesch, M.; Hanna, C.; and Petke, J. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 23–30. IEEE.

Wu, T.; Li, M.; Chen, J.; Ji, W.; Lin, W.; Gao, J.; Kuang, K.; Zhao, Z.; and Wu, F. 2024. Semantic Alignment for Multimodal Large Language Models. In *Proceedings of the 32nd ACM International Conference on Multimedia*, 3489–3498.

Xia, C. S.; Wei, Y.; and Zhang, L. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 1482–1494. IEEE.

Xia, C. S.; and Zhang, L. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 959–971.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Ye, H.; Martinez, M.; Luo, X.; Zhang, T.; and Monperrus, M. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.

Zhang, J.; Panthaplackel, S.; Nie, P.; Li, J. J.; and Gligoric, M. 2022. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–12.

Zhang, K.; Li, Z.; Li, J.; Li, G.; and Jin, Z. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.