Learner-Tailored Program Repair: A Solution Generator with Iterative Edit-Driven Retrieval Enhancement

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

[1]Zhejiang University

[2]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

{zhenlongdai, zhuoluozhao, whning, tangxiu, wusai, changy, zhipeng.gao, jingyuanchen}@zju.edu.cn

Abstract

With the development of large language models (LLMs) in the field of programming, intelligent programming coaching systems have gained widespread attention. However, most research focuses on repairing the buggy code of programming learners without providing the underlying causes of the bugs. To address this gap, we introduce a novel task, namely **LPR** (**L**earner-Tailored **P**rogram **R**epair). We then propose a novel and effective framework, **LSGEN** (**L**earner-Tailored **S**olution **G**enerator), to enhance program repair while offering the bug descriptions for the buggy code. In the first stage, we utilize a repair solution retrieval framework to construct a solution retrieval database and then employ an edit-driven code retrieval approach to retrieve valuable solutions, guiding LLMs in identifying and fixing the bugs in buggy code. In the second stage, we propose a solution-guided program repair method, which fixes the code and provides explanations under the guidance of retrieval solutions. Moreover, we propose an Iterative Retrieval Enhancement method that utilizes evaluation results of the generated code to iteratively optimize the retrieval direction and explore more suitable repair strategies, improving performance in practical programming coaching scenarios. The experimental results show that our approach outperforms a set of baselines by a large margin, validating the effectiveness of our framework for the newly proposed LPR task.

Code — https://github.com/PandaAB/LSGen

Introduction

As the significance of computer science and programming increases across various fields, the learning of programming has garnered widespread attention (Gulwani, Radiček, and Zuleger 2018; Zhang et al. 2022). Nowadays, inspired by the promising performance of large language models (LLMs) for code (Dai et al. 2024), researchers have applied code LLMs to intelligent tutoring for programming, utilizing them to help programming learners correct their programs. However, most studies have focused on generating the correct patches (Koutcheme, Dainese, and Hellas 2024; Koutcheme et al. 2025; Phung et al. 2023), and explanations for the causes of bugs are often overlooked, failing to meet the needs of actual programming learners. Consider the following practical scenarios in Fig. 1: Alice is a programming learner who attempts to fix the bug in her code by referring to the official solution on the programming platform since she doesn't know how to resolve the issue. Unfortunately, she finds that the official solution is implemented differently from her code, making it difficult for her to fix her code. The official solution adopted a dynamic programming algorithm, while she used the greedy algorithm and binary search to solve the problem of Longest Increasing Subsequence. Therefore, Alice tried to use LLM to help her fix these bugs. LLM provides a comprehensive and clear solution: a patch to correct the code (*e.g.*, colored in green) and a bug description (*e.g.*, colored in yellow) explaining why the modification is needed (*e.g.*, colored in red). As a result, Alice can understand the changes, knowing why and how to fix her code.

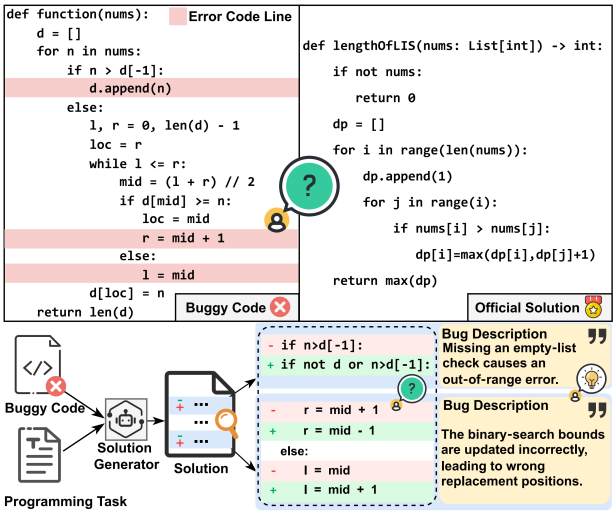

Figure 1: Example of LPR. The generated solution contains the repaired code and the corresponding bug description.

There is limited research in investigating how to generate solutions that contain repair code and the correspond-

*Both authors contributed equally to this research.

†Co-corresponding authors.

ing bug explanation. To address this gap, we propose a new task in this paper, namely **Learner-Tailored Program Repair**, denoted as **LPR**. This task aims to generate solutions that contain the fixed code and the corresponding bug description for the programming learner. LPR is a non-trivial task regarding the following key is a challenging task: (i) **Bugs in code written by programming learners are hard to identify and explain.** Unlike the code written by professional programmers, the diverse coding styles, poor naming conventions, and chaotic implementations of programming learners make it difficult to identify and understand bugs in buggy code. Even after discovering the issue, understanding why the bug occurs and how it affects the program's behavior can be challenging. (ii) **The various and complex bugs in code written by programming learners are hard to fix.** Due to the open-ended nature of programming problems, there are multiple possible approaches to solving the same problem. Unlike providing pre-written code, fixing bugs in different approaches demands a deeper understanding of the diverse programming patterns and problem-solving strategies employed by various users. (iii) **Evaluating bug descriptions for buggy code is challenging**. Unlike code correctness, which can be evaluated by executing test cases, there is no automatic evaluation metric to estimate bug descriptions. Evaluating the correctness of bug descriptions is a time-consuming manual process, hindering the research about personalized programming coaching. How to evaluate the correctness of bug descriptions quantitatively becomes another challenge for our study.

To tackle the above challenges, we propose a novel and effective framework named **LSGEN**, which is designed to generate repair code and the corresponding bug descriptions for programming users. First, we propose a Repair Solution Retrieval Framework to provide high-quality solution data and then leverage an edit-driven approach to obtain similar and valuable solutions for program repair. To address the challenge of identifying and explaining bugs, we propose a reference-inspired solution generation approach that integrates diff analysis and textual bug descriptions from retrieval code pairs, directs LLMs to capture code modifications and their underlying causes. To address the challenge of fixing various complex bugs in the code of programming learners, we propose an iterative retrieval enhancement method, which iteratively retrieves repair strategies that match the current generated incorrect code, thereby improving both usability and repair performance. To address the challenge of evaluating bug descriptions, we propose an automatic evaluation metric that utilizes LLMs to achieve fine-grained logical consistency assessment. In summary, our paper makes the following contributions: (1) Current research mainly focuses on repairing the buggy code. To the best of our knowledge, no prior work has deeply explored how to generate the repaired code and the corresponding bug description for programming learners. (2) We propose an automatic evaluation metric for estimating the correctness of bug descriptions. (3) We propose a novel and effective framework, named LSGEN, that leverages the submission and evaluation system of the Programming platform to generate the repaired code and the corresponding bug description. The experimental results show that our model outperforms a set of baselines, demonstrating its strong performance and practical usability. We hope our study can lay the foundations for this research topic.

Related Work

Recent advancements in LLMs (Dong, Chen, and Wu 2025a,b) have spurred their integration into automatic program repair in intelligent tutoring (Zhang et al. 2022; Koutcheme et al. 2024). Early work mainly focused on directly prompting LLMs to generate correct code (Koutcheme et al. 2023; Phung et al. 2024). Cref (Yang et al. 2024) and TreeInstruct (Kargupta et al. 2024) leverage multi-turn conversation to guide LLM in automatically repairing bugs. FastFixer (Liu et al. 2024) improves repair accuracy by fine-tuning. Recent research has adopted the retrieval-augmented method to improve the accuracy of program repair. PAR (Zhao et al. 2024) and PyDex (Zhang et al. 2024) both repair code by retrieving similar correct code based on test cases. PyFiXV (Phung et al. 2023) performs code repair by retrieving examples based on edit distance. Additionally, MMAPR (Zhang et al. 2022) and ASSIST (Van Praet, Hoobergs, and Schrijvers 2024) employ a hybrid of syntactic and logical repair to guide the LLM. Existing approaches to automated program repair for learners mainly focus on generating the correct program, but neglect personalized repair needs. Learners not only require a correct program but also seek to understand the root causes of their bugs. Recent studies have begun to explore the capability of LLMs to generate bug descriptions for learners (Koutcheme, Dainese, and Hellas 2024). However, current evaluation approaches depend on manual assessment (Koutcheme et al. 2025), which is time-consuming. To remedy this gap, we propose an automatic evaluation metric that automatically assesses the quality of generated bug descriptions.

Methodology

In this section, we introduce a generic and effective framework, **LSGEN**, aimed at enhancing program repair while providing the bug descriptions for the program learner. In the first stage, we utilize a repair solution retrieval framework to construct a solution retrieval database and then employ an edit-driven solution retrieval approach to retrieve valuable solutions. In the second stage, we propose a solution-guided program repair method, which generates the repair solution based on diff-based program analysis and textual bug description from candidate solutions. Moreover, we propose an iterative repair enhancement approach that utilizes the evaluation results of the program to iteratively optimize the retrieval direction and explore more suitable repair methods for the buggy code, improving performance in practical programming learning scenarios.

Task Definition

Given a specific programming problem q, a buggy code c, and a collection D of historical submissions from other users, the objective is to generate a solution s that contains

the fixed version y of the buggy code c along with corresponding bug descriptions $\mathcal{B}$.

Stage I: Repair Solution Retrieval Framework

Programming platforms contain a large number of submissions for the same programming problem, including incorrect submissions with bugs similar to the user's code and correct submissions that may contain potential solutions. These submissions could provide code LLMs with valuable reference solutions to enhance the performance of program repair. However, the open-ended nature of programming problems and differences in implementation approaches make precise and effective retrieval of solutions to buggy code challenging (Wang et al. 2021). To address this challenge, we propose a repair solution retrieval framework that first constructs a solution database to provide high-quality solution data and then leverages an edit-driven approach to obtain similar and valuable solutions for program repair.

Solution Retrieval Database Construction. To effectively retrieve valuable repair solutions for buggy code, we first construct a high-quality retrieval database from the historical submissions. Specifically, as illustrated in Fig. 2(a), for a given programming problem q, we obtain a sequence of submissions for this problem from each user by sorting them based on submission time. For an incorrect submission, we consider all subsequent correct submissions made after the submission time as potential fixes. Inspired by the previous study (Dai et al. 2025), we retain only the pair with the highest consistency score that exceeds the threshold A from each user's sequence of submissions. The high-quality retrieval database is constructed as follows:

$$D_p = \{(c_w, c_r) \mid \underset{(c_w,c_r)\in P, P\in D}{\arg\max} F_{\text{diff}}(c_w, c_r) \geq A\}, \quad (1)$$

where D_p is the retrieval database, P represents the code pairs of a user, each $p \in P$ consists of an incorrect and a correct code pair (c_w, c_r). $F_{\text{diff}}(\cdot)$ is the function for calculating the consistency between incorrect and correct code as follows:

$$F_{\text{diff}}(c_w, c_r) = \frac{R(c_w, c_r)}{K(c_r)}, \quad (2)$$

where K indicates the total number of code lines in the fixed code c_r. $R(c_w, c_r)$ indicates the number of code lines preserved in the after-modification code, which is implemented by the diff tool (*i.e.*, git[1]).

Edit-driven Solution Retrieval. The diversity of solutions to programming problems makes the implementation ideas of programs potentially different, even when the distribution of programs' test results (*i.e.*, passing and failing cases) is the same. This presents challenges in identifying similar solutions for the buggy code. To address this challenge, we propose an edit-driven solution retrieval approach that leverages the vectorized code editing process between buggy code and its fixed version to search for repair solutions.

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

Specifically, as illustrated in Fig. 2(b), for a programming problem q, the code edit representation is achieved by vectorizing the code editing process as follows:

$$h_* = \text{CodeEncoder}(c_*), \quad (3)$$

$$h_p = h_{c_r} - h_{c_w}, \quad (4)$$

where h_{c_r} and h_{c_w} represent the vector representation of the code c_r and c_w through the code encoder, respectively. h_p is the code edit representation of a program pair (c_r, c_w). For a buggy code c, we first vectorize the code as h_c through the code encoder. Then we select the top-k pairs with the closest vector distance between the virtual fixed version of buggy code c and the correct code c_r as follows:

$$h'_{c_r} = h_c + h_p, \quad (5)$$

$$D_k = \{(c_w, c_r) \mid \underset{(c_w,c_r)\in D_{p,q}}{\text{argmax}_k} F_{\text{smi}}(h'_{c_r}, h_{c_r})\}, \quad (6)$$

where D_k denotes the top-k program repairs, $D_{p,q}$ represents the program pairs belonging to the problem q, F_{smi} is the normalized cosine similarity. The closer the virtual fixed code h'_{c_r} is to the fixed code c_r, the more likely it is that codes c and c_w will exhibit similar bugs and repair processes. The edit-driven retrieval approach can identify more similar potential solutions to the buggy code, thereby offering valuable insights for program repair.

Stage II: Solution-Guided Program Repair

While LLMs possess strong problem-solving abilities in programming, they still face challenges in modifying others' code due to the diverse programming approaches and coding preferences of different learners (Hellas et al. 2023). To address this challenge, we propose a reference-inspired solution generation approach, which directs the Solution Generator to generate the repair solution based on diff-based program analysis and textual bug description. To further improve program repair performance, we incorporate an iterative retrieval enhancement method that refines solution search and optimization using evaluation results as retrieval signals, enhancing the reliability and applicability of repairs in programming learning scenarios.

Reference-Inspired Solution Generation. Code changes are often localized and subtle, making it difficult to track their logical evolution. Therefore, LLMs struggle to capture the modifications between different versions of the code, hindering the understanding of repair strategies within similar codes to fix bugs. Additionally, the diversity of implementation methods and the distinct programming ideas of students make it very difficult to identify and understand bugs. To address this challenge, we propose a reference-inspired solution generation approach that integrates diff analysis and textual bug descriptions. This approach directs LLMs to capture code modifications and their underlying causes, providing fix codes along with bug explanations. Specifically, to better understand the repair process and explain the reason for the bugs, we utilize an LLM to explain the bugs for each code pair in D_k, as described by the following equations:

$$B = \text{LLM}(q, c_w, c_r), \quad (7)$$

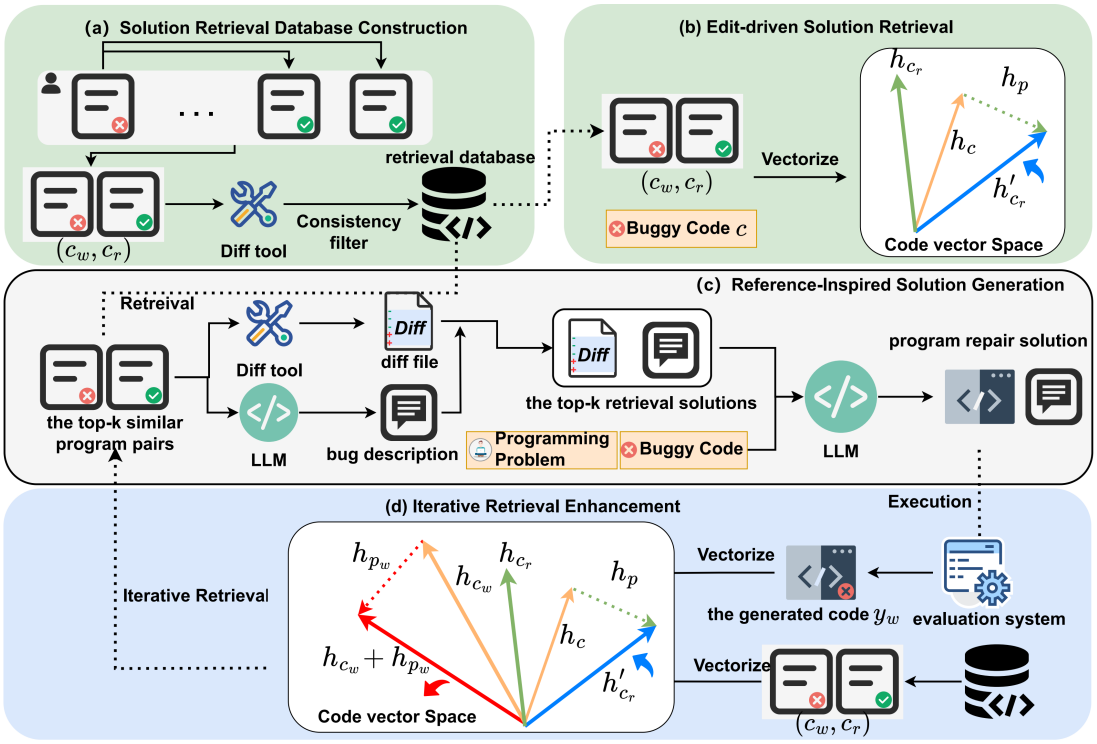

Figure 2: Overview of LSGEN. (a) Illustration of the Solution Retrieval Database Construction process. (b) Illustration of Edit-driven Solution Retrieval process. (c) Illustration of the Reference-Inspired Solution Generation process. (d) Illustration of the Iterative Retrieval Enhancement process.

where B is the textual bug description generated from the incorrect code c_w, providing direct insight into the causes of the bugs. Drawing inspiration from diff tools that help developers compare and understand patterns in code modifications, as shown in Fig. 2(c), we reformat each code pair (c_w, c_r) in the top-k retrieval solutions into a diff file $d_{w,r}$ using a diff tool to illustrate the modification process. Then, we generate solutions based on the top-k retrieval solutions that include the textual bug description and diff analysis as follows:

$$s = \text{LLM}(q, \{z_i\}_{i=1}^{k}, c), \tag{8}$$
$$z = (d_{w,r}, B), \tag{9}$$

where s is the generated solution for the buggy code c, which contains a fixed code y along with the corresponding bug descriptions $\mathcal{B}$. z is a specific solution from the retrieval dataset. The prompt for generation is constructed as:

This approach aims to effectively identify code modifications and their underlying causes in buggy code from similar solutions, ultimately enhancing automatic repair and the quality of bug descriptions.

Iterative Retrieval Enhancement. The programming platform can evaluate the user's code through its evaluation system to determine whether the user's code is correct. Inspired by this, we propose an iterative retrieval enhancement approach to enhance usability and repair performance in actual programming learning scenarios. This method iteratively uses the evaluation results of the generated code as a retrieval signal and optimizes the search direction for potential solutions in the retrieval vector space.

- **Instruction**: You are a skilled programmer experienced in debugging and providing optimal code fixes. Given a programming problem and a piece of buggy code, you are required to perform the following tasks:
1. **Fix the Buggy Code**: fix the buggy code to meet the problem's requirements, ensuring that the changes are minimal to preserve the original structure and logic as much as possible.
2. **Provide Bug Descriptions**: provide clear and complete point-by-point descriptions of the bugs present in the buggy code.

- **Programming Task**: q
- **The top-k program repairs for reference**: $\{d_{w_i,r_i}, B_i\}_{i=1}^{k}$
- **Buggy Code**: c

Specifically, we define a deviation measure function F_w to quantify the degree of deviation between the failed repair process of the generated code and the repair process of code pairs (c_w, c_r) in the solution retrieval dataset $D_{p,q}$ as:

$$F_w((c, y_w), c_r) = 1 - F_{\text{sim}}(h_{c_w} + h_{p_w}, h_{c_r}), \tag{10}$$
$$h_{p_w} = h_{y_w} - h_c, \tag{11}$$

where h_{p_w} is the code edit representation derived from the failed generated codes y_w and the buggy code c. A larger value of F_w indicates that the repair processes h_{p_w} and h_p

are more different. As shown in Fig. 2(d), we constrain the search space of the repair process through similarity and optimize the search direction based on the deviation of the generated error repair process, as described by the following equations:

$$F_{dis}((c, y_w), (c_w, c_r)) = F_w((c, y_w), c_r) + F_{\text{sim}}(h_c + h_p, h_{c_r}). \quad (12)$$

Utilizing the evaluation results of the generated code as retrieval signals, we retrieve valuable solutions for the failed generated code y_w using the function F_{dis}, and then generate solutions based on the retrieved results as follows:

$$\hat{D}_k = \{(c_w, c_r) \mid \underset{(c_w,c_r)\in D_{p,q}}{\text{argmax}_k} \; F_{\text{dis}}((c, y_w), (c_w, c_r))\}, \quad (13)$$

$$\hat{s} = \text{LLM}(q, \{z_i\}_{i=1}^{k}, c), \quad (14)$$

where $\hat{s}$ and $\hat{D}_k$ are the newly generated solution and retrieval dataset, and $z_i \in \hat{D}_k$ is a solution in the re-retrieval results. By iterating through the above process, we continuously use the evaluation results as retrieval signals to optimize the search for solutions required for error repair, thereby improving the repair performance in real-world programming learning scenarios.

Experiments

Experimental Setups

Benchmark. Existing mainstream program repair datasets for educational programming (Zhao et al. 2024; Zhang et al. 2024) lack both bug descriptions of the buggy code and a database accessible for retrieval. To address these limitations, we introduce a benchmark, named **LPR-Bench**, designed to evaluate the performance of Code LLMs in the LPR task. LPR-Bench includes students' buggy programs paired with their correct versions, detailed bug descriptions, and a large-scale retrieval database. In addition, LPR-Bench provides an automatic evaluation framework to execute code and assess the quality of generated bug descriptions. Our dataset is collected from the test set of ACPR (Dai et al. 2025) and further filtered by code length and the success rate of LLM-based repair to remove overly simple samples. We construct a retrieval database containing a rich variety of user submissions, which is collected from CodeNet (Puri et al. 2021). For the data of bug descriptions, we first use GPT-4o (OpenAI 2024) to generate initial bug descriptions. These generated bug descriptions are then reviewed and refined by three programmers with five years of programming experience. The overall statistics of the dataset and the retrieval database are given in Table 1. Further details can be found in the Appendix.

Evaluation Metrics. To comprehensively assess the performance of a method on the LPR task, we employ the following **program evaluation metrics** to measure the quality of the generated repaired code: (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):

set	user	sample	bug desc	problem	avg.problem test case
Retrieval	21,584	274,349	-	110	84
Test	306	407	912	65	89

Table 1: Dataset Statistics.

This metric 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 (Dai et al. 2025). To evaluate the quality of the generated bug descriptions, we propose an **automatic evaluation metric for bug descriptions** that determines whether the generated answers match the ground truth. Since natural language is difficult to assess, existing studies rely on time-consuming manual evaluation (Sarsa et al. 2022). To address this gap, we propose an automated evaluation metric that leverages LLMs to assess generated bug descriptions in a structured manner. Formally, given a programming task q, a buggy code c, the ground truth set of the bug descriptions $\mathcal{A} = \{a_i\}_i^u$ for c and the generated descriptions form the set $\mathcal{B} = \{b_i\}_i^v$. We ask the LLM to generate bug descriptions point by point. The metric is defined as follows:

$$M = \mathcal{K}(q, c, \mathcal{A}, \mathcal{B}), \quad (15)$$

where $\mathcal{K}(\cdot)$ is a function that measures the matching degree M between two sets of bug descriptions. For a pair of bug descriptions $(a_i \in \mathcal{A}, b_j \in \mathcal{B})$, consistency is computed as follows:

$$m_{i,j} = \mathcal{M}(q, c, a_i, b_j), \quad (16)$$

where $\mathcal{M}(\cdot)$ is a function that uses an LLM to determine whether two bug descriptions refer to the same bug in logic, which returns 1 if they are identical and 0 otherwise. Then we use Precision, Recall, and F1 (Fang et al. 2023) to evaluate the quality of the generated bug descriptions. Our evaluation metrics for bug descriptions can automatically assess generated bug descriptions at a fine-grained level, thereby reducing the comprehension burden on the LLM. Further details can be found in the Appendix.

Baselines. To evaluate the effectiveness of LSGEN, we conduct experiments on current representative models, including GPT-4o (OpenAI 2024), Claude-4[2], CodeLlama-7B (Rozière et al. 2024), and Qwen2.5-Coder-7B (Hui et al. 2024). The results of CodeLlama-7B are provided in the Appendix due to space constraints. We compare LSGEN to mainstream methods for program repair in programming coaching: (1)**NoRef**: We directly prompt the LLM to generate the fixed code and corresponding bug descriptions without extra context information. (2) **AdaPatcher** (Dai et al. 2025) (3) **PAR** (Zhao et al. 2024) (4) **PyDex** (Zhang et al. 2024) (5) **PyFiXV** (Phung et al. 2023) To evaluate the generalizability of our approach, we conduct experiments using GPT-4o on three retrieval models: Qwen3-Embedding-0.6B (Zhang et al. 2025), inf-retriever-v1 (Yang et al. 2025), and UniXcoder (Guo et al. 2022). Further implementation details can be found in the Appendix.

[2]https://www.anthropic.com/news/claude-4

Model	Method	Retrieval Method	Program		Bug Description		
			Acc	Improve	B-Precision	B-Recall	B-F1
GPT-4o	NoRef	-	18.43	22.66	8.90	12.66	9.93
	AdaPatcher	-	19.41	23.12	9.83	12.35	10.10
	PAR	PSM	40.54	43.83	20.47	29.23	22.83
	PyDex	Hamming Distance	37.10	41.61	21.17	23.33	21.15
	PyFiXV	Edit Distance	26.04	30.38	14.63	18.11	15.24
	LSGEN$_{base}$	Edit-driven Retrieval	80.59	81.80	30.20	46.18	34.20
	LSGEN$_{iter3}$	Iterative Retrieval Enhancement	**91.40**	**92.11**	**33.76**	**52.38**	**38.46**
Claude-4	NoRef	-	18.18	22.39	8.68	12.41	9.71
	AdaPatcher	-	18.43	22.47	9.09	12.55	9.72
	PAR	PSM	38.08	41.67	19.96	27.55	22.04
	PyDex	Hamming Distance	38.82	42.99	21.23	25.02	21.87
	PyFiXV	Edit Distance	42.01	45.23	22.78	30.51	24.59
	LSGEN$_{base}$	Edit-driven Retrieval	81.33	82.06	30.18	46.55	34.56
	LSGEN$_{iter3}$	Iterative Retrieval Enhancement	**91.65**	**92.02**	**33.50**	**51.55**	**38.27**
Qwen2.5-Coder-7B	NoRef	-	6.39	8.61	2.74	3.56	2.82
	AdaPatcher	-	8.35	10.71	3.81	3.36	3.40
	PAR	PSM	37.35	38.64	6.59	9.88	7.11
	PyDex	Hamming Distance	17.20	17.88	3.72	4.42	3.64
	PyFiXV	Edit Distance	8.60	10.94	3.24	2.93	2.86
	LSGEN$_{base}$	Edit-driven Retrieval	46.19	47.70	17.53	22.62	18.32
	LSGEN$_{iter3}$	Iterative Retrieval Enhancement	**57.49**	**58.47**	**21.62**	**27.50**	**22.48**

Table 2: Evaluation results on the LPR-Bench. All results in the table are reported in percentage (%). The best method is shown in boldface, and the best among the other baselines is underlined for each metric.

Implementation Details. For the retrieval model, we employ Qwen3-Embedding-0.6B. For retrieval, the number of top-k solutions is set to 5. In our experiment, the temperature of generation is 0.2. For the valuation of bug descriptions, we use GPT-4o-mini with the temperature set to 0.0. Further details can be found in the Appendix.

Experimental Results

RQ1. Effectiveness Evaluation. In this RQ, we aim to evaluate the effectiveness of LSGEN on the LPR task. Table 2 shows the experimental results of LSGEN and the baselines on LPR-Bench. We include two variants of LSGEN: LSGEN$_{base}$ (w/o Iterative Retrieval Enhancement) and LSGEN$_{iter3}$ that applies three iterations of Iterative Retrieval Enhancement. It is obvious that: (1) In terms of code accuracy, LSGEN outperforms other baselines (*e.g.*, PAR, PyDex) by a large margin. For example, LSGEN$_{iter3}$ achieves 91.40% on GPT-4o, 50.86% higher than the second-best method, and it reaches 91.65% on Claude-4, 49.64% above the second-best method. (2) In terms of bug description, LSGEN demonstrates an obvious advantage over other baselines. For example, LSGEN$_{iter3}$ achieves a B-F1 of 38.46% on GPT-4o, 15.63% higher than the second-best method, while it achieves 38.27% on Claude-4, 13.68% higher than the second-best method. LSGEN enables the LLM to generate precise bug descriptions that reflect a deeper understanding of the underlying bugs. Additionally, we conducted a case study to illustrate the effectiveness of LSGEN. Detailed analysis is provided in the Appendix. **Overall, LSGEN demonstrates superior performance compared to other baselines, validating the effectiveness of our framework for the proposed LPR task.**

RQ2. Ablation Study. In this RQ, we conduct an ablation study to assess the contribution of different techniques by removing each component from LSGEN. The specific ablation setting can be found in the Appendix. In particular, we set the number of iterations of our iterative retrieval enhancement to 1. The experimental results are illustrated in Table 3. It is obvious that: (1) Removing each component results in a decrease in the performance of accuracy and B-F1, which demonstrates the effectiveness of each component. (2) Across all models, Edit-driven Solution Retrieval delivers stable improvements on both accuracy and B-F1. (3) Removing reference-inspired solution generation results in a notable drop, particularly for smaller models (*i.e.*, Qwen2.5-Coder-7B), as this component effectively identifies code modifications and their underlying causes from similar solutions, ultimately improving accuracy and B-F1.

RQ3. Human Study for Bug Descriptions. To verify the effectiveness of our metric, in this RQ, we conduct a human study to manually assess the quality of the generated bug descriptions. We randomly select 189 samples generated by GPT-4o and pair each generated bug description with ground truth, yielding 1,390 pairs. These pairs are provided to two experienced evaluators, who independently judge whether each generated description matches the ground truth. The first author then leads a discussion to resolve any disagreements. Table 4 shows the results of the human study. At the sample level, our metric is consistent with that of hu-

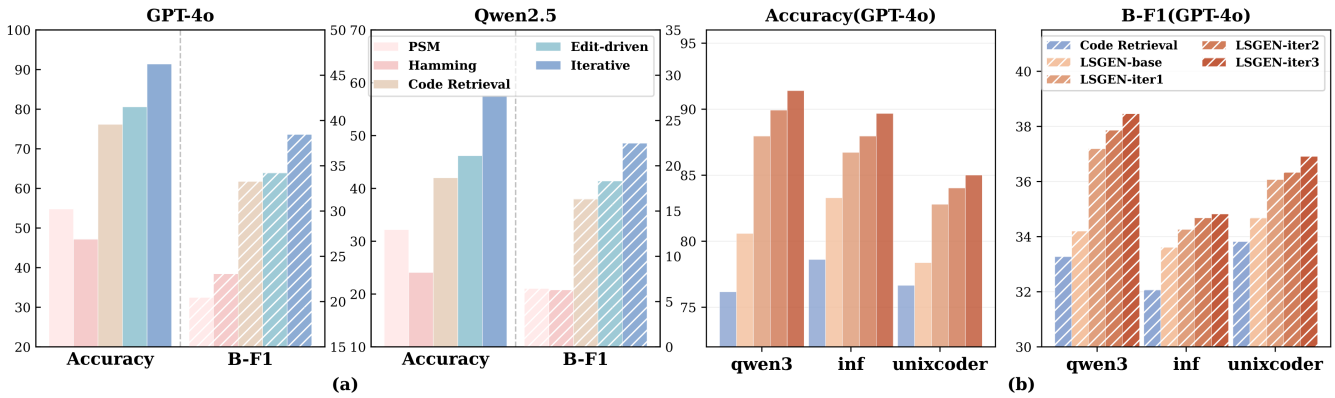

Figure 3: (a) illustrates the results of different retrieval methods on GPT-4o and Qwen2.5-Coder-7B.(b) shows the effect of varying iteration counts using different retrieval models on GPT-4o. All results in the table are reported in percentage (%).

Method	Acc	Improve	B-F1
LSGEN$_{\text{GPT-4o}}$	**87.96**	**88.53**	**37.19**
w/o Edit-driven Solution Retrieval	86.00	87.04	34.91
w/o Iterative Retrieval Enhancement	80.59	81.80	34.20
w/o Reference-Inspired Solution	77.40	78.81	32.23
LSGEN$_{\text{Claude-4}}$	**87.71**	**88.68**	**37.10**
w/o Edit-driven Solution Retrieval	86.49	87.22	34.83
w/o Iterative Retrieval Enhancement	81.33	82.06	34.56
w/o Reference-Inspired Solution	78.62	79.70	35.58
LSGEN$_{\text{Qwen2.5}}$	**51.11**	**51.93**	**19.91**
w/o Edit-driven Solution Retrieval	48.65	49.44	16.96
w/o Iterative Retrieval Enhancement	46.19	47.70	18.32
w/o Reference-Inspired Solution	25.55	26.61	7.95

Table 3: Ablation study.

man evaluators in 67.72% of cases. At the point level, this consistency is up to **93.02%**. Then, we calculate the Pearson correlation coefficient r (Zhou et al. 2024), obtaining $r = 0.848$, which indicates a significant linear correlation between the automatic metric and the human evaluation. This high consistency demonstrates the effectiveness of our automatic evaluation metric.

Level	Consistency	Inconsistency	Indeterminate
Sample	**67.72%**	21.16%	11.12%
Point	**93.02%**	5.11%	1.87%

Table 4: Human study.

RQ4. The effectiveness of Solution Retrieval. To validate the effectiveness of our proposed retrieval method, we conducted comparative experiments across different retrieval methods and retrieval models. As illustrated in Fig.3(a), we compare our method with common retrieval methods used in program repair. The iteration of our Iterative Retrieval Enhancement is set to 3. Code Retrieval refers to retrieval based on the cosine similarity between two buggy codes. The experimental result shows that: LSGEN significantly outperforms other methods in both code accuracy and bug description across open-source and closed-source models. For example, Iterative Retrieval Enhancement is 15.23% higher in code accuracy and 5.13% higher in B-F1 than Code Retrieval on GPT-4o. To evaluate the generalizability of LSGEN, we conduct experiments using three retrieval models (*e.g.*, Qwen3-Embedding-0.6B, inf-retriever-v1, and UniXcoder) on both GPT-4o and Qwen2.5-Coder-7B. As illustrated in Fig.3(b), we conduct experiments varying the number of iterations from 0 to 3. Comparing LSGEN$_{\text{iter1}}$ with LSGEN$_{\text{base}}$ shows a significant improvement in the first iteration, and LSGEN continues to improve steadily as the number of iterations increases. For example, on Qwen3-Embedding-0.6B, LSGEN$_{\text{iter1}}$ outperforms the LSGEN$_{\text{base}}$ by 7.37% in accuracy and 2.99% in B-F1. From LSGEN$_{\text{iter1}}$ to LSGEN$_{\text{iter3}}$, accuracy rises from 87.96% to 91.40%. Across various retrievers, LSGEN consistently achieves notable improvements in both accuracy and B-F1, demonstrating its generalizability. The results for Qwen2.5-Coder-7B are provided in the Appendix.

Conclusion

This study addresses the novel task of Learner-Tailored Program Repair, which aims to generate repaired code and corresponding bug descriptions for programming learners. We introduce LSGEN, an effective framework that first collects high-quality solution pairs and then uses reference-inspired solution generation to guide LLMs in capturing both code modifications and their underlying causes. It further improves performance through iterative retrieval enhancement. We also propose an automatic evaluation metric that leverages LLMs to assess the quality of generated bug descriptions. Experiments on LPR-Bench show that LSGEN outperforms a set of baselines by a large margin. We hope that our study establishes a foundation for personalized programming coaching and inspires future research on integrating repair and bug description generation for practical usability.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No.62307032, No.62507040), the "Pioneer" and "Leading Goose" R&D Program of Zhejiang (2025C02022), the Ningbo "Yongjiang Talent Program" Youth Innovation Project (2024A-156-G) and CCF-Zhipu Large Model Innovation Fund (No.CCF-Zhipu202409).

References

Dai, Z.; Chen, B.; Zhao, Z.; Tang, X.; Wu, S.; Yao, C.; Gao, Z.; and Chen, J. 2025. Less is More: Adaptive Program Repair with Bug Localization and Preference Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 128–136.

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.

Dong, Z.; Chen, J.; and Wu, F. 2025a. Knowledge is Power: Harnessing Large Language Models for Enhanced Cognitive Diagnosis. arXiv:2502.05556.

Dong, Z.; Chen, J.; and Wu, F. 2025b. LLM-driven cognitive diagnosis with solo taxonomy: A model-agnostic framework. *Frontiers of Digital Education*, 2: 20.

Fang, J.; Wang, X.; Meng, Z.; Xie, P.; Huang, F.; and Jiang, Y. 2023. MANNER: A variational memory-augmented model for cross domain few-shot named entity recognition. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 4261–4276.

Gulwani, S.; Radiček, I.; and Zuleger, F. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4): 465–480.

Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; and Yin, J. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. arXiv:2203.03850.

Hellas, A.; Leinonen, J.; Sarsa, S.; Koutcheme, C.; Kujanpää, L.; and Sorva, J. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, ICER 2023, 93–105. ACM.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; Dang, K.; Fan, Y.; Zhang, Y.; Yang, A.; Men, R.; Huang, F.; Zheng, B.; Miao, Y.; Quan, S.; Feng, Y.; Ren, X.; Ren, X.; Zhou, J.; and Lin, J. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186.

Kargupta, P.; Agarwal, I.; Hakkani-Tur, D.; and Han, J. 2024. Instruct, Not Assist: LLM-based Multi-Turn Planning and Hierarchical Questioning for Socratic Code Debugging. arXiv:2406.11709.

Koutcheme, C.; Dainese, N.; and Hellas, A. 2024. Using Program Repair as a Proxy for Language Models' Feedback Ability in Programming Education. In *Workshop on Innovative Use of NLP for Building Educational Applications*, 165–181. Association for Computational Linguistics.

Koutcheme, C.; Dainese, N.; Sarsa, S.; Hellas, A.; Leinonen, J.; Ashraf, S.; and Denny, P. 2025. Evaluating language models for generating and judging programming feedback. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, 624–630.

Koutcheme, C.; Dainese, N.; Sarsa, S.; Leinonen, J.; Hellas, A.; and Denny, P. 2024. Benchmarking educational program repair. *arXiv preprint arXiv:2405.05347*.

Koutcheme, C.; Sarsa, S.; Leinonen, J.; Hellas, A.; and Denny, P. 2023. Automated program repair using generative models for code infilling. In *International Conference on Artificial Intelligence in Education*, 798–803. Springer.

Liu, F.; Liu, Z.; Zhao, Q.; Jiang, J.; Zhang, L.; Li, G.; Sun, Z.; Li, Z.; and Ma, Y. 2024. FastFixer: An Efficient and Effective Approach for Repairing Programming Assignments. arXiv:2410.21285.

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. In *NeurIPS 2023 workshop on instruction tuning and instruction following*.

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

Phung, T.; Cambronero, J.; Gulwani, S.; Kohn, T.; Majumdar, R.; Singla, A.; and Soares, G. 2023. Generating high-precision feedback for programming syntax errors using large language models. *arXiv preprint arXiv:2302.04662*.

Phung, T.; Pădurean, V.-A.; Singh, A.; Brooks, C.; Cambronero, J.; Gulwani, S.; Singla, A.; and Soares, G. 2024. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation. arXiv:2310.03780.

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*.

Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; Rapin, J.; Kozhevnikov, A.; Evtimov, I.; Bitton, J.; Bhatt, M.; Ferrer, C. C.; Grattafiori, A.; Xiong, W.; Défossez, A.; Copet, J.; Azhar, F.; Touvron, H.; Martin, L.; Usunier, N.; Scialom, T.; and Synnaeve, G. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

Sarsa, S.; Denny, P.; Hellas, A.; and Leinonen, J. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM conference on international computing education research-volume 1*, 27–43.

Van Praet, L.; Hoobergs, J.; and Schrijvers, T. 2024. ASSIST: Automated Feedback Generation for Syntax and Logical Errors in Programming Exercises. In *Proceedings of the*

2024 ACM SIGPLAN International Symposium on SPLASH-E, 66–76.

Wang, W.; Kwatra, A.; Skripchuk, J.; Gomes, N.; Milliken, A.; Martens, C.; Barnes, T.; and Price, T. 2021. Novices' learning barriers when using code examples in open-ended programming. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 394–400.

Yang, B.; Tian, H.; Pian, W.; Yu, H.; Wang, H.; Klein, J.; Bissyandé, T. F.; and Jin, S. 2024. CREF: An LLM-based Conversational Software Repair Framework for Programming Tutors. arXiv:2406.13972.

Yang, J.; Wan, J.; Yao, Y.; Chu, W.; Xu, Y.; and Qi, Y. 2025. inf-retriever-v1 (Revision 5f469d7).

Zhang, J.; Cambronero, J.; Gulwani, S.; Le, V.; Piskac, R.; Soares, G.; and Verbruggen, G. 2022. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*.

Zhang, J.; Cambronero, J. P.; Gulwani, S.; Le, V.; Piskac, R.; Soares, G.; and Verbruggen, G. 2024. Pydex: Repairing bugs in introductory python assignments using llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1): 1100–1124.

Zhang, Y.; Li, M.; Long, D.; Zhang, X.; Lin, H.; Yang, B.; Xie, P.; Yang, A.; Liu, D.; Lin, J.; Huang, F.; and Zhou, J. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. arXiv:2506.05176.

Zhao, Q.; Liu, F.; Zhang, L.; Liu, Y.; Yan, Z.; Chen, Z.; Zhou, Y.; Jiang, J.; and Li, G. 2024. Peer-aided Repairer: Empowering Large Language Models to Repair Advanced Student Assignments. *arXiv preprint arXiv:2404.01754*.

Zhou, Y.; Liu, E.; Neubig, G.; Tarr, M.; and Wehbe, L. 2024. Divergences between language models and human brains. *Advances in neural information processing systems*, 37: 137999–138031.