Accelerating Automatic Program Repair with Dual Retrieval-Augmented Fine-Tuning and Patch Generation on Large Language Models

Anonymous ACL submission

Abstract

Automated Program Repair (APR) is essential for ensuring software reliability and quality while enhancing efficiency and reducing developers' workload. Although rule-based and learning-based APR methods have demonstrated their effectiveness, their performance was constrained by the defect type of repair, the quality of training data, and the size of model parameters. Recently, Large Language Models (LLMs) combined with Retrieval-Augmented-Generation (RAG) have been increasingly adopted in APR tasks. However, current code LLMs and RAG designs neither fully address code repair tasks nor consider code-specific features. To overcome these limitations, we propose *SelRepair*, a novel APR approach with integration of a fine-tuned LLM with a newly-designed dual RAG module. This approach uses a bug-fix pair dataset for fine-tuning and incorporates semantic and syntactic/structural similarity information through an RAG selection gate. This design ensures relevant information is retrieved efficiently, thereby reducing token length and inference time. Evaluations on Java datasets show *SelRepair* outperforms other APR methods, achieving 26.29% and 17.64% in terms of exact match (EM) on different datasets while reducing inference time by at least 6.42% with controlled input lengths.

1 Introduction

Program Repair (PR) is the process of identifying and correcting errors (often called bugs) in a software program by using automated tools or manual techniques with an aim to improve the software's reliability and performance (Urli et al., 2018). PR is a time-consuming and labor-intensive task, e.g., fixing bugs taking up more than 1/3 of the software maintenance time (Lientz et al., 1978) and 90% of software maintenance cost (Britton et al., 2012). To improve the efficiency of PR, automatic program repair (APR) (Le Goues et al., 2021) has been proposed to reduce time consumption and human efforts. Some APR approaches, such as heuristic-based approaches (Weimer et al., 2009), template-based approaches (Meng et al., 2023), and semantics-driven approaches (Nguyen et al., 2013) are limited by the manual process (involving laborious heuristic rule design and template design) and the types of bugs fixed. Although deep-learning-based APR can address the limitations related to the bug types, its performance is mainly influenced by the model parameters and the quality of the training data (Wang et al., 2023b).

Considering the limitations of conventional APR approaches, LLMs have recently been proposed to complete APR tasks (Huang et al., 2023) owing to their stronger natural language understanding and even code understanding capability obtained by extensive training on vast amounts of corpus. Nowadays, there are two ways of adopting LLMs to complete APR tasks (Soylu et al., 2024): *prompt engineering* and *fine-tuning* (refer to Appendix A for more details). As for prompt engineering, since most popular generalized LLMs do not include APR-related pre-training tasks, it is difficult to design an ideal set of prompts to target generic APR tasks. Regarding fine-tuning approaches, most of them are adopted on LLMs with fewer than 1B parameters. For models with more than 1B parameters, the primary fine-tuning method is Parameter-Efficient Fine-Tuning (PEFT) fine-tuning, which *cannot fully unleash the potential of LLMs in APR*. In addition, for both prompt engineering and fine-tuning, the design of most prompts includes natural language contexts, such as issue/error descriptions and function requirements. Although this kind of prompt can provide additional details to understand codes, it also *increases the prompt complexity and limits the usage scenarios*. Specifically, natural language descriptions are redundant to some simple syntax errors or common PR tasks, thereby easily causing the prompt to exceed the length limit (Chen et al., 2024). Moreover, prompts with natural language descriptions cannot handle those scenarios, in which developers or students provide error codes without detailed error descriptions at the initial stages of program development.

Recently, Retrieval-Augmented Generation (RAG) has been adopted to improve the performance of LLMs. RAG generates accurate outputs by firstly retrieving relevant information (usually from an external specialized knowledge base) and then feeding this retrieved information into LLMs as contexts, thereby greatly enhancing the ability of LLMs (Appendix B depicts an example of how RAG contributes to APR). In the LLM-based APR task, RAG has been utilized to both prompt engineering (Nashid et al., 2023) and fine-tuning (Wang et al., 2023b) modules. However, existing RAG approaches only adopt code semantics similarity while overlooking other code features, such as code structure and syntax information. The utilization of key code features needs a fine-grained program analysis while few approaches leverage RAG based on diverse code features. More importantly, these RAG-based approaches lack the *validation of RAG selection* since they do not judge the necessity of RAGs for APR tasks. The lack of judging RAGs may result in redundant information being added to the input, thereby *increasing the model inference time* and textitdegrading performance.

In order to fill the above gaps, we propose *SelRepair*, a novel Selective RAG-based program Repair framework by full-parameter fine-tuning LLM. This framework considers *both semantics and syntax information* matching for retrieval based on buggy codes. Moreover, a newly-designed *RAG selection gate* is adopted to determine the necessity of RAGs by setting a threshold to determine whether the extracted bug-fix pair needs to be added to the context. The RAG selection gate can achieve efficient retrieval and controlled prompt length, thereby decreasing inference time. Further, we utilize existing APR datasets and design a *code-only* prompt to *full-parameter fine-tune* a large-parameter code LLM for APR tasks. As a result, the capabilities of LLMs have been fully exploited for APR tasks. The code-only prompt can be applied to diverse scenarios while controlling the prompt length. We evaluate our method by conducting extensive experiments on a public APR dataset of Java and an enterprise dataset. The experimental results demonstrate that integrating full-parameter fine-tuned LLMs with dual RAG greatly contributes to outstanding APR performance in terms of Exact Match (EM) and CodeBLEU.

The contributions of this paper are fourfold:

- We propose *SelRepair*, an APR framework that leverages similar bug-fix pairs as the context and fine-tuned LLM to achieve better PR performance than other SOTA approaches.
- In order to extract the unique features of codes, we construct a *dual RAG module* considering not only semantics similarity but also syntax as well as structure similarity to retrieve relevant context for APR. Both the retrievals contribute to the superior performance of *SelRepair*.
- To ensure the effective validation of RAG, we design a *RAG selection gate* to determine whether the extracted information is input into the LLM as a context. With the utilization of the RAG selection gate, we control the average input length, which decreases to 60.53, 133.16, and 992.25 in Java datasets (with two different code lengths) and a C/C++ dataset, respectively, while the inference time decreases by 6.42%, 13.77%, and 9.95%, respectively.
- We utilize a *code-only* prompt for APR tasks and adopt it to full-parameter fine-tune LLM, thereby fully exploiting the capabilities of LLMs and making the prompt concise to apply to diverse scenarios. Our approach outperforms other state-of-the-art approaches in a public APR dataset and an enterprise dataset. It can achieve 26.29%, 17.64%, and 25.46% of EM in Java datasets (with two different code lengths) and a C/C++ dataset, respectively. It can also generate 59 correct patches in the enterprise dataset.

2 Related Work

2.1 Automatic Program Repair

As mentioned in § 1, conventional APR approaches can be categorized into the following four types. **Heuristic-based approaches** adopt heuristic rules or genetic algorithms to generate patches such as *GenProg* (Le Goues et al., 2012), Marriagent (Kou et al., 2016), pyEDB (Assiri and Bieman, 2014). **Template-based approaches** use predefined fix templates to guide code modifications (Meng et al., 2023). Typical template-based approaches include *TBar* (Liu et al., 2019) and *PAR* (Kim et al., 2013). **Semantics-driven approaches** such as *SemFix* (Nguyen et al., 2013) use symbolic execution and test suites to extract semantic constraints, and then synthesize repairs satisfying the extracted constraints by program synthesis (Le et al., 2018). Considering the fix-type limitations of the above APR methods, **deep-learning-based approaches** have kept rapidly evolving by adopting neural machine translation techniques in natural language processing to generate repair patches (Tufano et al., 2019; Jiang et al., 2021; Gupta et al., 2017).

Since deep-learning-based approaches are limited by model parameters and training-data quality, **LLM-based approaches** have been proposed for APR (Zhang et al., 2023; Xia and Zhang, 2022). Specifically, prompt-engineering-based methods extract knowledge by static analysis tools and combine the knowledge with buggy codes to construct prompts (Pearce et al., 2023). Fine-tuning-based methods, such as *RAP-Gen* (Wang et al., 2023b) adopt code-only prompts to fine-tune LLM. Some other approaches use PEFT fine-tuning and RAG for APR (Silva et al., 2024) or APR assistance (Li et al., 2024). In our research, we adopt full-parameter fine-tuning on an LLM with larger parameter sizes and optimize RAG by using the selection gate and dual retrievals.

2.2 LLM for SE tasks

With the rapid development of LLMs, many LLMs have been proposed to be adopted in software engineering (SE) tasks (Zheng et al., 2023, 2024). On the one hand, some studies utilized prompt engineering on generalized LLMs (Minaee et al., 2024). such as code summarization (Sun et al., 2023) and vulnerability detection (Zhou et al., 2024). On the other hand, some specialized LLMs (i.e., code LLM) such as *CodeBERT* (Feng et al., 2020), *GraphCodeBERT* (Guo et al., 2021), and *CodeT5* (Wang et al., 2021) were proposed in the SE field. Since these LLMs mainly use programming languages as the pre-training corpus and SE tasks as the pre-training tasks (e.g., code completion and identifier prediction), these models perform better than generalized LLMs in some complex SE tasks (Chen et al., 2023). Therefore, some studies adopted these models for specific SE tasks, such as vulnerability detection (Wang et al., 2024b) or code search (Wang et al., 2023a).

Recently, some models with larger parameter sizes (more than 1 billion parameters), e.g., *CodeLLama* (Rozière et al., 2024) and *StarCoder 2* (Lozhkov et al., 2024) have been proposed, although the research related to full-parameter fine-tuning LLMs is still relatively limited. These LLMs typically include *BASE* version and *INSTURCT* version. The *INSTURCT* version is the fine-tuned model by using specific natural language instructions. Instead of using *INSTURCT*, we only adopt the *BASE* version that utilizes code information as a pre-training corpus. The goal of this paper is to utilize RAG and full-parameter fine-tuning on code LLMs for APR.

3 Methodology

Figure 1 depicts the workflow of the proposed *SelRepair*. We design a dual patch retriever considering both semantics and structure-dependency information. An RAG selection gate is also added in the dual patch retriever (in § 3.1). Then, we adopt the retriever to get relevant bug-fix pairs as context and combine them with buggy code as code-only prompts to fine-tune code LLMs for APR (in § 3.2). At last, we utilize the fine-tuned models to generate fixed code (in § 3.3).

3.1 Dual Patch Retriever

Codebase Construction. RAG can enhance the ability of LLM since it can provide expert knowledge that contributes to the task. We construct a codebase that includes APR-related knowledge. Specifically, the codebase consists of existing method-level buggy codes and their corresponding fix patches (i.e., bug-fix pairs). Our goal is to retrieve the relevant bug-fix pairs as the context. In contrast to most existing methods, where the codebase built on top of *repository-level* only retrieves *repository-level contexts* (Zhang et al., 2024; Xia et al., 2024), we construct a *generalized codebase* based on *across repositories*.

Hybrid Retriever. The dual patch retriever is an RAG module that aims to retrieve the most relevant bug-fix pairs as the context. Among those studies related to LLM for SE tasks, some RAG modules have been utilized to retrieve relevant information. They adopt similarity metrics, such as BM25 and code embedding (Wang et al., 2023b; Nashid et al., 2023) to get the most relevant code. BM25 considers the code token fre-

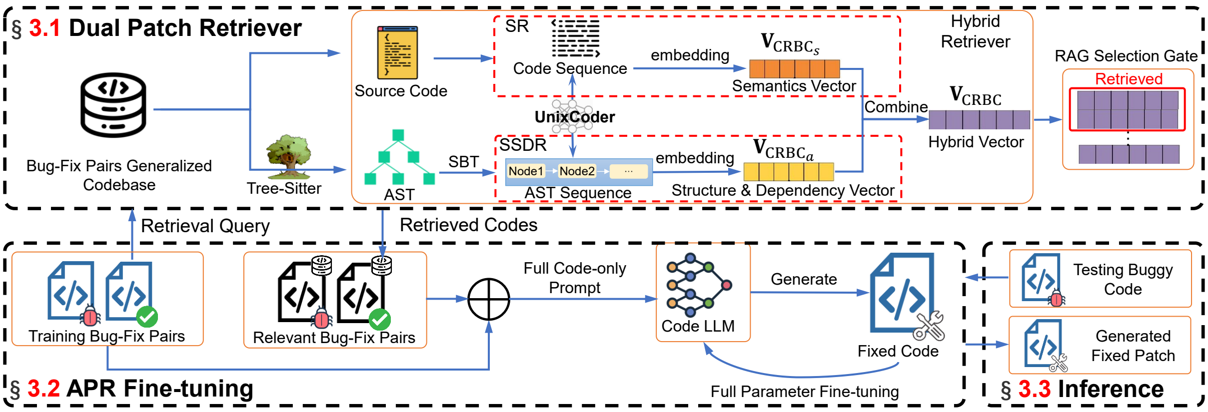

Figure 1: The Workflow of *SelRepair*

quency as the relevance metric while code embedding converts the code to vectors for similarity calculation. However, these two features only consider the source code information as a reference for relevant information, though the source code only contains superficial semantics without including other programming language features, such as syntax information, variable type information, control flow information, and so on. To tackle this issue, we introduce abstract syntax tree (AST), an abstract representation of the syntactic structure of source code to attain complete information for retrieval. AST is represented by a tree structure, in which each node has both *type* and *value* information. *Type* indicates the role of the node in the syntactic structure, such as `if_statement` and `formal_parameters`, etc., while *value* indicates specific code information, which is consistent with the source code information. By adopting ASTs, we can attain static structures. Based on the above information, we can also get the data dependency. Both source code and AST can construct complete program information, thus increasing the retrieval reliability. Considering both semantics and static structures of programs, we construct a hybrid retriever by combining a **semantics retriever (SR)** and a **static structure and dependency retriever (SSDR)**.

Algorithm 1 (in Appendix C) elaborates on the working procedure of the hybrid retriever. We aim to retrieve the most relevant bug-fix pairs from the codebase. Firstly, in line 3, we adopt `AST_Parse` function to get the AST of the buggy code to fix (i.e., target buggy code). In `AST_Parse`, we conduct an incremental grammar parsing library for parsing programming languages called Tree-Sitter (Latif et al., 2023) to generate the AST. In order to extract features of both code and AST, we utilize a code pre-trained model, *UnixCoder* (Guo et al., 2022) to convert code and AST into semantics vector and structure vector, respectively. We use *UnixCoder* mainly because it adopts both code and AST as the training corpus in pre-training tasks including Masked Language Modeling, Unidirectional Language Modeling, and Code Fragment Representation Learning. Thus, we do not need to fine-tune or retrain an embedding model of AST. Since *UnixCoder* takes a sequence as input, we traverse the source code to obtain a code sequence as *UnixCoder*'s input. It is necessary to traverse AST to a flattened sequence for model understanding (line 4) due to AST's tree structure. Referring to (Guo et al., 2022), we transform AST nodes to the sequence (refer to Appendix D for more details). After obtaining the sequences of both codes and ASTs, we input them to *UnixCoder* and get the source code vector $\mathbf{V}_{\mathrm{BCs}}$ and the AST vector $\mathbf{V}_{\mathrm{BCa}}$ of target buggy code (BC) (lines 5-6). Then, we calculate the average vector $\mathbf{V}_{\mathrm{BC}}$ to get the target buggy code's hybrid feature vector.

In the retrieval phase, we iterate each bug-fix pair in the codebase (line 8) and get the candidate relevant buggy code (CRBC) and candidate relevant fixed code (CRFC). Similarly, we adopt Tree-Sitter to parse the AST and `AST_traversal` to get the AST sequence of CRBC (lines 9-10). *UnixCoder* is also adopted to get the source code vector $\mathbf{V}_{\mathrm{CRBCs}}$ and the AST vector $\mathbf{V}_{\mathrm{CRBCa}}$ (lines 11-12). The hybrid feature vector of CRBC $\mathbf{V}_{\mathrm{CRBC}}$ is also attained (lines 13).

In the hybrid retriever, we use the cosine similarity of the hybrid feature vectors to measure the relevance between the target buggy code and the bug-fix pair in the codebase (line 14). The cosine similarity can be calculated as follows:

$$\kappa(\mathbf{V}_{\mathrm{CRBC}}, \mathbf{V}_{\mathrm{BC}}) = \frac{\mathbf{V}_{\mathrm{CRBC}} \cdot \mathbf{V}_{\mathrm{BC}}}{\|\mathbf{V}_{\mathrm{CRBC}}\| \times \|\mathbf{V}_{\mathrm{BC}}\|}, \quad (1)$$

where $\mathbf{V}_{\mathrm{CRBC}}$ represents the hybrid feature vector of CRBC in the bug-fix pair and $\mathbf{V}_{\mathrm{BC}}$ represents the hybrid feature vector of the BC to be fixed. The term $\mathbf{V}_{\mathrm{CRBC}} \cdot \mathbf{V}_{\mathrm{BC}}$ represents the dot product, which is calculated by $\mathbf{V}_{\mathrm{CRBC}} \cdot \mathbf{V}_{\mathrm{BC}} = \sum_{i=1}^{n} V_{\mathrm{CRBC}_i} V_{\mathrm{BC}_i}$, where V_{CRBC_i} and V_{BC_i} are the i^{th} elements of $\mathbf{V}_{\mathrm{CRBC}}$ and $\mathbf{V}_{\mathrm{BC}}$ vectors, respectively, with n vector dimension. Vectors $\mathbf{V}_{\mathrm{CRBC}}$ and $\mathbf{V}_{\mathrm{BC}}$ have norms denoted by $\|\mathbf{V}_{\mathrm{CRBC}}\| = \sqrt{\sum_{i=1}^{n} \mathbf{V}_{\mathrm{CRBC}_i}}$ and $\|\mathbf{V}_{\mathrm{BC}}\| = \sqrt{\sum_{i=1}^{n} \mathrm{V}_{\mathrm{BC}_i}}$. The greater $\kappa(\mathbf{V}_{\mathrm{CRBC}}, \mathbf{V}_{\mathrm{BC}})$, the more relevant the code to be fixed and the bug-fix pair in the codebase is. At last, we design an RAG selection gate to ensure that only retrieved bug-fix pairs fulfilling the requirements are used as relevant bug-fix pairs. The details are shown below.

RAG Selection Gate. As mentioned in § 3.1, we retrieve relevant bug-fix pairs based on semantics and AST similarity. The relevant information is used as a context for the model inputs to assist in repairing the target code. However, APR has a high requirement for efficiency and accuracy in real-world scenarios. If all retrieved bug-fixed pairs are added to the context, it may negatively affect the efficiency and accuracy of APR. On the one hand, it may cause the input sequence to be longer than the input length limit of the model, thereby incurring information loss due to truncation. On the other hand, if the extracted bug-fix pairs do not have a high enough degree of similarity with the target code, the added context will instead become a noisy input, degrading the accuracy. To address this challenge, we propose a selection gate mechanism. This process begins by using UniXcoder to encode and rank all retrieved information based on the similarity scores described previously. Given the token length limitations, we set a threshold for inclusion. Only bug-fix pairs with a similarity score exceeding this threshold are considered valid and added to the context. These selected pairs are then incorporated into the context in descending order of their similarity scores, until the token limit is reached. This approach ensures that the most relevant information is prioritized within the constrained context space. Therefore, the token length can be controlled while decreasing the inference time.

3.2 APR Fine-tuning

After selecting the relevant bug-fix pairs, we construct the input to fine-tune LLMs. The input consists of buggy codes from the training set and the retrieved valid bug-fix pairs. Inspired by (Wang et al., 2023b), we design a *code-only* prompt supplementing a [BUG] token and a [FIX] token to concatenate buggy code and valid bug-fix pairs. The concatenated input is shown as follows:

$$\text{[BUG] RBC}_1 \text{ [FIX] RFC}_1 \text{ [BUG] RBC}_2 \text{ [FIX]} \setminus$$
$$\text{RFC}_2 \ \ldots \ \text{RBC}_i \text{ [FIX] RFC}_i \ \ldots \text{[BUG] BC [FIX]},$$

where RBC_i represents the relevant buggy code in the i^{th} valid bug-fix pair, RFC_i represents the relevant fixed code in the i^{th} valid bug-fix pair, and BC represents the buggy code that needs to be repaired. The goal of the approach is to full-parameter fine-tune code LLMs to complete the fixed code in the above sequence. The objective function of fine-tuning is shown below:

$$P_\theta(Y_i|X_i) = \prod_{k=1}^{n} P_\theta(y_{i,k}|X_i, y_{i,1}, \ldots, y_{i,k-1}), \quad (2)$$

where θ is the parameter of the LLM, X_i is i^{th} the input sequence, Y_i is the sequence with correct completed fixed code, and $y_{i,k}$ represents the k^{th} token of the sequence with correct completed fixed code. The goal is to maximize the probability $P_\theta(Y_i|X_i)$ by optimizing the parameter θ.

3.3 Inference

In the inference phase, we utilize test datasets to evaluate the performance of fine-tuned LLMs in generating patches. Specifically, we take each test sample and retrieve the relevant bug-fix pair via RAG as a context. We construct the same prompt as fine-tuning. In other words, we input a code-only prompt into fine-tuned LLMs to evaluate the generated patches using evaluation metrics. There are two kinds of test datasets: 1) the public code datasets and 2) another dataset coming from code fixes made by developers in a software-development enterprise during the development process. To simulate a real APR scenario, we use a search algorithm called beam search (Freitag and Al-Onaizan, 2017) to generate multiple patches for each test sample in the real-world test dataset to generate patches. With these datasets, we evaluate the performance of *SelRepair* in both experimental and real scenarios.

Table 1: Evaluation Results for Compared Approaches

Approaches	Tufano Subset 1			Tufano Subset 2			VulRepair		
	EM (%)	BLEU-4	CodeBLEU	EM (%)	BLEU-4	CodeBLEU	EM (%)	BLEU-4	CodeBLEU
GPT-3.5	2.58	11.67	56.78	1.72	12.38	63.64	0.49	4.52	41.37
GPT-4o	0.17	6.24	57.46	0.00	7.19	59.68	0.00	2.14	30.95
DeepSeek-R1-Distill	0.09	3.24	37.52	0.00	2.47	34.68	0.00	0.83	18.84
RAP-Gen	24.80	69.77	76.33	15.84	85.27	85.92	23.02	48.20	51.67
SelRepairLlama	5.96	30.84	51.32	4.36	58.97	67.27	6.82	28.42	39.37
SelRepairT5	25.27	65.98	76.57	16.36	80.23	84.81	24.36	43.83	58.85
SelRepairLoRA	22.62	57.46	72.99	13.05	74.06	82.19	0.73	34.98	49.94
SelRepair	**26.29**	61.61	74.35	**17.64**	73.88	82.24	**25.46**	38.39	50.84

4 Experiments and Evaluation

This section presents experiments to evaluate *SelRepair*'s performance in APR tasks and analyze the influencing factors of its performance. We aim to answer the following four research questions.

RQ1: What is the proposed *SelRepair*'s performance compared with other state-of-the-art APR approaches?

RQ2: What are the effects of different modules on *SelRepair*'s APR performance?

RQ3: What are the effects of selection gate configuration on *SelRepair*'s APR performance?

RQ4: What is *SelRepair*'s performance in real-world scenarios?

4.1 Data Preparation & Experiment Configurations

Datasets. In order to evaluate *SelRepair*'s performance on program repair of different languages, we focus on Java and C/C++ program repair. Therefore, we conduct experiments on two Java datasets and a C/C++ dataset. We also introduce one additional dataset obtained from a software enterprise with an aim to evaluate the performance of *SelRepair* in real-world scenarios. Appendix E.1 gives more details.

Evaluation Metrics & Experiments Configuration. Three metrics are adopted to evaluate the APR performance: Exact Match (**EM**) (Zirak and Hemmati, 2024), 4-grams Bilingual Evaluation Understudy (Papineni et al., 2002) (**BLEU-4**) and **CodeBLEU** (Ren et al., 2020) (refer to Appendix E.2 for more details). The detailed hyperparameter settings are given in Appendix E.3.

4.2 RQ1: What is the proposed *SelRepair*'s performance compared with other state-of-the-art APR approaches?

We compare *SelRepair* with six state-of-art approaches, namely *GPT-3.5* (Koubaa, 2023), *GPT-4o* (Sun et al., 2024), *DeepSeek-R1-Distill* (DeepSeek-AI et al., 2025), *RAP-Gen* (Wang et al., 2023b), *SelRepair* with *CodeLlama*-based LLM (*SelRepairLlama*), *SelRepair* with *CodeT5*-based LLM (*SelRepairT5*) and *SelRepair* with LoRA fine-tuning (*SelRepairLoRA*). We describe the details of these approaches in Appendix E.4. We choose these approaches because comparative approaches combine RAG with full-parameter fine-tuning and adopt code-only prompts similar to *SelRepair*. Moreover, we only consider an approach based on PEFT (i.e., LoRA). While we focus on approaches using RAG-based fine-tuning comparative to *SelRepair*, we also include *GPT-3.5*, GPT-4o, and *DeepSeek-R1-Distill* in our comparison. These choices provide a baseline performance of a widely-used and general-purpose language model, thereby demonstrating the potential advantages of our approach in the APR context. The adoption of GPT-3.5 and GPT-4o also helps verify whether advanced generalized LLMs necessarily yield better APR performance. We leverage each approach to generate 1 repair candidate for each sample in the testing set.

The comparison results on Tufano's dataset are shown in Table 1. The EM results of *RAP-Gen* are obtained from the original paper while other metrics are experimentally evaluated by us. It can be found that *SelRepair* achieves new SoTA performance of 26.29 EM and 17.64 EM in Tufano Subset 1 (< 50 tokens) and Tufano Subset 2 (50-100 tokens), respectively, outperforming other SoTA LLMs. Specifically, it outperforms *GPT-3.5*, *GPT-4o*, *DeepSeek-R1-Distill*, *RAP-Gen*, *SelRepairLlama*, *SelRepairT5*, and *SelRepairLoRA* by 918.99%, 15364.71%, 29111.11%, 6.01%, 341.11%, 4.04%, and 16.22% respectively, in Tufano Subset 1. In Tufano Subset 2, *SelRepair* performs 925.58%, 11.36%, 304.59%, 7.82% and 35.17% better than *GPT-3.5*, *RAP-Gen*, *SelRepairLlama*, *SelRepairT5* and *SelRepairLoRA*, respectively. In summary, when inputting the code-only prompt, *SelRepair* outperforms existing

Table 2: Ablation Study

Module Construction	Tufano Subset 1			Tufano Subset 2			VulRepair		
	EM (%)	BLEU-4	CodeBLEU	EM (%)	BLEU-4	CodeBLEU	EM (%)	BLEU-4	CodeBLEU
w/o RAG & Ft	0.00	7.96	36.96	0.00	3.32	31.71	0.00	8.05	28.26
w/o Ft	0.00	2.88	40.67	0.00	6.26	42.35	0.00	9.96	31.17
w/o RAG	15.02	36.07	55.56	6.52	50.72	60.91	19.24	37.58	50.46
w/o SR	25.57	58.16	73.94	10.83	60.40	71.18	21.92	36.79	50.06
w/o SSDR	22.28	56.81	72.98	17.41	73.73	82.12	22.68	38.28	50.64
SelRepair	**26.29**	61.61	74.35	**17.64**	73.88	82.24	**25.46**	38.39	50.84

SoTA LLMs in terms of the percentage of correct repairs in both datasets with diverse code lengths. Moreover, we also observe that *SelRepairT5* outperforms 1.90% than *RAP-Gen* in Tufano Subset 1 and 5.28% than *RAP-Gen* in Tufano Subset 2. Since *SelRepairT5* and *RAP-Gen* utilize the same base code LLM, the results indicate that *the superiority of our design does not depend entirely on the scale of model parameters*. Other modules, including RAG and fine-tuning contribute to performance improvement. The exact contributions of RAG and fine-tuning are investigated in § 4.3. Moreover, *SelRepairLlama* performs poorly (5.96 in Tufano Subset 1 and 4.36 in Tufano Subset 2), indicating that CodeLlama does not work well for code-only prompts. The mediocre performance in *SelRepairLoRA* indicates that full parameter fine-tuning contributes more than PEFT.

Besides Java datasets (Tufano), Table 1 also reports the comparison results on VulRepair (i.e., C/C++ dataset). Notably, we reproduce *RAP-Gen* as well as other approaches on this dataset (*RAP-Gen* did not adopt this dataset). It can be found that *SelRepair* still achieves SOTA EM performance of 25.46, indicating its superior performance on different programming languages.

We also observe that *RAP-Gen* has a lower EM score than *SelRepair* despite its slightly higher BLEU-4 and CodeBLEU scores. This implies that *RAP-Gen* can generate results *semantically more similar* to ground truth than our *SelRepair*, but the correctness of the bug fixes generated by RAP-Gen may be less reliable than our *SelRepair*, as indicated by its lower EM score. Moreover, we do not consider deep-learning-based approaches since *RAP-Gen* is superior to most deep-learning-based approaches (Wang et al., 2023b). Further, *GPT-based* models and *DeepSeek-R1-Distill* have the worst performance, which may be attributed to the prompt design. As shown in Figure 5 in Appendix E.4, we do not provide any bug type information or fine-grained bug location information (e.g., buggy line) for a fair comparison. Therefore, this kind of prompt cannot contribute to a good performance for general-purpose LLMs like *GPT* and *DeepSeek-R1-Distill*. We also find that *GPT-4o* performs inferior to *GPT-3.5*. This may be because *GPT-4o* excels in general NLP tasks with complex data but performs poorly on some simple yet specific tasks.

4.3 RQ2: What are the effects of different modules on APR performance?

As mentioned in § 3, we adopt SR and SSDR as RAG and fine-tuning, respectively, to improve the APR performance. We design an ablation study to analyze how these modules contribute to the APR performance. We use "without (w/o) RAG and Fine-tuning", "without (w/o) Fine-tuning", "without (w/o) RAG", "without (w/o) SR", and "without (w/o) SSDR" as the module construction types and compare their performance with our baseline model. The results are shown in Table 2. It can be found that both fine-tuning and RAG (SR and SSDR) contribute to APR with different code lengths. *SelRepair* outperforms "w/o RAG", "w/o SR", and "w/o SSDR" by 75.03%, 2.82%, and 18.00%, respectively in Tufano Subset 1 (< 50 tokens). *SelRepair* performs 170.55%, 62.88%, and 1.32% better than "w/o RAG", "w/o SR", and "w/o SSDR" in terms of EM in Tufano Subset 2 (50-100 tokens). In Tufano Subset 1, SSDR has a more significant contribution because the static structure and dependency information of AST is more useful in short code to complement the semantic and syntax information, thereby helping the model to better understand the syntax and semantics of the code. Further, *SelRepair* outperforms "w/o RAG", "w/o SR", and "w/o SSDR" by 32.33%, 16.15%, and 12.26%, respectively in VulRepair. The results indicate that SR, SSDR and RAG all contribute to APR performance in different programming languages. *SelRepair* also has the best BLEU-4 and CodeBLEU scores among all the methods. Notably, "w/o RAG and Fine-tuning" and "w/o Fine-tuning" achieve an EM score of 0.00 in different datasets, suggesting that Code LLMs struggle to interpret code-only prompts without fine-tuning.

Table 3: RAG Selection Gate Setting & Efficiency Improvement

Threshold Setting	Tufano Subset 1					Tufano Subset 2					VulRepair				
	EM (%)	BLEU-4	CodeBLEU	Avg. Input Token Length	Infer. Time (%)	EM (%)	BLEU-4	CodeBLEU	Avg. Input Token Length	Infer. Time (%)	EM (%)	BLEU-4	CodeBLEU	Avg. Input Token Length	Infer. Time (%)
No Threshold	21.83	55.91	71.92	604.10	-	15.95	73.16	81.75	886.43	-	21.32	36.78	50.27	1175.09	-
0.5	23.47	60.84	73.63	571.07	0.53	12.89	71.84	80.46	867.42	7.79	21.68	36.70	50.39	1162.36	4.49
0.7	23.73	57.67	73.67	68.24	2.27	15.89	73.04	81.42	169.22	8.94	23.02	38.00	50.13	1033.31	9.13
0.8	24.43	57.88	73.77	61.36	4.59	**17.64**	73.88	82.24	133.16	13.77	**25.46**	38.39	50.84	992.25	9.95
0.9	**26.29**	61.61	74.35	60.53	6.42	14.72	72.74	81.31	131.05	29.68	23.75	38.90	51.93	986.70	15.96

4.4 RQ3: What are the effects of selection gate configuration on APR performance?

To find the optimal setting for the RAG selection gate, we design an experiment to analyze the effect of different selection gate threshold settings (0.9, 0.8, 0.7, 0.5, and No Threshold). Table 3 reports the results, showing that *SelRepair* has the best performance in Tufano Subset 1 (< 50 tokens) when the threshold is 0.9. In Tufano Subset 2 (50-100 tokens) and VulRepair, *SelRepair* has the best performance when the threshold setting is 0.8. The results indicate that too much RAG information may not enhance *SelRepair*'s performance.

We also analyze the efficiency of different threshold settings. We observe from Table 3 that both the input token length and the inference time decrease with the increased threshold. When the threshold value is 0.9, the inference time is 6.42%, 29.68%, and 15.96% less than no threshold setting in Tufano Subset 1 (< 50 tokens), Tufano Subset 2 (50-100 tokens) and VulRepair, respectively. Therefore, a suitable threshold setting not only improves the performance but also keeps the inference time within an acceptable range. This finding also has implications for the design of other RAG-based tasks. In other words, the RAG selection gate can also be adapted to other RAG-based LLM tasks. The acceleration of inference enhances its reliability in industrial practice.

Considering the trade-off between inference time and APR performance, we set 0.9 as the default threshold for Tufano Subset 1 and 0.8 for Tufano Subset 2 and VulRepair.

4.5 RQ4: What is *SelRepair*'s performance in real-world scenarios?

We also adopt a benchmark of 200 bug-fix pairs from an enterprise to verify the performance of *SelRepair* in real-world scenarios. This enterprise benchmark differs from open-source benchmarks like the Tufano dataset. While the Tufano dataset primarily addresses functional defects, the enterprise benchmark emphasizes coding bad practices and style issues, such as improper logging, unnecessary checks, and unused variables. The enterprise benchmark is designed to align with organizational coding standards. We intend to open-source this benchmark in the future, potentially providing a new resource for APR research. We count the number of correct patches for *SelRepair*, *SelRepair* w/o SR, *SelRepair* w/o SSDR, and *RAP-Gen* (fine-tuned with Tufano). The results are shown in Figure 2. As for beam search, we set the beam size to 10, indicating that each sample can have 10 generated patches. It can be found that *SelRepair* also achieves the best performance among all the approaches by generating 59 correct patches. When we adopt either SSDR or SR, *SelRepair* can still generate 42 and 55 correct patches. In contrast, *RAP-Gen* fine-tuned with the Tufano dataset can only generate 1 correct patch for our benchmark. The results demonstrate the excellent *generalizability* of *SelRepair*.

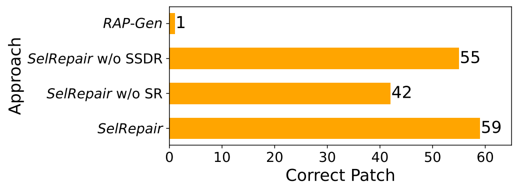

Figure 2: Performance on Real-world Enterprise Data

Other discussions. In Appendix F, we present a discussion of how *SelRepair* works, give a generated case study, and analyze *SelRepair*'s performance on *Defects4J*. In Appendix G, we present the threats to validity of *SelRepair*.

5 Conclusion

In this paper, we present *SelRepair*, an innovative APR approach leveraging fine-tuned LLMs with a dual RAG strategy. By fully fine-tuning LLMs using bug-fix pair datasets, we tailor the model to effectively address APR challenges. Our dual RAG module incorporates semantic, syntactic, and structural information, overcoming the limitations of current RAG mechanisms that focus solely on semantics. Additionally, we design an RAG selection gate to verify its role in the repair process. Our evaluation on three open datasets and one enterprise dataset shows *SelRepair* outperforms state-of-the-art methods, enhancing APR effectiveness and efficiency. Future work includes extending to other programming languages and integrating real-time feedback mechanisms to improve accuracy across diverse environments.

Limitations

Our current method is limited by datasets focused on individual methods, which simplifies research but misses real-world complexity. Many bugs result from interactions across methods or modules, requiring comprehensive analysis. Future work should expand techniques to handle larger code spans while preserving semantic integrity and control flows for holistic debugging.

The large parameter size of current state-of-the-art LLMs poses additional challenges by requiring significant resources for fine-tuning and limiting accessibility. Integrating these models into DevOps environments with quick response needs is difficult. While compression methods like distillation and quantization offer solutions, they can affect performance. Balancing expressiveness with deployment requirements remains challenging.

To overcome these limitations, we are exploring several research directions. We are developing techniques for cross-method and cross-component bug handling by integrating structural information. We're exploring efficient fine-tuning methods like meta-learning to optimize resource use. For deployment, we're modularizing models and enhancing caching and indexing to reduce latency, aiming to improve LLM-based debugging in real-world software development.

References

Fatmah Yousef Assiri and James M. Bieman. 2014. An assessment of the quality of automated program operator repair. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 273–282.

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, page 30–39, New York, NY, USA. Association for Computing Machinery.

Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2012. Quantify the time and cost saved using reversible debuggers. *Cambridge Judge Business School, Tech. Rep.*

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 654–668, New York, NY, USA. Association for Computing Machinery.

Yuxiao Chen, Jingzheng Wu, Xiang Ling, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. 2024. When large language models confront repository-level automatic program repair: How well they done? In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 459–471, New York, NY, USA. Association for Computing Machinery.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng

Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA. Association for Computing Machinery.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, Vancouver. Association for Computational Linguistics.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 935–947, New York, NY, USA. Association for Computing Machinery.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

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).

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Ryotaro Kou, Yoshiki Higo, and Shinji Kusumoto. 2016. A capable crossover technique on automatic program repair. In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 45–50.

Anis Koubaa. 2023. Gpt-4 vs. gpt-3.5: A concise showdown. *Preprints*.

Afshan Latif, Farooque Azam, Muhammad Waseem Anwar, and Amina Zafar. 2023. Comparison of leading language parsers – antlr, javacc, sablecc, tree-sitter, yacc, bison. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*, pages 7–13.

Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 163, New York, NY, USA. Association for Computing Machinery.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic program repair. *IEEE Software*, 38(4):22–27.

Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2024. Hybrid automated program repair by combining large language models and program analysis. *Preprint*, arXiv:2406.00992.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*. Reproducibility Certification.

B. P. Lientz, E. B. Swanson, and G. E. Tompkins. 1978. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471.

Xinyu Lin, Wenjie Wang, Yongqi Li, Shuo Yang, Fuli Feng, Yinwei Wei, and Tat-Seng Chua. 2024. Data-efficient fine-tuning for llm-based recommendation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 365–374, New York, NY, USA. Association for Computing Machinery.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 31–42, New York, NY, USA. Association for Computing Machinery.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. 2024. Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62st Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509.

Igor Melnyk, Vijil Chenthamarakshan, Pin-Yu Chen, Payel Das, Amit Dhurandhar, Inkit Padhi, and Devleena Das. 2023. Reprogramming pretrained language models for antibody sequence infilling. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 24398–24419. PMLR.

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1456–1468.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *Preprint*, arXiv:2402.06196.

Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th*

International Conference on Software Engineering (ICSE), pages 772–781.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA. Association for Computational Linguistics.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

André Silva, Sen Fang, and Martin Monperrus. 2024. Repairllama: Efficient representations and fine-tuned adapters for program repair. *Preprint*, arXiv:2312.15698.

Dilara Soylu, Christopher Potts, and Omar Khattab. 2024. Fine-tuning and prompt optimization: Two great steps that work better together. *Preprint*, arXiv:2407.10930.

Tao Sun, Yang Yang, Xianfu Cheng, Jian Yang, Yintong Huo, Zhuoren Ye, Rubing Yang, Xiangyuan Guan, Wei Zhang, Hangyuan Ji, Changyu Ren, Mengdi Zhang, Xunliang Cai, and Zhoujun Li. 2024. Repofixeval: A repository-level program repair benchmark from issue discovering to bug fixing.

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic code summarization via chatgpt: How far are we? *Preprint*, arXiv:2305.12865.

Qwen Team. 2025. Qwen2.5-vl.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4).

Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to design a program repair bot?: insights from the repairnator project. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 95–104, New York, NY, USA. Association for Computing Machinery.

Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2024a. Teaching code llms to use autocompletion tools in repository-level code generation. *Preprint*, arXiv:2401.06391.

Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023a. One adapter for all programming languages? adapter tuning for code search and summarization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 5–16.

Rongcun Wang, Senlei Xu, Yuan Tian, Xingyu Ji, Xiaobing Sun, and Shujuang Jiang. 2024b. Scl-cvd: Supervised contrastive learning for code vulnerability detection via graphcodebert. *Computers & Security*, 145:103994.

Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. 2023b. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 146–158, New York, NY, USA. Association for Computing Machinery.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374.

Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *Preprint*, arXiv:2302.11382.

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

Chunqiu Steven Xia and Lingming Zhang. 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*, ESEC/FSE 2022, page 959–971, New York, NY, USA. Association for Computing Machinery.

Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2024. Automated c/c++ program repair for high-level synthesis via large language models. *Preprint*, arXiv:2407.03889.

Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F. Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-objective fine-tuning for enhanced program repair with llms. *Preprint*, arXiv:2404.12636.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023a. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, Jie Zhou, Siming Chen, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023b. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *Preprint*, arXiv:2303.10420.

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.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA. Association for Computing Machinery.

Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. *Preprint*, arXiv:2308.11396.

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2024. A survey of large language models for code: Evolution, benchmarking, and future trends. *Preprint*, arXiv:2311.10372.

Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 47–51, New York, NY, USA. Association for Computing Machinery.

Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Trans. Softw. Eng. Methodol.*, 33(3).

A Taxonomy of Adopting LLMs

A.1 Prompt engineering

Prompt engineering refers to the design and optimization of input prompts to obtain the best output from an LLM (White et al., 2023). It does not require extra training but its performance depends on pre-training tasks. Since most popular generalized LLMs (e.g., GPT-3.5, GPT-4 etc. (Ye et al., 2023b)) and code LLMs (e.g., CodeT5 (Wang et al., 2021), CodeBERT (Feng et al., 2020), StarCoder (Li et al., 2023), StarCoder 2 (Lozhkov et al., 2024) etc.) do not include APR-related pre-training tasks, it is difficult to design an ideal set of prompts to target generic APR tasks.

A.2 Fine-tuning

Fine-tuning is the use of task-specific data (e.g., APR data) to further train a model based on an LLM (Lin et al., 2024). Despite many fine-tuned code pre-trained models for APR-related tasks, most of them are adopted on LLMs with less than 1B parameters (Mashhadi and Hemmati, 2021; Huang et al., 2023). Although other approaches fine-tune LLMs with more than 1B parameters (Yang et al., 2024), they adopt Parameter-Efficient Fine-Tuning (PEFT) techniques (Melnyk et al., 2023) (e.g., LoRA (Hu et al., 2022), adaptor tuning (Houlsby et al., 2019; Silva et al., 2024)) rather than full-parameter fine-tuning (Lv et al., 2024). As a result, they cannot fully unleash the potential of LLMs in APR.

B A Motivation Example of RAG

Figure 3 shows an example to describe how RAG contributes to APR. In the target buggy code, it contains a null pointer exception (NPE) bug since an attempt is made in a `for` loop to access an element in an array of strings that may not have been initialized (i.e, `greetings[1]`). By using the RAG, the APR module can retrieve an NPE-related bug-fix pair example. In this bug-fix pair, the key to the fix is to check if an array element is `null` before attempting to access it. The LLM then uses this bug-fix pair as a context to generate fixed code for the original bug, i.e., adding a `null` check.

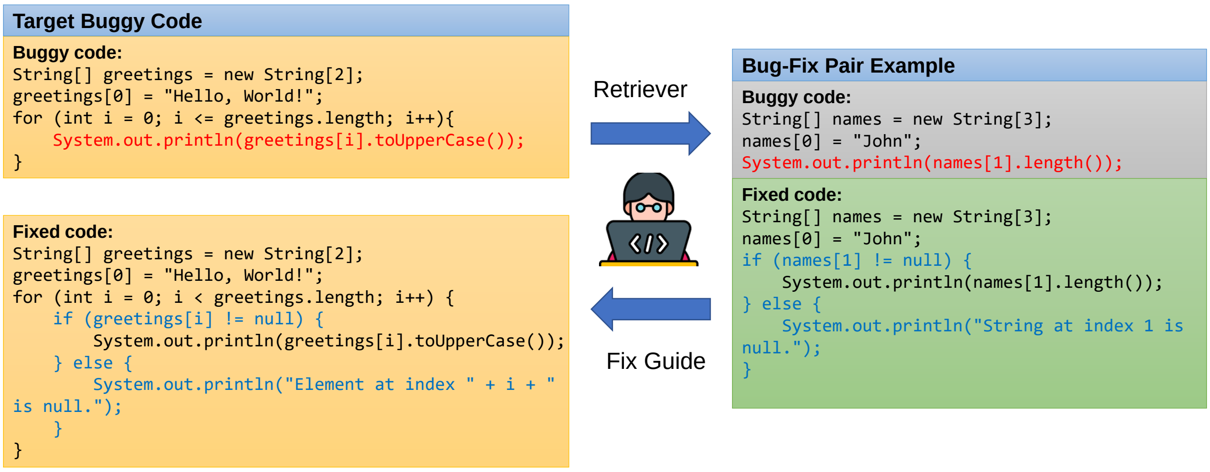

Figure 3: An Example of RAG in APR

C Hybrid Retriever Algorithm

Algorithm 1 depicts how the hybrid reviewer algorithm works.

Algorithm 1: Hybrid Retriever

```
Input: C: Bug-fix pairs; T: Target buggy code;
       t: Similarity threshold.
Output: BF: Retrieved bug-fix pair set
function hybrid_retriever(C, T, t)
BF ← [∅]
AST_T = AST_Parse(T)
ASTSeq_T = AST_traversal(AST_T)
V_BCs = UnixCoder(T)
V_BCa = UnixCoder(ASTSeq_T)
V_BC = (V_BCs + V_BCa)/2
for CRBC, CRFC in C do
    AST_BF = AST_Parse(CRBC)
    ASTSeq_BF = AST_traversal(AST_BF)
    V_CRBCs = UnixCoder(CRBC)
    V_CRBCa = UnixCoder(ASTSeq_BF)
    V_CRBC = (V_CRBCs + V_CRBCa)/2
    if κ(V_CRBC, V_BC) > t then
        BF.append((CRBC, CRFC))
    end
end
return BF
```

D Details of AST traversal

Algorithm 2 depicts the procedure of AST traversal, in which we add nodes to the sequence in the order of pre-order traversal. If the node is a leaf node, the node *value* information is added to the sequence directly (lines 3-4). If the node is a non-leaf node, it will transform to `AST#node_type#left` and `AST#node_type#right` tokens, while the information of its corresponding child nodes is added between these two tokens (lines 5-10). Figure 4 shows a toy example of AST and the corresponding AST sequence.

Algorithm 2: AST Traversal

```
   Input: R: The root node of the AST;
   Output: S: The traversed AST sequence;
1  function AST_traversal(R)
2  S ← [∅]
3  if R is leaf_node then
4      S.append(R.value)
5  else
6      S.append('AST#'+R.type+'#Left')
7      for node in R.children do
8          S.extend(AST_traversal(node))
9      end
10     S.append('AST#'+R.type+'#Right')
11 end
12 return S
```

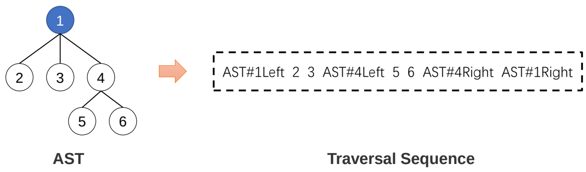

Figure 4: A Toy Example of AST Traversal

E Details of Experiment Setup

E.1 Details of Dataset Construction

We consider two Java datasets, a C/C++ dataset and a software enterprise's Java dataset to evaluate the performance of *SelRepair*.

We firstly evaluate *SelRepair* on a public dataset proposed by Tufano et al. (Tufano et al., 2019). It consists of bug-fix pairs at the method level and it is collected from fix commit records from GitHub. Specifically, it contains two data subsets of split according to the length of the code token. One is the subset with code lengths of less than 50 tokens (i.e., < 50 tokens dataset), and the other is the subset with code lengths of 50-100 tokens (i.e., 50-100 tokens dataset). These two subsets are named Tufano Subset 1 and Tufano Subset 2. The distribution of these two subsets is shown in Table 4. As for each subset, we random sample 1,000 samples as an RAG codebase. For the remaining samples, we split 80% of the dataset as a training set, 10% as a validation set, and 10% as a test set.

Another dataset is a C/C++ dataset proposed by Fu et al. (Fu et al., 2022), which is called VulRepair. It consists of bug-fix pairs combined by CVE-Fixes (Bhandari et al., 2021) and Big-Vul (Fan et al., 2020). We filtered out invalid samples, such as samples that were null. The distribution of this datsaet is also shown in Table 4. Similarly, we randomly sample 2000 samples as an RAG codebase. For the remaining samples, we split the dataset into training set, validation set, and a test set in a ratio of 8:1:1.

In order to evaluate the performance of *SelRepair* in real scenarios, we also introduce one additional dataset, which comes from a software enterprise. This dataset consists of 200 semantic bug-fix pairs caused by enterprise developers in real development scenarios. We verify the effectiveness of *SelRepair* to fix errors in realistic scenarios by using this dataset.

E.2 Evaluation Metrics

We adopt EM, BLEU-4, and CodeBLEU to evaluate the APR performance.

- **EM** refers to the ratio of generated fixes identical to the ground truth made by developers (i.e., reference fixes). Although there may be multiple fixes for the same bug, it can be used as an indicator of the performance of fixing logic bugs.

- **BLEU-4** is a commonly used machine translation evaluation metric that measures the similarity between the predicted text and the reference text. We utilize BLEU-4 as a looser metric to evaluate the similarity between generated fixes and reference fixes. It first splits the generated fix and the reference fix into 1-gram to 4-grams. Then, for each n-gram (1 to 4), BLEU-4 calculates the number of overlaps between the n-gram in the generated fix and the n-gram in the reference fix, as well as a weighted geometric mean of the 1-gram to 4-grams precision. The specific calculation process of BLEU-4 is given as follows:

$$\mathrm{BLEU4} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{4} \omega_n \log p_n\right), \tag{3}$$

where ω_n is the weight of n-grams, p_n is the precision of n-gram, and BP refers to the brevity penalty factor for the generated fix length. BP is given as follows:

$$\mathrm{BP} = \begin{cases} 1 \quad , & f_g > f_r, \\ \exp\left(1 - \dfrac{f_r}{f_g}\right), & f_g \leq f_r, \end{cases} \tag{4}$$

where f_g is the length of generated fix and f_r represents the reference fix.

- **CodeBLEU** is a code-specific evaluation metric derived from BLEU. It enables the quality assessment of APR tasks while preserving BLEU's benefits through n-gram matching and injecting code syntax and semantics through ASTs and data flows. CodeBLEU is calculated as follows:

$$\begin{aligned} \mathrm{CodeBLEU} = {} & \alpha \cdot \mathrm{BLEU} + \beta \cdot \mathrm{BLEU}_{\text{weight}} + \\ & \gamma \cdot \mathrm{Match}_{\text{ast}} + \epsilon \cdot \mathrm{Match}_{\text{df}}, \end{aligned} \tag{5}$$

where BLEU is the standard BLEU calculated by Eq. (3) (ω_1 to ω_4 are all equivalent). $\mathrm{BLEU}_{\text{weight}}$ refers to the weighted n-gram match calculated by Eq. (3) (ω_1 to ω_4 can be different). $\mathrm{Match}_{\text{ast}}$ refers to syntactic AST match, addressing the syntactic information of code. $\mathrm{Match}_{\text{ast}}$ is calculated as follows:

$$\mathrm{Match}_{\text{ast}} = \frac{\mathrm{Count}_{\text{clip}}(\mathrm{ST}_{\text{gen}})}{\mathrm{Count}(\mathrm{ST}_{\text{ref}})}, \tag{6}$$

where $\mathrm{Count}(\mathrm{ST}_{\text{ref}})$ refers to the total number of the subtrees of ASTs parsed from reference fixes, and $\mathrm{Count}_{\text{clip}}(ST_{\text{gen}})$ is the number of the subtrees of ASTs parsed from generated fixes that are matched the reference. $\mathrm{Match}_{\text{df}}$ refers to the semantic data-flow match score, which is calculated as follows:

$$\mathrm{Match}_{\text{df}} = \frac{\mathrm{Count}_{\text{clip}}(\mathrm{DF}_{\text{gen}})}{\mathrm{Count}(\mathrm{DF}_{\text{ref}})}, \tag{7}$$

where $\mathrm{Count}(\mathrm{DF}_{\text{ref}})$ is the total number of the reference fixes' data flows, and $\mathrm{Count}_{\text{clip}}(\mathrm{DF}_{\text{gen}})$ is the number of matched data-flows from generated fixes. α, β, γ and ϵ are weight coefficients designed by the user.

Table 4: Distribution of Dataset

Datasets	Language	Code Length	RAG Codebase	Train	Valid	Test
Tufano Subset 1	Java	$<$ 50 tokens	1,000	45,880	5,735	5,735
Tufano Subset 2	Java	50-100 tokens	1,000	51,565	6,447	6,447
VulRepair	C/C++	-	200	6,574	822	821

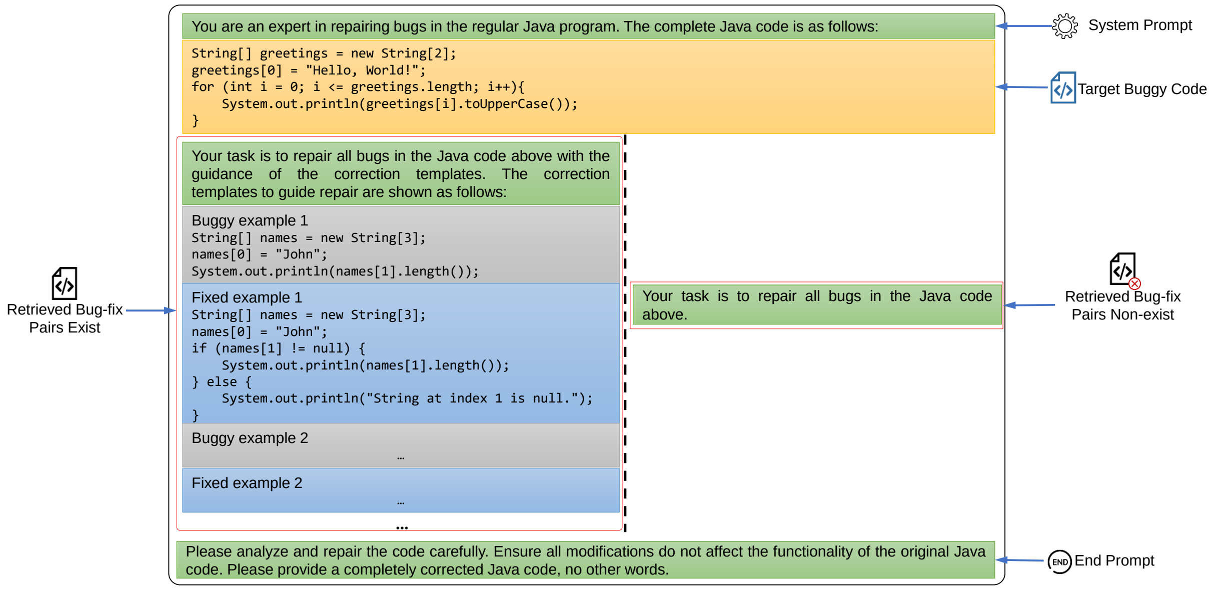

Figure 5: GPT-3.5 & GPT-4o Prompt Template

E.3 Experiment Configuration

The hyperparameter setting is shown as follows. Referring to (Wang et al., 2024a), we set the fine-tuning epochs as 3 for the large parameter ($>$ 1B) LLM. We set the context window as 512 tokens for the Tufano Subset 1 ($<$ 50 tokens), 1,024 tokens for the Tufano Subset 2 (50-100 tokens) and 1,500 tokens for VulRepair dataset. We adopt *StarCoder2-7B* as the foundation code LLM for fine-tuning. As for the optimizer, we utilize Adam (Kingma and Ba, 2015) with the learning rate 5×10^{-5} for supervised fine-tuning (SFT). The threshold of RAG selection gate is set as 0.9 for Tufano Subset 1 and 0.8 for Tufano Subset 2 and VulRepair dataset. More details are shown in § 4.4. All experiments are conducted on a server configured with 4 GPUs of NVIDIA GeForce RTX 3090.

E.4 Baselines

We adopt four state-of-the-art approaches as the baselines to compare with *SelRepair*, which are shown as follows:

- ***GPT-3.5***: *GPT-3.5* is a General-purpose Large Language Model developed by OpenAI that offer significant architectural and performance improvements compared with previous LLMs. It is based on the Transformer architecture. Since *GPT-3.5* is a General-purpose Large Model, referring to (Xu et al., 2024), we design an instruction-based prompt to implement the APR task, as shown in Figure 5. It includes system prompt and target buggy code. If retrieved bug-fix pairs exist, we add them to the prompt. Otherwise, we directly tell the model to perform the APR task via instructions. Finally, we add an end prompt to ask the model to generate the fixed code. For a fair comparison, the prompt does not contain a description of the bug type and bug location information. The utilization of *GPT-3.5* aims to measure whether *SelRepair* outperforms APR methods by adopting instruction-based prompt engineering.

- ***GPT-4o***: GPT-4o is one of the latest LLMs developed by OpenAI, and it has been significantly improved and enhanced in several aspects compared to GPT-3.5, including larger parameter sizes, and more training data, as well as support

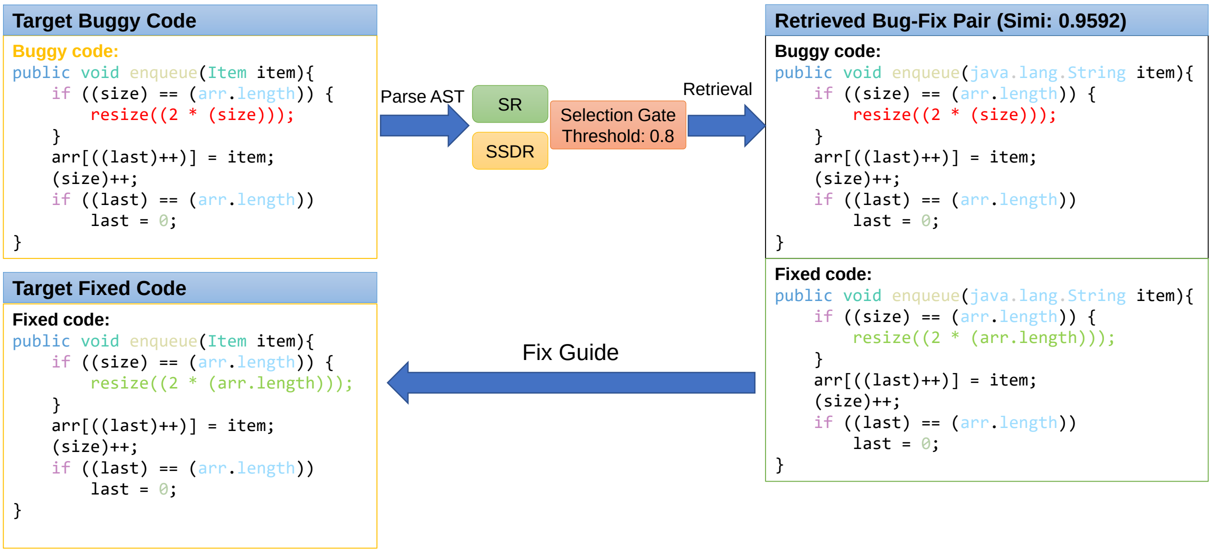

Figure 6: Detailed Process of *SelRepair*

for multimodal inputs and outputs. We design the same instruction-based prompt as GPT-3.5 to implement the APR task.

- ***DeepSeek-R1-Distill***: *DeepSeek-R1* is a general-purpose inference model developed by DeepSeek AI company. *DeepSeek-R1* uses reinforcement learning for post-training and is designed to improve inference, and is particularly adept at complex tasks such as mathematical, coding, and natural language reasoning. *DeepSeek-R1-Distill* models are fine-tuned based on open-source models, using samples generated by *DeepSeek-R1*. For a fair comparision, we adopt a 7B-parameter *DeepSeek-R1-Distill* model, that is, *DeepSeek-R1-Distill-Qwen-7B*. It is fine-tuned based on *Qwen2.5* (Team, 2025) LLM.

- ***RAP-Gen***: This approach adopt fine-tuning on *CodeT5* and semantics similarity as RAG. As mentioned in (Wang et al., 2023b), it outperforms most popular deep-learning-based APR approaches and code-LLM-based approaches. So, we adopt *RAP-Gen* as one of SoTA LLMs.

- ***SelRepairLlama***: In Appendix E.3, we adopt *StarCoder2-7B* as the foundation code LLM. We also consider another earlier code LLM called *CodeLlama* as the foundation code LLM. We aim to compare the performance of code LLMs released at different times on the target task. We also adopt three fine-tuning epochs for this approach.

- ***SelRepairT5***: To verify that our approach also improves in code LLM with small-scale parameters, we replace the foundation code LLM with *CodeT5*. Referring to the design of *RAP-Gen*, we adopt 50 fine-tuning epochs for this approach.

- ***SelRepairLoRA***: Considering PEFT-based methods, we try to adopt LoRA fine-tuning strategy for *SelRepair*. LoRA (Low-Rank Adaptation) (Hu et al., 2021) is an approach for fine-tuning LLMs. It enables efficient fine-tuning by adjusting some of the weights of the model without significantly increasing the number of parameters. We also adopt three fine-tuning epochs for this approach.

F Discussions

F.1 Detailed Process of How *SelRepair* Works

We present a real buggy code snippet in the test set and show how *SelRepair* fixes this buggy code. The detailed fixing process is shown in Figure 6. The size variable in line 3 needs to be replaced with variable `arr.length` to ensure the rationalization of array expansion. When *SelRepair* receives the buggy code, the code will be parsed into AST. The code and AST will be input to the SR and SSDR module to get the feature vector and calculate the similarity with each sample in the codebase. Then we adopt the selection gate and set the similarity threshold to 0.8. A bug-fix pair in the codebase is retrieved and the similarity is 0.9592. The bug-fix pair provides a similar fix pattern so that *SelRepair* can fix the bug with the use of this RAG information.

F.2 Case Study

In this section, we propose a patch case generated by the *SelRepair* and other SOTAs. Figure 7 presents an example from Tufano Subset 2 (50-100 tokens). The buggy code at line 8 incorrectly uses the `appendQuoted` method instead of `append`. The key difference between them is as follows:

- `append`: This method is used to append a string value to the existing content of a StringBuilder object without adding any quotation marks.
- `appendQuoted`: This method is specifically designed to append a string value to the StringBuilder object while enclosing the value in quotation marks.

In the given context, using `append` is the appropriate choice since the `getAliasName()` method already returns the column name without quotation marks. Using `appendQuoted` may result in extra quotation marks being added around the column name, leading to incorrect syntax. Among the compared approaches, only *SelRepair* successfully generates the correct patch for this bug. In contrast, *SelRepairT5* and *SelRepairLoRA* generate the same code as buggy code. *SelRepairLlama* changes `public static` to `package`, which indicates that this approach misunderstands method `fcolumnsWithFunction`. *GPT-3.5* makes an invalid modification that change the type of `functionName` from `java.lang.String` to `String`. *GPT-4o* also makes the invalid modification (change `public static` to `function`. As for *DeepSeek-R1-Distill*, we find that it outputs excessively long reasoning content, suffers from over-reasoning, and does not generate the repaired code at the end, which may indicate that the model is impaired in comprehending this code. Therefore, we do not present the content generated by *DeepSeek-R1-Distill*. Considering *RAP-Gen*, it cannot comprehend the semantics of the code and generate the wrong patch.

The case highlights *SelRepair*'s ability to generate accurate patches for code implementation errors that cannot be detected by static analysis tools. Such errors often require a deeper understanding of code semantics and the intended functionality. By accurately fixing these implementation errors, *SelRepair* demonstrates its robustness and effectiveness in program repair tasks. It showcases the model's ability to comprehend the nuances of code semantics and generate patches that align with the intended functionality, even when errors are not detectable by traditional static analysis tools.

F.3 Performance on *Defects4J*

Defects4J (Just et al., 2014) is one of the most widely adopted APR datasets. Based on our collation of its two versions (v1.2 and v2.0), *Defects4J* contains 1,273 bug-fix pairs at the method level from 17 open-source Java projects on GitHub. As mention in (Wang et al., 2023b), *RAP-Gen* adopts a project-specific training data curated by SelfAPR (Ye et al., 2023a) and evaluate *Defects4J*. Specifically, *RAP-Gen* is trained with a dataset constructed by the same projects as *Defects4J*. This configuration may cause data leaks and weaken the generalizability of the approach. Therefore, referring to (Huang et al., 2023), we utilize a dataset proposed by Jiang et al. (Jiang et al., 2023) to fine-tune the *SelRepair* and test the performance on *Defects4J*. Besides, we adopt beam search and set the beam size as 10. For a fair comparison, we set *GPT-3.5* to generate 10 patches for each bug. We count the number of patches that can pass the test cases. The results are shown in Table 5. It can be found that *SelRepair* can generate 35 patches and 11 patches for v1.2 and v2.0, respectively. As for *RAP-Gen* fine-tuned with the data from the same projects, it can generate 32 patches for v1.2 and 12 patches for v2.0. *SelRepair* outperforms RAP-Gen at a beam size of 10 in Defects4J V1.2 and achieves a close performance to RAP-Gen in Defects4J V2.0. In general, *SelRepair* has better performance on cross-project APR data. In particular, we do not consider some other prompt-engineering-based approaches such as (Li et al., 2024). Although this kind of approach is orthogonal to *SelRepair* and may have better results for simple APR tasks, for complex tasks, it is not possible to extend beyond the original capabilities of the LLMs due to the dependence on the capabilities of the pre-trained models, so our approach extends the complex APR capabilities through fine-tuning, which is based on the addition of prompt engineering.

Buggy Code

```
public static com.raizlabs.android.dbflow.sql.language.ColumnAlias columnsWithFunction(java.lang.String functionName,
com.raizlabs.android.dbflow.sql.language.ColumnAlias... columnAliases) {
   com.raizlabs.android.dbflow.sql.QueryBuilder queryBuilder = new
com.raizlabs.android.dbflow.sql.QueryBuilder(functionName).append("(");
   for (int i = 0; i < (columnAliases.length); i++) {
      if (i > 0) {
         queryBuilder.append(",");
      } queryBuilder.appendQuoted(columnAliases[i].getAliasName());
   } queryBuilder.append(")");
   return com.raizlabs.android.dbflow.sql.language.ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

SelRepair

```
public static com.raizlabs.android.dbflow.sql.language.ColumnAlias columnsWithFunction(java.lang.String functionName,
com.raizlabs.android.dbflow.sql.language.ColumnAlias... columnAliases) {
   com.raizlabs.android.dbflow.sql.QueryBuilder queryBuilder = new
com.raizlabs.android.dbflow.sql.QueryBuilder(functionName).append("(");
   for (int i = 0; i < (columnAliases.length); i++) {
      if (i > 0) {
         queryBuilder.append(",");
      } queryBuilder.append(columnAliases[i].getAliasName());
   } queryBuilder.append(")");
   return com.raizlabs.android.dbflow.sql.language.ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

SelRepairT5

```
public static com.raizlabs.android.dbflow.sql.language.ColumnAlias columnsWithFunction(java.lang.String functionName,
com.raizlabs.android.dbflow.sql.language.ColumnAlias... columnAliases) {
   com.raizlabs.android.dbflow.sql.QueryBuilder queryBuilder = new
com.raizlabs.android.dbflow.sql.QueryBuilder(functionName).append("(");
   for (int i = 0; i < (columnAliases.length); i++) {
      if (i > 0) {
         queryBuilder.append(",");
      } queryBuilder.appendQuoted(columnAliases[i].getAliasName());
   } queryBuilder.append(")");
   return com.raizlabs.android.dbflow.sql.language.ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

SelRepairLoRA

```
public static com.raizlabs.android.dbflow.sql.language.ColumnAlias columnsWithFunction(java.lang.String functionName,
com.raizlabs.android.dbflow.sql.language.ColumnAlias... columnAliases) {
   com.raizlabs.android.dbflow.sql.QueryBuilder queryBuilder = new
com.raizlabs.android.dbflow.sql.QueryBuilder(functionName).append("(");
   for (int i = 0; i < (columnAliases.length); i++) {
      if (i > 0) {
      queryBuilder.append(",");
      } queryBuilder.appendQuoted(columnAliases[i].getAliasName());
   } queryBuilder.append(")");
   return com.raizlabs.android.dbflow.sql.language.ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

SelRepairLlama

```
package com.raizlabs.android.dbflow.sql.language.ColumnAlias columnsWithFunction(java.lang.String functionName,
com.raizlabs.android.dbflow.sql.language.ColumnAlias... columnAliases) {
   com.raizlabs.android.dbflow.sql.QueryBuilder queryBuilder = new
com.raizlabs.android.dbflow.sql.QueryBuilder(functionName).append("(");
 for (int i = 0; i < columnAliases.length; i++) {
      if (i > 0) {
         queryBuilder.append(",");
      } queryBuilder.appendQuoted(columnAliases[i].getAliasName());
   } queryBuilder.append(")");
   return com.raizlabs.android.dbflow.sql.language.ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

GPT-3.5

```
public static com.raizlabs.android.dbflow.sql.language.ColumnAlias columnsWithFunction(String functionName,
com.raizlabs.android.dbflow.sql.language.ColumnAlias... columnAliases) {
   com.raizlabs.android.dbflow.sql.QueryBuilder queryBuilder = new
com.raizlabs.android.dbflow.sql.QueryBuilder(functionName).append("(");
   for (int i = 0; i < columnAliases.length; i++) {
      if (i > 0) {
         queryBuilder.append(",");
      } queryBuilder.appendQuoted(columnAliases[i].getAliasName());
   } queryBuilder.append(")");
   return com.raizlabs.android.dbflow.sql.language.ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

GPT-4o

```
function columnsWithFunction(functionName, ...columnAliases) {
   let queryBuilder = new QueryBuilder(functionName).append("(");
   for (let i = 0; i < columnAliases.length; i++) {
      if (i > 0) {
         queryBuilder.append(",");
      }
      queryBuilder.appendQuoted(columnAliases[i].getAliasName());
   }
   queryBuilder.append(")");
   return ColumnAlias.columnRaw(queryBuilder.getQuery());
}
```

RAP-Gen

```
private void appendTypeVarsFromEnclosingFunctions ( java.util.List <gw.lang.ir.IRSymbol > parameters,
gw.internal.gosu.parser.IGosuClassInternal gs_Class ) {
   while ( gsClass. isEmpty ( ) ) {
      gw.lang.parser.IDynamicFunctionSymbol dfs = getEnclosingDFS ( gsClass ) ;
      if (dfs == null ) {
         break ;
      } if ( VAR_lang.reflect.Modifier.isReified (dfs. getModifiers ( ) ) ) {
         add ( TYPE_2 ) ;
      } gs_Class = ( VAR.internal.gos.IG.IGosuClassInternal ) ( VAR.getGosuClass ( ) ) ;
   }
}
```

Figure 7: Case Study

G Threats to Validity

The threats to validity include internal validity, external validity and construct validity.

Internal validity addresses the correctness and reliability of our experiments and data processing. Issues can arise from errors in the bug-fix dataset and biases during language model fine-tuning, such as overfitting. To mitigate these, we implemented rigorous data preprocessing and validation steps. Another concern is the threshold settings

Table 5: Performance on *Defects4J*

Approaches	Defects4J V1.2	Defects4J V2.0
***SelRepair* (Beam Size = 10)**	35	11
***RAP-Gen* (Beam Size = 10)**	32	12
***GPT-3.5*(10 Generated Patches)**	11	3

in the RAG selection gate, where coarse-grained thresholds were used for different code lengths. Future work will focus on automatically setting customized thresholds for diverse code lengths.

External validity concerns whether our findings extend beyond the Java and C/C++ datasets used. While *SelRepair* showed promise in repairing Java programs, its effectiveness with other languages like Python or JavaScript is untested. Language syntax, semantics, and bug patterns may impact performance. Future work will involve evaluating diverse datasets from multiple languages to assess and refine *SelRepair*'s adaptability, ensuring broader applicability and robustness across different software development environments.

Construct validity ensures our metrics and benchmarks accurately reflect program repair effectiveness. We plan to evaluate our approach on diverse datasets from different lengths to ensure generalizability and compare results with established benchmarks and other methods. Testing in real-world environments will assess practical applicability. Developer feedback will provide insights into perceived utility and accuracy. So far, we have used open-source data for training and testing. We also use an internal enterprise dataset to ensure broader applicability. These steps will strengthen construct validity by ensuring accurate and applicable performance across contexts.

H Data Availability

We make our approach available at https://anonymous.4open.science/r/SelRepair-5F1D/.