Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair

Kai Huang, Jian Zhang, Xinlei Bao, Xu Wang, and Yang Liu, *Senior Member, IEEE*

***Abstract*—Automated program repair (APR) research has entered the era of large language models (LLM), and researchers have conducted several empirical studies to explore the repair capabilities of LLMs for APR. Many studies adopt the zero/few-shot learning paradigm for APR, which directly use LLMs to generate the possibly correct code given its surrounding context. Though effective, the repair capabilities of LLMs based on the fine-tuning paradigm have yet to be extensively explored. Also, it remains unknown whether LLMs have the potential to repair more complicated bugs (e.g., multi-hunk bugs). To fill the gap, in the conference version of this work, we conduct an initial study on the program repair capability of million-level LLMs in the fine-tuning paradigm. We select 5 popular million-level LLMs with representative pre-training architectures, including CodeBERT, GraphCodeBERT, PLBART, CodeT5, and UniXcoder. We consider 3 typical program repair scenarios (i.e., bugs, vulnerabilities, and errors) involving 3 programming languages (i.e., Java, C/C++, and JavaScript). Our experimental results show that fine-tuning these LLMs can significantly outperform previous state-of-the-art APR tools. However, the repair capabilities of billion-level LLMs for APR remain largely unexplored. Moreover, their substantial model sizes significantly increase the computational cost of fine-tuning. While parameter-efficient fine-tuning (PEFT) techniques offer a promising solution, their effectiveness in repair tasks and the selection of appropriate PEFT strategies remain unclear. Similarly, many novel APR strategies have been developed for non-pre-trained models, yet their applicability and effectiveness on LLMs are still unexamined. To address these gaps, we extend our prior study through three key dimensions: 1) LLM4APR, which evaluates the repair capabilities of five billion-level LLM families (InCoder, CodeGeeX, CodeGen, StarCoder, and CodeLlama) under the fine-tuning paradigm; 2) PEFT4LLM, which compares full-parameter fine-tuning (FPFT) with three PEFT techniques (LoRA, AdaLoRA, and IA3) to determine optimal strategies that balance repair cost and performance of LLMs; and 3) APR4LLM, which investigates the potential of a basic neural machine translation (NMT) approach alongside three advanced repair strategies (TENURE, ITER, and KATANA) to enhance the repair capabilities of LLMs. Overall, our extensive results suggest that larger scale models typically have better repair capabilities. The LoRA technique is still the best choice for LLM4APR studies. Different repair strategies result in different repair capabilities for the foundation models, but some of the strategies that performed well on the non-pre-trained model did not show an advantage on LLMs. Besides, we released all LLMs fine-tuned with repair tasks to facilitate LLM4APR research, and we encourage researchers to develop more powerful APR tools on the basis of these repair LLMs.**

***Index Terms*—Large language model, automated program repair, fine-tuning.**

Received 24 July 2024; revised 21 December 2024; accepted 16 January 2025. Date of publication 23 January 2025; date of current version 18 April 2025. This work was supported in part by the National Research Foundation, Singapore, and DSO National Laboratories through the AI Singapore Programme (AISG) under Grant AISG2-GC-2023-008, in part by the National Research Foundation Singapore and the Cyber Security Agency through the National Cybersecurity R&D Programme under Grant NCRP25-P04-TAICeN, and in part by the National Research Foundation, Prime Minister's Office, Singapore through the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Recommended for acceptance by Á. Beszédes. *(Corresponding author: Jian Zhang.)*

Kai Huang is with the Technical University of Munich, 74076 Heilbronn, Germany (e-mail: kai-kevin.huang@tum.de)

Jian Zhang and Yang Liu are with Nanyang Technological University, Singapore 639798 (e-mail: jian_zhang@ntu.edu.sg; yangliu@ntu.edu.sg).

Xinlei Bao and Xu Wang are with Beihang University, Beijing 100191, China (e-mail: leileidesu233@buaa.edu.cn; xuwang@buaa.edu.cn).

Digital Object Identifier 10.1109/TSE.2025.3532759

I. Introduction

Automated program repair (APR) techniques [1], [2], [3], [4], [5], [6] aim to automate the repair of software defects to reduce manual work and guarantee the software quality. Among them, learning-based APR techniques [4] have attracted much attention in recent years. In general, the Neural Machine Translation (NMT) model is adopted for supervised training on bug-fix pairs (BFPs) [7], which translates the buggy program to its fixed version. Compared to traditional APR techniques [1], [2], [3], both the quantity and diversity of fixed bugs have been improved through the use of learning-based APR tools [4].

In particular, with the recent rapid development of large language models (LLMs) [8], [9], APR research has also entered the era of LLMs. Currently, researchers have conducted several empirical studies [10], [11], [12], [13], [14], [15], [16] to explore the repair capability of LLMs and proposed many LLM-based APR tools or frameworks [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29]. For example, recent empirical studies [10], [11], [12], [13], [14], [15], [16] have explored the repair capability of LLMs under fine-tuning and prompt learning (or zero/few-shot learning) paradigms, and have shown that fine-tuning is an effective means to further enhance the repair capability of LLMs. Among the research papers, Xia et al. [19] propose ChatRepair, a conversational repair paradigm based on ChatGPT; Jin et al. [23] implement fine-tuning of Codex to build an end-to-end APR tool, InferFix, for industrial deployments; and Wei et al. [20] propose the Repilot framework to assist the LLM to better synthesize patches and thus enhance the repair capability.

Despite all this, most existing studies [10], [11], [16], [17], [19], [20], [22], [28], [30] focus on directly using the LLM with zero-shot or few-shot prompting, while the benefit of fine-tuning LLMs for APR is not yet fully understood [5]. The missed opportunities are twofold: 1) Even though Jiang et al. [13] explored the LLM fine-tuning, however, as stated, the applied fine-tuning is straightforward and simple. It is still unclear regarding the impact of specific design choices with fine-tuning, and what factors limit the repair capability of LLMs. 2) The scope of prior work typically lies within limited bug types (e.g., single-hunk and common bugs), the generality of APR tools has been largely neglected (e.g., multi-hunk bugs and vulnerabilities).

To address the above issues, the conference version of this paper [15] comprehensively explores the repair ability of million-level LLMs under the NMT fine-tuning paradigm, aiming to provide more empirical guidance for the research of Large Language Models for Automated Program Repair (LLM4APR). Our initial LLM4APR study investigates the repair capability of 5 million-level LLMs in the NMT fine-tuning paradigm by following an encoder-decoder generation procedure on BFPs. We apply the fine-tuned LLMs on 7 popular evaluation benchmarks of different programming languages (PLs), and take into account software bugs, security vulnerabilities, and programming errors. For simplicity, we collectively refer to them as *defects*. Apart from single-hunk defects, we also evaluate the ability of LLMs for repairing multi-hunk defects. Specifically, we focus on the following key aspects when using million-level LLMs for APR.

Repair Effectiveness of Million-level LLMs. We study the repair effectiveness of 5 million-level LLMs under the NMT fine-tuning paradigm in 6 repair tasks (Section IV-C2 Task ❶-❻). Our results show that LLMs' repair capability outperforms many previous APR tools. For bug repair, the best model fixes 34 and 25 bugs more than Jiang et al. [13] and Li et al. [31]; For vulnerability repair, the best repair accuracy is improved by 20.04% compared to VulRepair [24]. For error repair, our best repair accuracy outperforms TFix [32] by 11.32%. Besides, we also examine the performance of LLMs for multi-hunk fixes. The best million-level LLM fixed 9 more multi-hunk bugs than DEAR [31]. These results demonstrate that fine-tuning million-level LLMs has great potential for APR research.

Design Choices. We undertake an in-depth exploration of different design choices in fine-tuning (Section IV-C1), including code abstraction [7] (Section II-A2), code representation [33] (Section II-A3), and checkpoint selection [34] (Section II-C). The results imply that: 1) The code abstraction strategy used in earlier work [7] is unsuitable for LLMs and may even reduce the repair capability of LLMs. 2) Using the representation of buggy code with fault locations and fix code without surrounding tokens can yield better repair results. 3) Different repair scenarios may require different evaluation metrics for checkpoint selection to obtain the best fine-tuned model. And we find the ensemble strategy that combines multiple selected checkpoints is a good way to enhance the repair effectiveness.

Future Directions. The limitations and challenges of fine-tuning LLMs for APR are also analyzed (Section IV-C3). We thoroughly discussed two major factors that affect the repair capability of LLMs. On the one hand, LLMs often fail to generate correct patches due to the lack of essential repair ingredients [35]. For instance, important elements such as method names, variables, or other components related to the buggy code may not be captured by the model, preventing it from fully understanding the root cause of the bug and leading to incorrect fixes [29]. On the other hand, the candidate patch space is greatly restricted. For example, when using the beam search strategy to generate patches for LLMs, the available patch space is typically constrained by GPU memory, making it difficult to scale to larger patch sizes [36]. Since the hardware resources often cannot meet the high computational demands of LLMs for massive patches. Based on this, we propose some mitigation measures and future directions.

However, the rapid development of LLMs has brought new challenges and opportunities, which were not involved in our previous LLM4APR study [15]. These include: 1) Large-scale billion-level LLMs built on decoder-only architecture have yet to be fully investigated. Considering the computing resource constraints, our previous study [15] only covered million-level LLMs. Besides, although recent studies [13], [14] have implemented fine-tuning on billion-level LLMs, however, they used model sizes no more than 6B (i.e., InCoder-6B and CodeGen-6B), and such model sizes do not fully reflect the current state-of-the-art LLMs (e.g., CodeLlama-70B and StarCoder-15B) in terms of their repair capabilities. 2) The effectiveness of different parameter-efficient fine-tuning (PEFT) techniques [37] designed for large-scale LLMs is unclear. Fine-tuning large-scale billion-level LLMs under computing resource constraints is difficult, fortunately the emergence of PEFT techniques has alleviated this challenge. PEFT techniques are designed to save memory without sacrificing performance. Although PEFT techniques have been applied to APR work [26], it remains unclear how to select appropriate PEFT techniques to better unlock the repair capability of LLMs. 3) Potential repair strategies for improving those decoder-only LLMs remain unexplored. In fact, researchers have proposed a variety of novel repair strategies [33], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49] and implemented them on non-pre-trained models (e.g., RNN, CNN, Transformer, etc.). Theoretically, many repair strategies are model-independent, so these approaches can be applied to LLMs as well. However, it is not clear whether these novel approaches work well on LLMs or whether these strategies can further stimulate the repair potential of LLMs.

To bridge above gaps, this study extend our previous work by further focusing on the repair ability of billion-level LLMs under the NMT fine-tuning paradigm. We investigate the repair capability of 5 billion-level LLM families for APR (i.e., LLM4APR study) on the test benchmark without the data leakage risk. In particular, we explore the impact of mainstream PEFT techniques for LLMs (i.e., PEFT4LLM study) and reveal the generalization ability of novel APR techniques for LLMs (i.e., APR4LLM study) on the program repair task. Specifically,

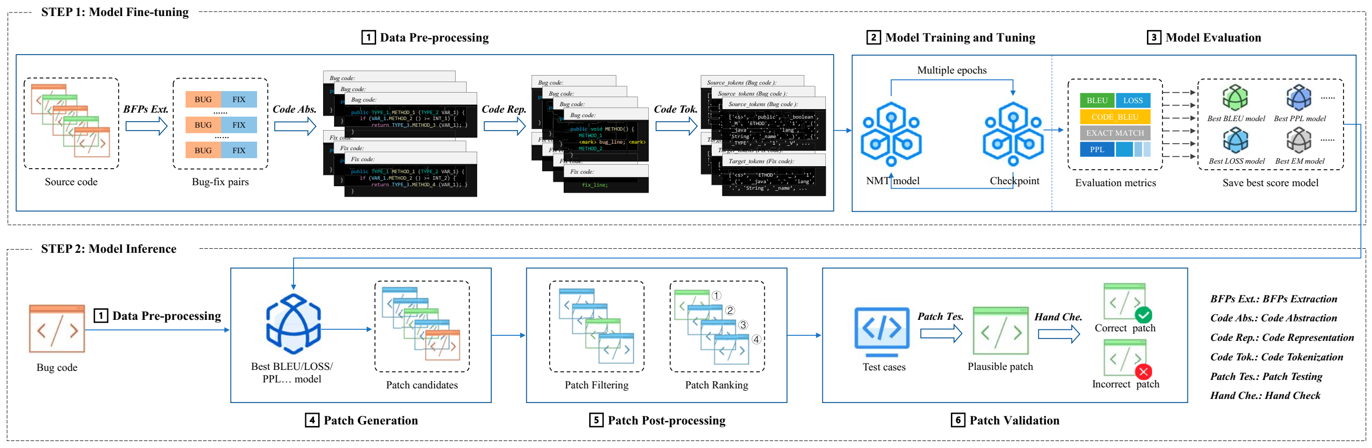


Fig. 1. The workflow of NMT fine-tuning based LLM4APR.

we focus on the following keys aspects when using billion-level LLMs for APR.

Repair Effectiveness of Billion-level LLMs. We study the repair effectiveness of 5 billion-level LLM families under the NMT fine-tuning paradigm on Defects4J V1.2 [50] and HumanEval-Java [13] (Section IV-C1 Task ❼). Our results show that larger model sizes bring better repair capabilities while also tend to take more memory cost and time cost. Therefore, how to reduce the memory as well as accelerate the speed of model training and inference in LLM4APR research is a key issue.

PEFT Options. We explore the impact of 3 PEFT techniques [37] (i.e., LoRA [51], AdaLoRA [52], and IA3 [53]) on billion-level LLMs for APR research (Section V). Our results indicate that PEFT techniques can outperform full-parameter fine-tuning (FPFT) [13] strategy in LLM4APR research. For example, the repair effectiveness of LLMs after fine-tuning with the LoRA technique rivaled or even exceeded that of the FPFT strategy. Notably, some more recent PEFT techniques did not show significant advantages in the APR task. For example, AdaLoRA does not show significant improvement in repair effectiveness, memory cost, and time cost compared to other PEFT or FPFT strategies. And our results suggest that LoRA remains the best choice for LLM4APR. However, it is undeniable that the PEFT technique slows down model training while reducing the GPU memory cost.

Repair Strategies. We examine the effects of 3 novel repair strategies - TENURE [49], ITER [45], and KATANA [54] along with the basic NMT strategy [7], [13] in enhancing the repair capabilities of LLMs (Section VI). Our results indicate that these state-of-the-art APR techniques can be generalized to LLMs, yielding effective repair results. Moreover, each repair strategy contributes uniquely to the repair capabilities of LLMs. However, some repair strategies, when implemented with LLMs, perform worse than expected. For instance, the implementation of TENURE on LLMs underperformed compared to the basic NMT fine-tuning. This highlights the need to revisit the effectiveness of previous APR techniques in the era of LLMs.

To sum up, this paper makes the following contributions:

- **Initial Study Contributions.** The conference version [15] of this paper presents following contributions: First, we study the repair effectiveness of 5 million-level LLMs (CodeBERT, GraphCodeBERT, PLBART, CodeT5, and UniXcoder) in the fine-tuning paradigm for APR across 3 typical repair scenarios (software bugs, security vulnerabilities, and programming errors). Second, we conduct an in-depth investigation of design choices that may enhance the repair capability, and achieve significant improvements over vanilla fine-tuning using same models. Third, our study reveals that LLMs are capable of repairing fairly complex multi-hunk defects to some extent, and there is a good potential to tackle more complicated ones. Fourth, we analyze several key factors that limit the repair capability of LLMs and propose feasible solutions to mitigate them.
- **Extended Study Contributions.** The extended parts of this paper presents following contributions: First, we study the repair effectiveness of 5 billion-level LLM families (InCoder, CodeGeeX, CodeGen, CodeLlama, and StarCoder) under the NTM fine-tuning paradigm. Second, we provide guidance on PEFT options for LLM4APR research to better unlock LLMs' repair performance under the computing resource constraint. Third, we reveal the potential of recent novel repair approaches in enhancing the repair capabilities of LLMs. Finally, we release 22 fine-tuned LLMs using different repair strategies and encourage future researchers to develop more powerful APR tools based on these LLMs. Our artifacts including the code and data are made publicly available [55].

II. Methodology

In this section, we present the methodology of LLM4APR. The workflow of fine-tuning LLMs for APR follows the basic learning-based APR technique [4], as shown in Fig. 1. Generally, applying LLMs to the APR workflow in the NMT fine-tuning paradigm involves the following steps: 1) data pre-processing, 2) model training and tuning, 3) model evaluation, 4) patch generation, 5) patch post-processing, and 6) patch validation. Next, we describe the technical details of each step.

A. Data Pre-Processing

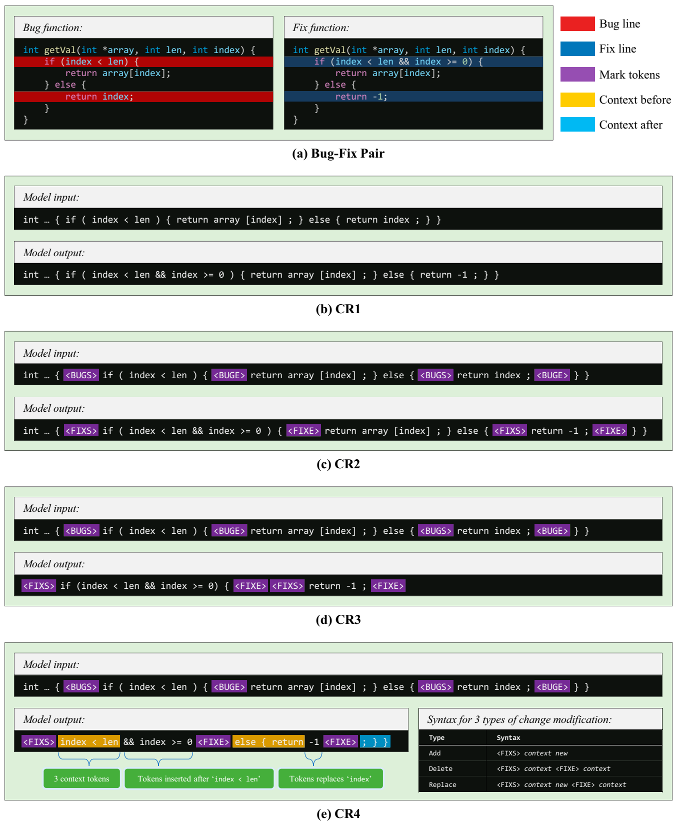
Data pre-processing phase aims to convert the raw source code into a format that LLM can efficiently process. We adopt the common practice of using BFPs [7] for learning to transform the buggy code to fixed code at the method level.

1) BFPs Extraction: Here, we follow the basic NMT processing workflow, using training data of BFPs [7] to learn for converting the buggy code to fixed code. We adopt the approach of most work, extracting method-level BFPs from the source code; this is done because the input/output length of models is usually limited, and it is often difficult to process the entire source code file. Therefore, it is a common practice to use method-level contextual information.

2) Code Abstraction: Code abstraction processing was first introduced to bug repair tasks by Tufano et al. [7]. This technique alleviates the out-of-vocabulary (OOV) problem by normalizing code elements and facilitates models to learn generic fixing patterns [4], [5]. Subsequently, many following works [31], [38], [42], [56] adopt the similar strategy for improvement. However, it is unclear whether code abstraction could benefit LLMs. Therefore, we explore the impact of code abstraction [7] as the first design choice.

3) Code Representation: In learning-based APR techniques, code representation is an essential factor for the repair capability. Earlier works only focused on single-hunk bugs and design the code representation specific to them. Recently, VRepair [33] has extended the NMT model to multi-hunk fixes by improving the code representation. To explore the impact of different code representations on LLMs' repair capability, we consider four code representations, abbreviated as *CR1*, *CR2*, *CR3*, and *CR4*. As shown in Fig. 2, all of them are based on token sequence because existing LLMs are generally limited to such a representation. The details are illustrated below.

- **CR1**: This is the original representation of NMT-based APR work [7], which takes a whole buggy method as input and a whole fixed method as output. *CR1* aims to allow the model to automatically fix defects without fault localization (FL).
- **CR2**: *CR2* is based on CR1, where the bug/fix hunk are marked with special tokens (<BUGS>, <BUGE>, <FIXS>, <FIXE>) so that the model learns the transition from bug code to fixed code with the help of FL information. Hence, we use it to analyze the impact of FL information on the repair capability.
- **CR3**: Inspired by SequenceR [38], we remove the context of fixed code from *CR2* to reduce the model output length and speed up the training and prediction. This representation is used to analyze the impact of simplifying the learning target (i.e., the output) on the repair capability.
- **CR4**: This is VRepair's code representation for multi-hunk fixes [33]. Unlike *CR3*, *CR4* uses different mark ways to distinguish between different repair behaviors (i.e., add, delete, replace) and therefore has a finer marker granularity. Through comparison, we can analyze the impact of fine-grained representation on repair capability.



Bug function:

```
int getVal(int *array, int len, int index) {
    if (index < len) {
        return array[index];
    } else {
        return index;
    }
}
```

Fix function:

```
int getVal(int *array, int len, int index) {
    if (index < len && index >= 0) {
        return array[index];
    } else {
        return -1;
    }
}
```

(a) Bug-Fix Pair

Model input:

```
int … { if ( index < len ) { return array [index] ; } else { return index ; } }
```

Model output:

```
int … { if ( index < len && index >= 0 ) { return array [index] ; } else { return -1 ; } }
```

(b) CR1

Model input:

```
int … { <BUGS> if ( index < len ) { <BUGE> return array [index] ; } else { <BUGS> return index ; <BUGE> } }
```

Model output:

```
int … { <FIXS> if ( index < len && index >= 0 ) { <FIXE> return array [index] ; } else { <FIXS> return -1 ; <FIXE> } }
```

(c) CR2

Model input:

```
int … { <BUGS> if ( index < len ) { <BUGE> return array [index] ; } else { <BUGS> return index ; <BUGE> } }
```

Model output:

```
<FIXS> if (index < len && index >= 0) { <FIXE> <FIXS> return -1 ; <FIXE>
```

(d) CR3

Model input:

```
int … { <BUGS> if ( index < len ) { <BUGE> return array [index] ; } else { <BUGS> return index ; <BUGE> } }
```

Model output:

```
<FIXS> index < len && index >= 0 <FIXE> else { return -1 <FIXE> ; } }
```

Syntax for 3 types of change modification:

Type	Syntax
Add	<FIXS> *context new*
Delete	<FIXS> *context* <FIXE> *context*
Replace	<FIXS> *context new* <FIXE> *context*

(e) CR4

Fig. 2. Four code representation forms.

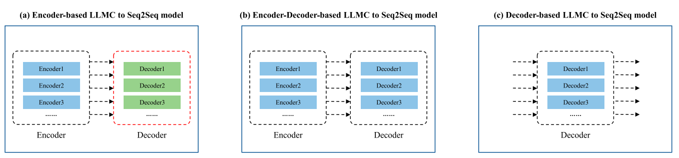


Fig. 3. Extending the LLM to NMT model architecture.

Note that the above four code representations support both single-hunk and multi-hunk repair scenarios.

4) Code Tokenization: Following previous works [24], [32], [40], [56], [57], we use a subword-level tokenizer namely byte-pair encoding (BPE). It replaces frequently occurring sequences of characters with a single symbol, resulting in a more compact vocabulary. It can effectively alleviate the OOV problem in APR [4], [5] and is superior to the word-level tokenizer [24]. Therefore, we follow previous works' experience [24], [32], [40], [56], [57] using the BPE tokenization technique.

B. Model Training and Tuning

This step aims to extend LLMs into the NMT model architecture for fine-tuning. As shown in Fig. 3, we show the model architectures for the three classes of LLMs when applied to the NMT task. For encoder-only LLMs, we add decoders to build the Seq2Seq architecture and fine-tune them in a supervised manner. For encoder-decoder LLMs, they are the Seq2Seq architecture, so no changes to the structure are needed. For Decoder-based LLMs, this type of generative model can be directly applied to sequence prediction and can be applied to NMT tasks as well. However, for decoder-only LLMs, such generative models need to concatenate the input and output for fine-tuning, which weakens the ability of understanding buggy code semantics due to the length limit. And a recent study [58] indicates that decoder-only million-level LLMs can perform significantly worse than the above two kinds of LLMs. Considering the computing resource constraint, our initial study main focus on encoder-only and encoder-decoder LLMs. Nowadays, as Decoder-only architectures have become the dominant trend in LLM and PEFT techniques have alleviated the computing resource constraint, our extended study will mainly focus on larger scale decoder-only LLMs. After building the NMT model, multiple training iterations are performed on the training dataset to enable the model to learn the domain knowledge for defect repair.

C. Model Evaluation

During model training and tuning, the performance of checkpoints after each training round need to be evaluated on the validation set to find the best trained model (i.e., checkpoint selection). Researchers have proposed various metrics for model evaluation [59], such as *PPL*, *BLEU*, *Code BLEU*, etc. In previous APR works based on LLM fine-tuning, Mashhadi et al. [34] and Huang et al. [60] used the BLEU score to evaluate model quality in real-time to filter the best repair model, and Jiang et al. [13] used the PPL score as the evaluation metric. BLEU (i.e., Bilingual Evaluation Understudy) [61] is a string-based metric, is largely adopted in text translation and summarization, which measures how similar the predicted sequence and the ground truth are. PPL (i.e., Perplexity) [61] is a reference-less metric, is a standard evaluation metric in natural language generation tasks, which quantifies how well a language model predicts data by evaluating predictive capabilities. However, it is still unclear how they can affect the selection of the best repair model. Therefore, we explore them to guide the checkpoint selection in APR tasks. In our experiments, we used the metrics PPL and BLEU (which are commonly used in previous works [13], [34], [60]) to derive the best models, which we call the *best PPL/BLEU model*. Besides, we also keep the last round of checkpoints that is irrelevant evaluation metrics, which we call the *Last model*.

D. Patch Generation

In the patch generation phase, we use the repair model obtained in the model evaluation phase to perform patch synthesis. Generally, we generate multiple candidate patches for an input bug method, which we call the candidate patch space. Based on the experience of previous work [13], [14], [31], [38], [39], [40], [41], [42], [46], [47], we use the beam search strategy to control the patch generation. We generate a specified number of candidate patches by adjusting the beam size (or beam width).

E. Patch Post-Processing

The purpose of patch post-processing is to filter and rank the candidate patches to filter out patches that violate programming language rules (e.g., syntax error patches) and to push the most likely correct candidates to the front of the patch queue. There are many post-processing strategies [4], [5] that have been born out of APR techniques. Notably, the post-processing step is independent of the LLM patch generation capability, and a previous study [16] has explored the impact of the post-processing step on improving LLMs' repair results. Therefore, we exclude the effect of the post-processing strategy on repair capability. We do not perform any post-processing strategies during specific experiments.

F. Patch Validation

Patch validation aims to filter out the correct patch from the candidate patch space. In the benchmark with test cases, we followed the validation strategy from previous works [17], [31], [38], [39], [40], [42], [43], [44], [46], [48], [57]. Specifically, for each bug version, we allocate a maximum of 5 hours for the validation run, retaining only the first (top-1) plausible patch candidate that successfully passes all test cases. Then two authors manually check the plausible patches to determine whether a plausible patch is correct or incorrect patch. Finally, the result is *correct patches / plausible patches* (X/Y). In the benchmark without test cases, we follow previous works [7], [24], [32], [33], [56], [62] and use the *exact match* strategy to calculate the *repair accuracy* (Z%).

III. Experimental Setup

In this section, we describe the experimental details of this study, which unfolds in four main aspects: 1) The research questions of this study. 2) LLMs chosen for the study. 3) PEFT techniques chosen for the study. 4) APR strategies chosen for the study.

A. Research Questions

In the initial study section, we explore the repair capability of million-level LLMs in different scenarios by answering the following research questions in software bug repair, security vulnerability repair, and programming error repair, respectively:

- **RQ1: How do different design choices affect LLMs' repair capability?** RQ1 investigates the impact of different design choices on LLMs' repair capability, which can help better compare LLMs and provide guidance on fine-tuning LLMs. We will explore the impact of code abstraction, code representation, and checkpoint selection on the results.
- **RQ2: How well does the million-level LLM perform compared to the state-of-the-art APR works?** RQ2 aims to explore the repair capability of million-level LLMs. We systematically evaluate their performance under multiple defect types, programming languages, and defect complexities. Further, we compare LLMs to SOTA APR works to know whether LLMs are superior.

TABLE I
DETAILS OF THE SELECTED LLMs

Initial Study				Extended Study			
Model	**Size**	**Type**	**Pre-training Dataset**	**Model**	**Size**	**Type**	**Pre-training Dataset**
CodeBERT-base [63]	125M	Encoder-only	CodeSearchNet [64]	StarCoder(2)-15B [65], [66]	15B	Decoder-only	The Stack v1.2, The Stack v2
GraphCodeBERT-base [67]	125M	Encoder-only	CodeSearchNet [64]	CodeLlama-13/34/70B [68], [69], [70]	13/34/70B	Decoder-only	Unavailable
PLBART-base [71]	140M	Encoder-Decoder	StackOverflow and BigQuery	CodeGen25-7B [72]	7B	Decoder-only	The Stack v1.2
CodeT5-base [73]	220M	Encoder-Decoder	CodeSearchNet [64] and BigQuery	CodeGeeX2-6B [74]	6B	Decoder-only	Pile, CodeParrot, GitHub
UniXcoder-base [75]	125M	Encoder-Decoder	CodeSearchNet [64]	InCoder-6B [76]	6B	Decoder-only	GitHub & GitLab, StackOverflow

- **RQ3: What are the factors that limit the effectiveness of fine-tuning LLMs?** RQ3 aims to reveal the shortcomings of LLMs for APR tasks when fine-tuning and point out some future directions for improvement.

In the extended study section, we explore the repair capability of billion-level LLMs by answering the following research questions:

- **RQ4: How well does the billion-level LLM perform compared to the state-of-the-art APR works?** RQ4 aims to explore the repair capability of billion-level LLMs. We systematically evaluate their performance under the basic NMT fine-tuning startegy. Further, we compare LLMs to SOTA APR works to know whether LLMs are superior.
- **RQ5: How do different PEFT options affect LLMs' repair capability?** RQ5 investigates the impact of different PEFT techniques and FPFT strategy on LLMs' repair capability, which can help us fine-tune billion-level LLMs with limited memory resources and provide guidance on choosing appropriate PEFT strategies for LLM4APR research.
- **RQ6: How well do recent repair strategies work on the LLM basis?** RQ6 aims to reveal the generalization of various novel repair strategies proposed in recent APR work in the era of LLMs. This helps us understand the potential of different repair strategies in enhancing the repair capability of LLMs and reveals the shortcomings of previous repair techniques in the era of LLMs.

B. Studied LLMs (LLM4APR)

Considering that program repair is a code-related task, we focus only on the Large Language Model of Code (or Big Code Model [77]).

In the initial study, we follow the following criteria for selecting LLMs. First, given that our computing resource (an RTX 3090 GPU, 24G), which means that the model size is at the million level. Also, the model parameters of all LLMs should be of similar size for a fairer comparison of the repair capability. Second, the pre-trained model and its data should be open-sourced, which allows for fine-tuning models and analyzing the pre-training data (e.g., data leak). Third, we do not choose decoder-only million-level models (e.g., GPT-C [78] and CodeGPT [79]), as a recent study [58] suggests that such LLMs perform more worse than encoder-only and encoder-decoder LLMs on bug fixing tasks. Finally, we choose 5 models: CodeBERT [80], GraphCodeBERT (GCodeBERT) [81], PLBART [82], CodeT5 [83], and UniXcoder [84]. In the specific experiments, we use the base model of above 5 million-level LLMs (i.e., CodeBERT-base, GraphCodeBERT-base, PLBART-base, CodeT5-base, UniXcoder-base) for the experiments.

In the extend study, we have access to sufficient computing resources (a TESLA A100 GPU, 80G) and the emergence of PEFT techniques alleviates the computing resource limitation, so we focus on billion-level LLMs. For model selection, we chose Top-5 model families from the Big Code Models Leaderboard [77], i.e., CodeLlama, StarCoder, Flacon, CodeGeeX, and CodeGen. In particular, considering that Flacon's model size is too large (180B), which would be costly for fine-tuning, we did not choose it for our experiments. In addition, we also chose InCoder, the powerful LLM in the previous study [13]. Finally, we choose 5 model families: CodeLlama [85], StarCoder [86], CodeGeeX [87], CodeGen [88], and InCoder [89]. Note that billion-level LLMs usually provide variants of multiple sizes. Considering the fine-tuning cost of billion-level LLMs, we use StarCoder(2)-15B, CodeLlama-13B/34B/70B in our experiments. For the remaining 3 LLM families which are slightly smaller in size, we use their strongest variants on the Big Code Models Leaderboard, i.e., CodeGeeX2-6B, CodeGen25-7B, and InCoder-6B.

In summary, we selected 10 LLM families for our experiments. More details of models are shown in Table I.

C. Studied Fine-Tuning Techniques (PEFT4LLM)

In our early initial study, we focused only on million-level LLMs, which can be fine-tuned and deployed on consumer-grade GPUs. Today, the size of LLMs has risen to the billion-level. The traditional paradigm is to fine-tune all of a model's parameters for each downstream task, but this is becoming exceedingly costly and impractical because of the enormous number of parameters in models today. PEFT techniques [90] only fine-tune a small number of (extra) model parameters - significantly decreasing computational and storage costs - while yielding performance comparable to a fully fine-tuned model [91]. In our extended study, we added five larger billion-level LLMs, so we will focus on exploring the impact of PEFT techniques on LLM4APR research, in order to provide guidance on the selection of PEFT techniques for APR research in the era of LLMs. In the specific experiments, considering the convenience and reproducibility of the experiments, we selected the PEFT methods to be studied from the popular PEFT libraries [91]. The PEFT library provides 3 types of methods, Prompt-based [92] (e.g., Prompt tuning [93], Prefix tuning [94], P-tuning [95], etc.), Adapter-based [96] (e.g, LoRA [51], LoHa [97], LoKr

[98], etc.), and IA3 [99] methods. Given that recent studies have shown that prompt-based methods do not have advantages over other types of techniques [100], we will not consider it in our experiments. Furthermore, given that our experiments are to be conducted on 5 different model families, we need to ensure that the chosen PEFT technique can be deployed on all models. After the initial experimental deployment, we finally choose 3 generalized PEFT techniques (which can be successfully run on the experimental models in this paper), namely LoRA [51], AdaLoRA [52], and IA3 [53]. In addition, we also add the FPFT strategy to compare it with PEFT techniques.

In summary, we selected 4 fine-tuning strategies for our experiments. More details are shown below.

- **FPFT.** Full-Parameter Fine-Tuning (FPFT) is a general paradigm applied to fine-tuning workflows, which tunings and deploys pre-trained models to various downstream tasks by adjusting all model parameters. Although tuning all parameters requires more costs, yet this usually allows the model to learn and gain the ability to handle specific downstream tasks better. To the best of our knowledge, most current fine-tuning based LLM4APR studies [12], [13], [14], [15] use the FPFT strategy.
- **LoRA.** Low-Rank Adaptation (LoRA) [51] is a adapter-based method, which accelerates fine-tuning LLMs while consuming less memory. It usually adds extra trainable parameters after the attention and fully-connected layers of a frozen pre-trained model to reduce memory-usage and speed up training [91]. To the best of our knowledge, LoRA is one of the most popular PEFT methods and has been successfully applied to the fine-tuned implementation of LLM-based APR tool [26].
- **AdaLoRA.** Adaptive Low-Rank Adaptation (AdaLoRA) [52] is an evolutionary variant of LoRA, which manages the parameter budget introduced from LoRA by allocating more parameters. Specifically, it will assign higher `rank` for important weight `matrices` that are better adapted for a task and pruning less important ones [91]. In the original AdaLoRA paper [52], it has been experimented on natural language processing tasks and demonstrated better results than baselines such as LoRA.
- **IA3.** IA3 [53] is also a PEFT technique aimed to improve LoRA by tuning fewer parameters for better performance. Unlike LoRA, which learns low-rank updates to weight `matrices`, IA3 scales activations by learned `vectors`, attaining stronger performance while only introducing a relatively tiny amount of new parameters. IA3 introduces an even smaller number of trainable parameters than LoRA. As a result, it is faster, cheaper and more efficient to fine-tune for a new downstream task [91].

D. Studied Repair Strategies (APR4LLM)

In learning-based APR techniques [12], researchers have proposed various novel repair strategies that are deployed for implementation on non-pre-trained or pre-trained models. However, it is still unclear about the effectiveness of those APR techniques deployed on non-pre-trained models in the era of LLM. Therefore, we would like to investigate the potential of these repair strategies in enhancing the repair capability of LLMs through further experiments. In the selection of repair strategies, we focus on APR work that is domain-general, model-independent, and open-science. Based on the above requirements, we selected the 3 novel repair strategies (KATANA [54], TENURE [49], ITER [45]) from the recent APR living review [6]. In addition, we also used the basic NMT fine-tuning strategy as the baseline.

In summary, we selected 4 repair strategies for our experiments. More details are shown below.

- **NMT.** Back in 2018, Tufano et al. [7] brought the NMT workflow to bug fixing tasks and it became one of the common paradigms for later learning-based APR work. In particular, recent LLM4APR studies [12], [13], [14] have also followed the NMT fine-tuning paradigm. Basic NMT fine-tuning typically takes the bug code along with the context as input, and the output of the model is the fix code. The model learns the transformation from bug code to fix code through fine-tuning, which is similar to the natural language translation task. Here, we will use the basic NMT paradigm [7] to explore the basic repair capabilities of LLMs and to serve as a basic baseline.
- **TENURE.** In ICSE 2023, Meng et al.'s work TENURE [49] combines template-based and learning-based technical paths. Unlike traditional template-based techniques [48], [101] that regard the repair process as a two-stage process (template selection and patch generation), which plainly use all templates and synthesize patches heuristically, TENURE is a one-stage strategy that allows the model to predict both templates and patches by constructing an NMT model. The insight behind this is that patch synthesis can be greatly guided by repair templates to explore the patch space more efficiently. Similar template-guided strategies have also been applied to LLM-based work (e.g, AlphaRepair [17] and GAMMA [30].), which are not considered in this study given that they employ the prompt learning paradigm and rely on LLMs to support the infilling task.
- **ITER.** In ICSE 2024, Ye et al. [45] propose an iterative program repair paradigm called ITER founded on the concept of improving partial patches until they become plausible and correct. Specifically, this iterative repair strategy can backfill previously generated patches into the bug code and continue iterating on top of the previous patches in order to synthesize patches. In other words, ITER's novelty is its ability to optimize in-depth for some patches that have the potential for successful fixes through multiple iterations, which differs from most APR works that synthesize all fixes at once [33], [38], [39], [40], [41], [42], [43], [44], [46], [47], [48], [49], [56], [57]. ITER's iterative strategy can further extend the patch space, which may be able to alleviate the patch space limitation [15] of LLM4APR.
- **KATANA.** Recently, Sintaha et al. [54] introduce the concept of program slicing into the APR domain and present the repair tool KATANA. An innovation of KATANA is

TABLE II
DATASETS, BASELINES, AND PARAMETER SETTINGS FOR LLM4APR STUDY. (I.O.: MAX INPUT/OUTPUT LENGTH; L.R.: LEARNING RATE; T.E.: TRAINING EPOCH; B.W.: BEAM WIDTH; P.N.: PATCH NUMBER)

LLM4APR Study	Repair Task		Train Dataset			Test Benchmark			Language	Defect Hunk		Baseline	Parameter Setting				
	Defect	Task	Dataset	#Bugs	#BFPs	Dataset	#Bugs	#BFPs		Single	Multi		I.O.	L.R.	T.E.	B.W.	P.N.
Initial Study (Million-level LLMs)	Bug	❶	BFP_S	-	52,515	BFP_S	-	5,835	Java	YES	YES	Tufano et al.	512	5e-5	30	5	1
			BFP_M	-	58,909	BFP_M	-	6,546	Java	YES	YES	Tufano et al.	512	5e-5	30	5	1
		❷	SeqRD	-	35,551	SeqRD	-	4,707	Java	YES	-	Chen et al.	512	5e-5	30	50	50
		❸	RecD	-	143,666	D4J V1.2	395	560	Java	YES	-	Jiang et al.	512	5e-5	30	100	10
			RecD	-	143,666	D4J V2.0	444	719	Java	YES	-	Jiang et al.	512	5e-5	30	100	10
		❹	CPMD	44,154	80,501	D4J V1.2	395	560	Java	YES	YES	Li et al.	512	1e-4	30	100	100
	Vul.	❺	VulRD	-	6,776	VulRD	-	1,706	C/C++	YES	YES	Fu et al.	512	2e-5	75	50	50
	Error	❻	TFixD	-	94,300	TFixD	-	10,504	JavaScript	YES	-	Berabi et al.	512	2e-5	30	5	1
Extended Study (Billion-level LLMs)	Bug	❼	TraD	-	101,475	D4J V1.2	395	560	Java	YES	YES	Xia et al.	2048	5e-5	1	10	10
			TraD	-	101,475	HEV	163	163	Java	YES	-	Jiang et al.	2048	5e-5	1	10	10

in the data pre-processing phase, where it proposes a concept called double slicing to help extract the context of the buggy code. Specifically, KATANA helps to extract the contexts that have data dependencies and control dependencies with the bug code to provide sufficient repair ingredients for guiding bug fixes and to further reduce the redundancy of contexts. The original implementation of KATANA worked on GNN, and given its novel design in the data pre-processing, we believe that KATANA's strategy has the same potential for LLMs.

IV. LLM4APR Study

A. Study Setup

In the LLM4APR study, we aimed to explore the repair capabilities of LLMs on APR tasks. In the initial study (Task ❶-❻), we focus on the repair ability of 5 million-level LLMs under FPFT strategy, and we conduct experiments on a 24G RTX 3090 GPU. In the extended study (Task ❼), we additionally explored the repair ability of 5 billion-level LLMs, and we conduct experiments on a 80G TESLA A100 GPU. Here, we use 8-Bit QLoRA [102] technique for billion-level LLMs to save memory without sacrificing performance. Besides, we fine-tune only one epoch based on the experience of previous work [13]. As shown in Table II, we present the details of the train datasets, test benchmarks, and parameter settings used in the LLM4APR study. More details are shown below.

- **Task ❶.** We use the **BFP dataset** including the small and medium versions (BFP_S and BFP_M) provided by Tufanol et al. [7] for training and testing, and take their work as baselines.
- **Task ❷.** We use the **SequenceR dataset** (SeqRD) provided by Chen et al. [38] for training and testing, and take their work as baselines.
- **Task ❸.** For the single-hunk bug repair, we use the **Recoder dataset** (RecD) provided by Jiang et al. [13] for model training, and the testing is on **Defects4J** [50] (D4J) V1.2 and 2.0.
- **Task ❹.** For the multi-hunk bug repair, we use the **CPatMiner dataset** (CPMD) provided by Li et al. [31] for training, and test them on the Defects4J V1.2. As Li et al. only provided repair results on Defects4J V1.2, we keep consistent with them to avoid bias on V2.0.
- **Task ❺.** We use VulRepair by Fu et al. [24] as the baseline and their **VulRepair dataset** (VulRD) for model training and testing, which include Big-Vul [103] and CVEfixes [104].
- **Task ❻.** We use TFix by Berabi et al. [32] as the baseline and their publicly available **TFix dataset** (TFixD) for training and testing.
- **Task ❼.** We use the **Transfer dataset** (TraD) provided by Meng et al. [48], [49] as the traing dataset and the Defects4J V1.2 and **HumanEval-Java** [13] (HEV) as test benchmarks. In particular, considering the cost of fine-tuning billion-level LLMs, we randomly select 100K+ samples from the Transfer dataset. We selected recent ChatGPT-based APR work [19], [28] as baselines.

B. Empirical Results

1) Initial Study: Table III shows the repair results of 5 million-level LLMs in different repair tasks: **1) In Task ❶**, we first study the impact of different design choices for bug repair on the BFP dataset [7] to provide guidance for subsequent experiments. These choices include different code abstraction strategies (*abs*/*raw*), code representation forms (*CR1*/*CR2*/*CR3*), and model evaluation metrics for selecting checkpoints (*PPL*/*BLEU*/*Last*). **2) In Task ❷**, we follow the insights gained from Task ❶ and use the best combination of the code representation (*CR3*) + without code abstraction (*raw*) to perform the experiments on the SequenceR dataset [38]. **3) In Task ❸**, we evaluate LLMs for single-hunk bugs on Defects4J V1.2 and V2.0 [50]. **4) In Task ❹**, we evaluate the multi-hunk bugs on Defects4J V1.2. **5) In Task ❺**, we explore the vulnerability repair capability of LLMs on the VulRepair dataset. The repair results using our previously Task ❶ obtained best representation (*CR3*) and VRepair's representation (*CR4*) [33]. **6) In Task ❻**, we explore the single-hunk error repair capability of LLMs on the TFix dataset.

2) Extended Study: Table IV shows the repair results of 5 billion-level LLM families in bug repair tasks. Specifically, we additionally add the recently released CodeLlama-70B [70] and StarCoder2 [105]. **In Task ❼**, we explore the repair capability of LLMs on the Defects4J V1.2 and HumanEval-Java.

C. Research Questions

1) RQ1: **How do different design choices affect LLMs' repair capability?**

TABLE III
REPAIR RESULTS OF MILLION-LEVEL LLMS IN TASK ❶-❻. (X/Y: CORRECT FIXES/PLAUSIBLE FIXES; Z%: REPAIR ACCURACY)

Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❶	BFP_S	CR1_abs	12.94%	**16.90%**	**16.90%**	13.32%	17.55%	**17.62%**	17.96%	8.86%	**19.40%**	20.15%	**21.80%**	21.37%	**19.47%**	18.89%	19.20%
		CR1_raw	12.17%	16.30%	**17.48%**	14.81%	17.26%	**17.46%**	14.41%	17.86%	**17.93%**	19.69%	22.45%	**25.18%**	18.44%	**23.79%**	**23.79%**
		CR2_raw	**34.28%**	**34.28%**	**34.28%**	31.11%	33.44%	**33.52%**	31.65%	34.14%	**34.52%**	43.27%	**47.82%**	47.80%	36.71%	41.70%	**42.33%**
		CR3_raw	34.82%	**35.25%**	34.82%	36.49%	**39.32%**	**39.32%**	33.13%	34.57%	**36.00%**	42.71%	**47.66%**	47.34%	42.67%	**45.02%**	**45.02%**
Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❶	BFP_M	CR1_abs	4.72%	3.16%	**9.73%**	3.96%	3.96%	**10.36%**	**13.17%**	0.31%	**13.17%**	8.27%	**13.95%**	13.92%	5.39%	2.55%	**8.08%**
		CR1_raw	4.22%	**9.41%**	**9.41%**	5.27%	**10.45%**	**10.45%**	4.58%	0.69%	**10.13%**	8.72%	**16.03%**	**16.03%**	6.59%	11.38%	**11.64%**
		CR2_raw	26.11%	**31.75%**	31.61%	27.56%	30.88%	**30.96%**	23.47%	**27.96%**	**27.96%**	35.77%	**41.88%**	**41.88%**	32.21%	**36.85%**	**36.85%**
		CR3_raw	28.40%	**33.48%**	**33.48%**	29.01%	33.35%	**33.43%**	27.04%	30.39%	**30.74%**	36.24%	**42.12%**	**42.12%**	36.29%	39.53%	**39.65%**
Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❷	SeqRD	CR3_raw	**19.06%**	14.40%	14.49%	**19.35%**	14.74%	14.53%	**17.87%**	13.13%	12.96%	**36.22%**	26.20%	26.03%	**33.91%**	22.33%	22.24%
Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❸	D4J V1.2	CR3_raw	21/37	**25/35**	23/31	24/39	26/35	**26/37**	12/24	**16/23**	**16/23**	**41/53**	30/43	30/43	**46/63**	36/46	38/45
	D4J V2.0	CR3_raw	22/38	12/26	**23/26**	**23/43**	22/38	22/37	**23/39**	15/27	15/27	**30/39**	16/26	16/26	**37/55**	20/33	22/35
	D4J V1.2	CR3_raw	**7.76%**	6.68%	6.14%	**8.84%**	6.86%	7.04%	4.51%	**4.69%**	**4.69%**	**11.01%**	8.84%	8.84%	**11.73%**	8.48%	9.21%
	D4J V2.0	CR3_raw	**5.84%**	2.78%	3.20%	**5.70%**	4.03%	4.03%	**6.12%**	4.59%	4.59%	**8.34%**	6.12%	6.12%	**8.84%**	5.70%	5.84%
Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❹	D4J V1.2	CR3_raw	**40/54**	34/48	31/46	**41/61**	39/53	39/53	**32/52**	25/39	25/41	**69/95**	53/77	53/77	**66/102**	52/76	54/82
	D4J V1.2	CR3_raw	**11.19%**	10.47%	10.83%	**10.11%**	8.84%	8.84%	**10.29%**	7.94%	7.76%	**17.69%**	12.64%	12.64%	**18.41%**	13.36%	14.08%
Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❺	VulRD	CR3_raw	43.96%	**51.58%**	**51.58%**	43.61%	**53.28%**	53.22%	47.30%	59.85%	**60.08%**	52.93%	**62.78%**	**62.78%**	50.64%	62.08%	**62.19%**
		CR4_raw	31.07%	49.00%	**49.24%**	25.91%	41.85%	**41.97%**	46.19%	53.58%	**53.93%**	35.87%	55.86%	**55.92%**	40.62%	55.28%	**55.63%**
Task	Benckmark	Code Rep.+ Abs.	CodeBERT			GraphCodeBERT			PLBART			CodeT5			UniXcoder		
			PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last	PPL	BLEU	Last
❻	TFixD	CR3_raw	48.28%	52.39%	**52.83%**	48.53%	**52.33%**	**52.33%**	44.78%	**48.23%**	**48.23%**	50.44%	54.93%	**55.31%**	48.25%	**54.33%**	54.31%

TABLE IV
REPAIR RESULTS OF BILLION-LEVEL LLMS IN TASK ❼. (X/Y: CORRECT FIXES/PLAUSIBLE FIXES)

Task	Benchmark	Code Rep. + Abs.	InCoder-6B	CodeGeeX-6B	CodeGen25-7B	CodeLlama-13B	StarCoder-15B	StarCoder2-15B	CodeLlama-70B
❼	D4J V1.2	CR3_raw	74/93	75/93	87/111	93/112	101/122	95/124	117/140
	HEV	CR3_raw	78/89	86/94	105/119	118/128	108/121	134/141	124/134

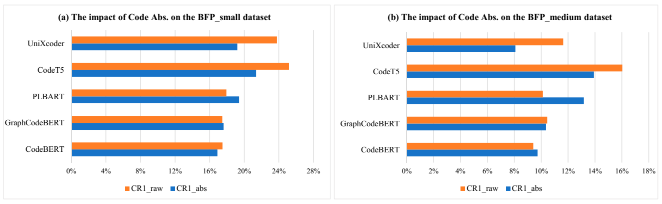


Fig. 4. The impact of Code Abs. on the BFP dataset.

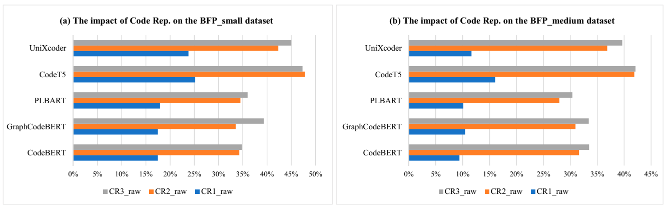


Fig. 5. The impact of Code Rep. on the BFP dataset.

a) Code Abstraction: We first analyze the impact of code abstraction. From Table III (Task ❶), we observe that using code abstraction (*CR1_abs*) and without code abstraction (*CR1_raw*) have a more or less impact on the repair results. To present a more intuitive picture of the impact of code abstraction, we extract the *Last model* repair results for each LLM for comparison (*Last model* is irrelevant to evaluation metrics). As shown in Fig. 4, for most LLMs, the potential impact of code abstraction on the repair capability may be limited or negligible. For example, CodeT5 and UniXcoder achieved the best repair results using *raw* code input, while CodeBERT and GraphCodeBERT have close results on *abs* and *raw*. This suggests that *raw* code is already adequate, and it is unnecessary to use code abstraction for LLMs.

There are two main reasons for the phenomenon. First, since LLMs are usually pre-trained on *raw* source code, they are better suited to the same unprocessed *raw* data for downstream tasks. Second, as Chen et al. argue, code abstraction may lose some semantic information (e.g., special function and variable names) [38], which makes it difficult to learn fix patterns.

Finding 1: Code abstraction does not significantly improve the repair capability of most million-level LLMs. Using the data format without code abstraction processing (i.e., *raw* source code) is more suitable for fine-tuning LLMs.

b) Code Representation: In order to investigate the impact of different code representations, we again compare LLMs with the *Last model* according to the results from Table III (Task ❶). We present the impact of the different CRs on the repair results in Fig. 5. As shown in Fig. 5, the use of *CR2* with fault location and repair location information outperforms the repair results of *CR1*. In addition, we observe that *CR3*, a representation that removes the repair code context information, has a slightly better repair effect than *CR2*. Combining these results, we conclude that *CR3* is a more suitable code representation for the repair task.

TABLE V
RESULTS OF DIFFERENT DESIGN CHOICES ON TASK ❺

Model + Design Choices	Repair Accuracy
CodeT5 + CR4_raw + LOSS (VulRepair)	44.67%
CodeT5 + CR4_raw + PPL	35.87% (-8.8%)
CodeT5 + CR4_raw + BLEU	55.86% (+11.19%)
CodeT5 + CR4_raw + Last	55.92% (+11.25%)
CodeT5 + CR4_raw + Ensemble Learning	57.33% (+12.66%)
CodeT5 + CR3_raw + LOSS	52.35% (+7.68%)
CodeT5 + CR3_raw + PPL	52.93% (+8.26%)
CodeT5 + CR3_raw + BLEU	62.78% (+18.11%)
CodeT5 + CR3_raw + Last	62.78% (+18.11%)
CodeT5 + CR3_raw + Ensemble Learning	64.71% (+20.04%)

We now analyze why *CR3* can work more effectively on LLMs. First, using special tokens to mark fault/repair locations enables the model to focus on code repair behaviors for targeted learning. Second, LLMs suffer from the long sequence problem [24]. As the length of the input/output grows, the repair accuracy of LLMs decreases. Removing irrelevant context from the output sequence is equivalent to reducing the output length, so the model's repair capability may be improved.

Finding 2: The fine-tuning of million-level LLMs for APR can be improved by delicate code representations. More accurate fault location, precise repair location information, and removing the contextual code of the fixes are all beneficial for improvement.

Besides, in Task ❺, we further explore the vulnerability repair capability of LLMs on the VulRepair dataset. Table III shows the repair results using our previously obtained best representation (*CR3*) and VRepair's representation (*CR4*) [33]. We conduct ablation study on CodeT5 used in VulRepair to explore the impact of different design choices.

As shown in Table V, the repair results using *CR3* are all better than those using *CR4*. Obviously, *CR3* is a more useful representation. This result again supports our **Finding 2**. This is because the two representations differ in the complexity of marking repair behaviors. As shown in Fig. 2, in *CR3*, all repair behaviors are seen as replace operations. In *CR4*, three distinct marks are used to represent add, delete, and replace actions, providing finer token-level fix locations. However, the complexity of this strategy might impede the model in understanding the different repair actions and accurately implementing fixes at precise locations. As a result, the model's repair capability could be compromised. In fact, all repair actions can be simplified to replacement operations (i.e., the fix replaces the bug location). Although *CR3* uses a coarse-grained markup approach, it simplifies the repair operation, thereby enhancing the model's repair capability.

Finding 3: Using finer-grained code representations is not conducive to fully exploiting the repair capability of million-level LLMs, and CR3 remains the best representation on vulnerability repair.

c) Checkpoint Selection: We track the impact of different model evaluation metrics for checkpoint selection on the results under the best input/output format *CR3_raw*. In Table III (Task ❶ ❺ ❻), on the BFP/VulRepair/TFix dataset, the *best BLEU model* and the *Last model* tend to achieve higher repair accuracy than the *best PPL model*. However, the contrary result is obtained on the SequenceR dataset and Defects4J. In Table III (Task ❷-❹), the *best PPL model* achieves the best repair accuracy.

This motivates us to explore further. First, we noticed that in the repair scenario with *BLEU* as the best metric (Task ❶ ❺ ❻), the train/val/test datasets were obtained from a random split on the BFP dataset. Similarly, VulRepair and TFix datasets are divided by a randomized splitting strategy. Thus the train/val/test datasets hold similar data characteristics. Second, we found that in the repair scenarios where *PPL* was the best metric (Task ❷-❹), the train/val/test datasets were not split from the same dataset. For example, on Task ❷, the training data of the SequenceR dataset consists of CodRep 1/2/3/5 [106] and the BFP dataset [7], whereas the test data comes from CodRep 4 [106]. Similarly, on Task ❸ and Task ❹, the training data (Recoder dataset and CPatMiner dataset) do not contain the software projects from Defects4J. We can conclude that *BLEU* aligns better with training data, while *PPL* shows better generalization. When data characteristics are alike, *BLEU* is better; otherwise, *PPL* is preferred. However, it is hard to know the difference between the training and testing samples in practice. Therefore, we follow the practice of previous works [18], [39] to use the ensemble strategy by combining multiple checkpoints (*PPL*/*BLEU*/*Last*) to enhance the repair capability.

Finding 4: Different repair scenarios may have different best evaluation metrics for checkpoint selection. In practice, using ensemble learning is an appropriate strategy.

2) RQ2 & 4: **How well does the million & billion-level LLM perform compared to the state-of-the-art approaches?**

a) Repair Result: Based on findings obtained from RQ1, we use the *CR3_raw* and the ensemble strategy to obtain the best performance of million-level LLMs and compare them with baselines. Besides, we also use *CR3_raw* and the basic NMT fine-tuning strategy to compare billion-level LLMs with recent APR works.

Task ❶. As shown in Table VI, on the *BFP dataset*, LLMs improve over the baseline Tufano et al. [7] as follows: 1) BFP_S: CodeT5 (+43.82%) > UniXcoder (+41.16%) > GraphCodeBERT (+34.12%) > PLBART (+33.30%) > CodeBERT (+26.27%). 2) BFP_M: CodeT5 (+43.88%) > UniXcoder (+42.65%) > GraphCodeBERT (+34.43%) > CodeBERT (+34.13%) > PLBART (+32.98%).

Task ❷. As shown in Table VII, on the *SequenceR dataset*, all LLMs' results outperform SequenceR [38]: CodeT5 (+17.93%) > UniXcoder (+16.45%) > GraphCodeBERT (+2.28%) > CodeBERT (+1.74%) > PLBART (+0.13%).

Task ❸. As shown in Table VIII, our best results on Defects4J outperform previous APR tools [40], [41], [44], [46] and the recent study [13]. On Defects4J V1.2 and V2.0, our CodeT5-base and PLBART-base fixed 35 and 2 more bugs than Jiang et al. [13] when using the same model fine-tuning. Notably, our small-scale LLMs UniXcoder/CodeT5 outperform large-scale LLM InCoder-6B used by Jiang et al. Compared to

TABLE VI
BEST REPAIR RESULTS FOR LLMS ON TASK ❶

Benchmark	CodeBERT	GCodeBERT	PLBART	CodeT5	UniXcoder	Tufano et al. [7]
BFP_S	36.59%	44.44%	43.62%	54.14%	51.48%	10.32%
BFP_M	37.74%	38.04%	36.59%	47.49%	46.26%	3.61%

TABLE VII
BEST REPAIR RESULTS FOR LLMS ON TASK ❷

Benchmark	CodeBERT	GCodeBERT	PLBART	CodeT5	UniXcoder	SequenceR [38]
SeqRD	21.92%	22.46%	20.31%	38.11%	36.63%	20.18%

TABLE VIII
BEST REPAIR RESULTS FOR LLMS ON TASK ❸

Benchmark	**Our Work**					**Baseline (Jiang et al. [13])**										
	CodeBERT	GCodeBERT	PLBART	CodeT5	UniXcoder	PLBART	CodeT5	CodeGen			InCoder		DL-based APR Tool			
	base	base	base	base	base	base	base	350M	2B	6B	1B	6B	CURE	Reward	Recoder	KNOD
D4J V1.2	33/47	34/47	16/31	49/62	**57/71**	25/-	30/-	23/-	32/-	38/-	27/-	41/-	6/-	20/-	24/-	20/-
D4J V2.0	25/44	28/48	24/41	33/45	**46/67**	13/-	17/-	20/-	23/-	23/-	24/-	28/-	6/-	8/-	11/-	13/-

TABLE IX
BEST REPAIR RESULTS FOR LLMS ON TASK ❹

Bug Type	**Our Work**					**Baseline (Li et al. [31])**			
	CodeBERT	GCodeBERT	PLBART	CodeT5	UniXcoder	DEAR	CURE	CoCoNut	DLFix
1. One-Hunk of One-Stmt	21/27	24/31	16/25	42/54	42/52	33/-	38/-	37/-	35/-
2. One-Hunk of Multi-Stmts	8/10	7/11	7/9	11/14	11/16	4/-	3/-	3/-	1/-
3. Multi-Hunks of One-Stmt	10/11	9/14	9/13	12/16	11/18	13/-	6/-	3/-	4/-
4. Multi-Hunks of Multi-Stmts	6/6	4/5	4/4	5/5	6/8	1/-	0/-	0/-	0/-
5. Multi-Hunks of Mix-Stmts	7/8	3/6	3/7	7/17	8/22	2/-	1/-	1/-	0/-
Total	**52/62**	**47/67**	**39/58**	**77/106**	**78/117**	**53/-**	**48/-**	**44/-**	**40/-**

their best model InCoder-6B, our results are as follows: 1) Defects4J V1.2: UniXcoder (+16) > CodeT5 (+8) > GraphCodeBERT (-7) > CodeBERT (-8) > PLBART (-25). 2) Defects4J V2.0: UniXcoder (+18) > CodeT5 (+5) > GraphCodeBERT (+0) > CodeBERT (-3) > PLBART (-4).

Task ❹. As shown in Table IX, our best repair results on Defects4J V1.2 outperform existing works [31], [39], [40], [42]. LLMs improve over the best multi-hunk APR tool DEAR [31] as follows: UniXcoder (+25) > CodeT5 (+24) > CodeBERT (-1) > GraphCodeBERT (-6) > PLBART (-14).

Task ❺. As shown in Table X, on the VulRepair dataset, all LLMs outperform VulRepair [24] and VRepair [33]. In particular, LLMs improve over the best baseline VulRepair [24] as follows: CodeT5 (+20.04%) > UniXcoder (+19.10%) > PLBART (+16.23%) > GraphCodeBERT (+9.49%) > CodeBERT (+7.50%).

Task ❻. As shown in Table XI, on the TFix dataset, LLMs improve over the best baseline TFix [32] (T5-large) as follows: CodeT5 (+11.32%) > UniXcoder (+10.80%) > CodeBERT (+8.12%) > GraphCodeBERT (+7.15%) > PLBART (+3.98%).

Task ❼. In this task, we fine-tuned five million-level LLMs to evaluate and compare their repair performance with that of billion-level models. As presented in Table XII, the larger billion-level LLMs demonstrate significantly superior repair capabilities compared to the million-level models. Besides, fine-tuned billion-level LLMs using smaller patch spaces (10 patches) rival or even surpass recent ChatGPT-based APR work [19], [28]. On the Defects4J V1.2, CodeLlama-70B fixed 3 more bugs than ChatRepair. [19]. On the HumanEval-Java, StarCoder2-15B fixed 134 bugs, which is close to the results of the best baseline ContrastRepair [28]. Notably, we use far less patch space than these ChatGPT-based baseline works. This suggests that LLMs can use fewer candidate patches after fine-tuning comparable to APR tools based on ChatGPT in the prompt paradigm. Compared to the best baseline, our results are as follows: 1) Defects4J V1.2: CodeLlama-70B (+3) > StarCoder-15B (-13) > StarCoder2-15B (-19) > CodeLlama-13B (-19) > CodeGen25-7B (-26) > CodeGeeX2-6B (-39) > InCoder-6B (-40). 2) HumanEval-Java: StarCoder2-15B (-3) > CodeLlama-70B (-13) > CodeLlama-13B (-19) > StarCoder-15B (-29) > CodeGen25-7B (-32) > CodeGeeX2-6B (-51) > InCoder-6B (-59).

According to the results from Task ❶-❻, we find that using the million-level LLMs UniXcoder and CodeT5 has surpassed previous works on bug/vulnerability/error repair tasks. This demonstrates that fine-tuning million-level LLMs has great

TABLE X
BEST REPAIR RESULTS FOR LLMS ON TASK ❺

Benchmark	CodeBERT	GCodeBERT	PLBART	CodeT5	UniXcoder	VulRepair [24]	VRepair [33]
VulRD	52.17%	54.16%	60.90%	64.71%	63.77%	44.67%	23.00%

TABLE XI
BEST REPAIR RESULTS FOR LLMS ON TASK ❻

Benchmark	CodeBERT	GraphCode	PLBART	CodeT5	UniXcoder	TFix [32]	CoCoNuT [39]	SequenceR [38]
TFixD	57.42%	56.45%	53.28%	60.62%	60.10%	49.30%	11.70%	17.90%

TABLE XII
REPAIR RESULTS FOR LLMS ON TASK ❼

LLM	Million-Level LLMs					Billion-Level LLMs							Baseline [19], [28]	
	CodeBERT	GCodeBERT	PLBART	UniXcoder	CodeT5	InCoder-6B	CodeGeeX2-6B	CodeGen25-7B	CodeLlama-13B	StarCoder-15B	StarCoder2-15B	CodeLlama-70B	ContrastRepair	ChatRepair
Beam Size	10	10	10	10	10	10	10	10	10	10	10	10	-	-
Patch Size	10	10	10	10	10	10	10	10	10	10	10	10	120	500
Chat	3/5	1/4	6/8	9/13	10/12	11/14	12/14	11/14	12/14	12/15	12/16	12/15	12/-	15/-
Closure	11/14	5/9	13/16	18/22	20/24	22/29	19/23	26/30	27/32	34/37	28/34	34/43	32/-	37/-
Lang	7/10	5/8	5/10	12/16	12/17	13/19	14/19	15/23	19/25	16/23	15/25	25/31	19/-	21/-
Math	9/14	6/7	13/16	17/21	18/23	21/24	24/29	28/35	29/32	30/36	32/41	33/39	30/-	32/-
Mockito	2/2	2/3	3/3	4/5	4/5	4/4	4/5	5/6	5/5	6/7	5/5	7/7	8/-	6/-
Time	1/1	0/1	2/2	2/2	3/3	3/3	2/3	3/3	3/4	3/4	3/3	4/5	2/-	3/-
D4J V1.2	33/46	19/32	42/55	62/79	67/84	74/93	75/93	88/111	95/112	101/122	95/124	117/140	103/-	114/-
HEV	14/18	7/8	21/28	42/47	50/60	78/89	86/94	105/119	118/128	108/121	134/141	124/134	137/151	130/143

potential for APR research. In addition, in Task ❼, larger-scale billion-level LLMs have better repair capabilities than smaller-scale million-level LLMs, and large-scale models such as CodeLlama-70B and StarCoder2-15B rival or even surpass recent ChatGPT-based work in terms of repair capabilities.

Finding 5: Million-level LLMs showcase competitive repair capabilities in APR. Billion-level LLMs, with their larger scale, generally exhibit even stronger repair capabilities. Notably, fine-tuned open source billion-level LLMs achieve repair performance comparable to recent ChatGPT-based APR approaches.

b) Multi-Hunk: We also pay close attention to multi-hunk bug fix, since little research has been done on the repair of complex bugs. As mentioned earlier, we use *CR3* to extend the NMT workflow to multi-hunk repair scenarios.

Task ❹. The results are provided in Table IX. Obviously, compared to single-line bugs (Type 1), single-hunk (Type 2) and multi-hunk bugs (Type 3-5) are far more difficult to repair. This is because such bugs entail intricate dependencies from both inner and outer of one buggy method. Nonetheless, our work achieves a great improvement over existing approaches on fixing such complex bugs. In particular, UniXcoder and CodeT5 outperform the advanced multi-hunk APR tool DEAR and fixed 9 and 8 more multi-hunk bugs. Furthermore, to our surprise, there is little gap between the number of fixed single-hunk bugs and the number of multi-hunk bugs. We hope such results can encourage researchers to explore more advanced approaches for repairing complex bugs, as they are more common in real-world projects.

Finding 6: Fine-tuning million-level LLMs to fix multi-hunk bugs is also promising, though it is more difficult compared with single-line bug repair.

Task ❻. We also study the repair capability of LLMs for different vulnerability hunks. As shown in Table XIV, LLMs have the highest repair accuracy in single-hunk fixes. In the multi-hunk fixing scenario, when the number of hunks is not big (i.e., < 5), the accuracy is not significantly behind that of the single-hunk fixes. However, as the vulnerable hunks increase, the repair accuracy of LLMs drops sharply. To sum up, although we can obtain a fairly good results for repairing complex vulnerabilities, there is still a long way to go for overly complex ones.

Finding 7: The fine-tuned million-level LLMs show great potential for vulnerability repair. They can also deal with multi-hunk vulnerabilities to some extent unless the hunks are too many.

Task ❼. Our extended study also investigates the performance of billion-level LLMs in repairing multi-hunk bugs. Following the methodology of the recent multi-hunk repair work, ITER [45], we selected 33 multi-hunk bugs to evaluate the repair capabilities of the models. As shown in Table XIII, billion-level LLMs generally fix more multi-hunk bugs than million-level LLMs, highlighting their advantages in handling complex repair scenarios. Notably, the best billion-level model, StarCoder2 (14 correct fixes), performs comparably to ITER [45] (15 correct fixes) and surpasses the ChatGPT-based RepairAgent [29] (8 correct fixes). Since ITER does not rely on perfect fault localization, we reference it here to illustrate the potential of billion-level LLMs in multi-hunk repair scenarios, rather than as a basis for direct comparison.

Finding 8: Billion-level LLMs also show promise in multi-hunk repair scenarios, with larger-scale models offering significant advantages over smaller-scale million-level LLMs when addressing complex multi-hunk repairs.

TABLE XIII
MULTI-HUNK REPAIR RESULTS (33 MULTI-HUNK BUGS IN DEFECTS4J V1.2) FOR LLMS ON TASK ❼

Multi-hunk Bugs		**Million-level LLMs**					**Billion-level LLMs**							**Baseline [19], [28], [29], [45]**			
Bug ID	Edits	CodeBERT -base	GCodeBERT -base	PLBART -base	UniXcoder -base	CodeT5 -base	InCoder -6B	CodeGeeX2 -6B	CodeGen25 -7B	CodeLlama -13B	StarCoder -15B	StarCoder2 -15B	CodeLlama -70B	ITER	RepairAgent	ChatRepair	ContrastRepair
Chart-14	4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
Chart-16	3				✓		✓	✓	✓	✓	✓	✓		✓			
Closure-4	2													✓			
Closure-78	2						✓				✓	✓			✓	✓	✓
Closure-79	2										✓	✓		✓			
Closure-106	2																
Closure-109	2												✓				
Closure-115	2	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Closure-131	2													✓			
Lang-7	2																
Lang-27	2												✓			✓	✓
Lang-34	2	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			
Lang-35	2												✓				
Lang-41	4																
Lang-47	2	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓			
Lang-60	2								✓	✓	✓	✓	✓		✓		
Math-1	2													✓			
Math-4	2	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			
Math-22	2	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Math-24	2																
Math-35	2							✓	✓	✓		✓		✓			
Math-43	3													✓			
Math-46	2													✓	✓		
Math-62	3																
Math-65	2														✓		
Math-71	2																
Math-77	2		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓		
Math-79	2	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Math-88	3																
Math-90	2																
Math-93	4																
Math-98	2				✓	✓	✓	✓	✓	✓	✓	✓	✓				
Math-99	2													✓			
#Total(33 bugs)		7	3	6	10	8	11	11	12	11	13	14	12	15	8	3	3

TABLE XIV
PERFORMANCE OF LLMS WITH VULNERABILITY HUNKS

Vul. #Hunks	CodeBERT	GCodeBERT	PLBART	CodeT5	UniXcoder
1	60%	63%	71%	75%	74%
2	53%	55%	60%	64%	63%
3	56%	57%	62%	66%	65%
4	40%	42%	51%	54%	58%
5	25%	27%	31%	31%	33%
6	24%	24%	24%	27%	24%
7	17%	17%	22%	22%	17%
8	13%	13%	13%	13%	13%
9	7%	7%	7%	7%	7%
10+	5%	5%	5%	5%	5%

c) Data Leakage: Prior work AlphaRepair [17] uncovered the data leak issue when using LLMs for APR, that is, the overlap between the pre-training data (CSN, i.e.,CodeSearchNet) and the test benchmark (D4J, i.e., Defects4J). To reveal its impact, we follow AlphaRepair to analyze the LLMs' repair results with respect to the data leak. We searched for the exact match and identified 48 overlaps between D4J IDs and CSN. Then we checked if the patches generated by LLMs were identical to those present in D4J. In Task ❸, among the 48 patches, only Lang_43, Math_22, and Mockito_5 were identified, and a similar pattern was observed in Task ❹. The results imply that fine-tuned LLMs are minimally affected by data leakage. However, this also means that million-level LLMs may tend to lose certain pre-trained knowledge during the fine-tuning process, highlighting the presence of the catastrophic forgetting problem [107], [108] during this phase. Besides, we observe catastrophic forgetting in billion-level models as well. For example, in a previous study [13], there were 25 HumanEval bugs that could be directly fixed by CodeGen-6B, yet the fine-tuned CodeGen-6B fixes failed.

Finding 9: The LLM is minimally affected by data leakage after sufficient fine-tuning. However, this exposes the catastrophic forgetting problem of LLMs under the fine-tuning paradigm.

```
   // fix no-undef Undefined variable.
   NL.triggerMapMoveEnd();
-  NL.respondLast200(NL.json.MapWmsLayers.records.regular);
+  NL.respondLast200(fx.regular);
   var layers = NL.vw.MAP.getWmsLayers();
```

Fig. 6. An error-fix example of *no-undef*.

3) RQ3: What are the factors that limit the effectiveness of fine-tuning LLMs?

a) Lack of Repair Ingredients: One of the factors is the method-level BFPs and the limited input/output length. When using method-level BFPs, it is difficult for the model to synthesize correct patches based on the incomplete context if repair ingredients (e.g., method names, variable names, etc.) are outside of that method. In particular, we note that the TFix dataset only records two lines of code before and after the error line as context, and such limited contextual information is not sufficient. Taking the *no-undef* error type as an example (see Fig. 6), we observe that fixing this error requires sufficient context to substitute the undefined variable with the defined one. However, the TFix dataset solely furnishes context from the line preceding and following the error location. As a result, it might lack the essential repair components, potentially resulting in an incorrect patch. Besides, if a method exceeds the maximum input/output length, it is difficult to provide a complete method for the model. This may result in a lack of necessary contextual information to guide the repair.

Finding 10: Method-level BFPs and the limited model input/output lengths may miss the necessary contextual information to guide the repair, thus limiting the repair capability of LLMs.

b) Computing Resource and Model Size: The lack of computing resources and the overly-large model size can hinder LLMs from generating more candidate patches. For example,

TABLE XV
THE % PERFECT PREDICTIONS OF LLMS FOR ALL CWES ON THE VULREPAIR DATASET

	VulRepair Dataset			Repair Accuracy				
No.	CWE Type	Train Samples	Test Samples	CodeBERT	GCodeBERT	PLBART	UniXcoder	CodeT5
1	CWE-119	1390	423	37%	39%	53%	55%	56%
2	CWE-125	606	166	49%	50%	52%	54%	55%
3	CWE-20	458	136	60%	60%	68%	70%	71%
4	CWE-000	412	105	60%	59%	65%	67%	67%
5	CWE-476	254	67	63%	64%	61%	66%	66%
6	CWE-200	253	59	73%	73%	76%	76%	76%
7	CWE-264	250	68	72%	75%	76%	75%	76%
8	CWE-399	216	53	64%	68%	75%	74%	70%
9	CWE-362	197	58	47%	48%	53%	59%	66%
10	CWE-787	188	62	42%	42%	44%	47%	48%
11	CWE-190	179	61	46%	44%	54%	56%	64%
12	CWE-416	175	64	59%	59%	62%	67%	64%
13	CWE-189	158	38	66%	66%	71%	71%	71%
14	CWE-400	103	37	62%	68%	76%	76%	76%
15	CWE-284	102	35	40%	49%	71%	74%	77%
16	CWE-59	65	11	64%	64%	64%	64%	64%
17	CWE-310	52	17	76%	76%	76%	76%	76%
18	CWE-401	50	12	8%	33%	50%	33%	58%
19	CWE-835	48	10	70%	70%	80%	80%	80%
20	CWE-415	46	17	65%	65%	65%	65%	65%
21	CWE-772	45	10	80%	80%	80%	80%	80%
22	CWE-617	43	21	57%	57%	57%	62%	62%
23	CWE-19	42	10	60%	60%	60%	60%	60%
24	CWE-269	38	14	86%	86%	86%	86%	86%
25	CWE-17	37	8	88%	88%	88%	88%	88%
26	CWE-120	35	10	20%	20%	20%	30%	40%
27	CWE-285	24	5	40%	40%	40%	40%	40%
28	CWE-369	24	8	50%	50%	50%	50%	50%
29	CWE-254	23	4	50%	50%	75%	75%	75%
30	CWE-22	23	6	50%	50%	50%	50%	50%
31	CWE-295	23	6	17%	17%	17%	17%	17%
32	CWE-78	22	4	25%	25%	25%	25%	25%
33	CWE-404	22	9	78%	78%	78%	78%	89%
34	CWE-674	21	6	0%	0%	17%	33%	17%
35	CWE-552	19	4	25%	25%	25%	25%	50%
36	CWE-834	15	6	33%	33%	33%	50%	33%
37	CWE-287	15	4	75%	75%	75%	75%	75%
38	CWE-74	15	3	67%	33%	33%	33%	67%
39	CWE-908	13	2	100%	100%	100%	100%	100%
40	CWE-326	12	3	100%	100%	100%	100%	100%
41	CWE-134	10	2	50%	50%	50%	50%	50%
42	CWE-252	10	1	0%	0%	0%	0%	0%
43	CWE-436	9	2	100%	100%	100%	100%	100%
44	CWE-77	9	4	75%	75%	75%	75%	75%
45	CWE-611	9	2	0%	0%	0%	50%	0%
46	CWE-444	9	1	0%	0%	0%	0%	0%
47	CWE-388	9	0	0%	0%	0%	0%	0%
48	CWE-763	8	4	25%	25%	25%	25%	25%
49	CWE-352	8	1	0%	0%	0%	0%	0%
50	CWE-732	8	1	0%	0%	0%	0%	0%
51	CWE-79	7	1	100%	100%	100%	100%	100%
52	CWE-522	7	1	0%	0%	0%	0%	0%
53	CWE-191	7	2	50%	50%	50%	50%	50%
54	CWE-770	6	1	0%	0%	0%	0%	0%
55	CWE-704	5	2	0%	0%	0%	0%	0%
56	CWE-665	5	4	75%	75%	75%	75%	75%
57	CWE-89	5	4	75%	75%	100%	100%	100%
58	CWE-129	4	4	25%	25%	25%	75%	75%
59	CWE-862	4	1	0%	0%	0%	0%	0%
60	CWE-354	4	2	50%	50%	50%	50%	50%
61	CWE-755	4	2	50%	50%	50%	50%	50%
62	CWE-90	4	0	0%	0%	0%	0%	0%
63	CWE-255	4	2	50%	50%	50%	50%	50%
64	CWE-426	4	1	100%	100%	100%	100%	100%
65	CWE-824	4	0	0%	0%	0%	0%	0%
66	CWE-667	4	3	33%	33%	33%	67%	67%
67	CWE-863	4	2	50%	50%	50%	50%	50%
68	CWE-172	3	0	0%	0%	0%	0%	0%
69	CWE-273	3	0	0%	0%	0%	0%	0%
70	CWE-754	3	0	0%	0%	0%	0%	0%
71	CWE-94	3	0	0%	0%	0%	0%	0%
72	CWE-601	3	1	0%	0%	0%	0%	0%
73	CWE-532	3	0	0%	0%	0%	0%	0%
74	CWE-347	3	2	50%	50%	100%	100%	100%
75	CWE-345	2	1	0%	0%	0%	0%	0%
76	CWE-358	2	3	67%	67%	67%	67%	67%
77	CWE-276	2	0	0%	0%	0%	0%	0%
78	CWE-327	2	0	0%	0%	0%	0%	0%
79	CWE-1187	2	0	0%	0%	0%	0%	0%
80	CWE-494	2	1	0%	0%	0%	0%	0%
81	CWE-346	2	0	0%	0%	0%	0%	0%
82	CWE-682	2	1	100%	100%	100%	100%	100%
83	CWE-918	2	1	0%	0%	0%	0%	0%
84	CWE-320	1	1	100%	100%	100%	100%	100%
85	CWE-681	1	0	0%	0%	0%	0%	0%
86	CWE-203	1	0	0%	0%	0%	0%	0%
87	CWE-193	1	0	0%	0%	0%	0%	0%
88	CWE-16	1	0	0%	0%	0%	0%	0%
89	CWE-290	1	1	100%	100%	100%	100%	100%
90	CWE-639	1	0	0%	0%	0%	0%	0%
91	CWE-319	1	0	0%	0%	0%	0%	0%
92	CWE-88	1	1	100%	100%	100%	100%	100%
93	CWE-668	1	0	0%	0%	0%	0%	0%
94	CWE-672	1	0	0%	0%	0%	0%	0%
95	CWE-502	1	0	0%	0%	0%	0%	0%
96	CWE-434	1	0	0%	0%	0%	0%	0%
97	CWE-18	1	0	0%	0%	0%	0%	0%
98	CWE-706	0	1	0%	0%	0%	0%	0%

when we use 5 million-level LLMs to perform patch synthesis on Defects4J, we can only go up to a max beam size of 200 and generate 200 patches for each bug in a RTX 3090 GPU. Furthermore, we only set the max beam size is 10 for CodeLlama-70B in a TESLA A100 GPU. Unlike traditional deep learning (DL) models [39], [40], [41], [49], it is non-trivial to use a larger beam size and produce a larger patch space. Previous studies [7], [24] have confirmed that the size of patch space positively impacts the overall performance of DL models for APR. Therefore, such a limitation makes it difficult for further improvement. On the other hand, we find that larger model sizes bring better repair capabilities while also tend to take more memory cost and time cost. Briefly, when implementing fine-tuning and inference for larger-scale billion-level models, more memory resources tend to be required and longer training and inference times are needed.

c) Long-tailed Class Imbalance: Table XV presents the results of the million-level LLM's repair capabilities across various CWE types. It is important to note that the distribution of training samples is highly imbalanced across CWE types, as illustrated in Fig. 7. Specifically, while the training set includes 97 CWE types (No. 1–97), a significant portion of these, namely 55 types (No. 43–97) have fewer than 10 training samples. This long-tailed class imbalance [109] may restrict the model's ability to perform effectively on few-shot types, as the limited training data for these categories poses inherent challenges to achieving robust performance.

Finding 11: Addressing small sample sizes and class imbalance constitutes a primary challenge in vulnerability repair.

d) Long Sequence: We examine how the repair capability of LLMs is influenced by the input and output lengths. From Table XVI, we observe that LLMs suffer from the long sequence problem, i.e., as the length of input/output sequences increases, the repair capability of the model decreases. Even many studies [7], [24], [38] have highlighted the long sequence problem, alleviating this problem remains the way forward.

Finding 12: The repair capability of LLMs suffers from the long sequence problem in general.

V. PEFT4LLM STUDY

A. Study Setup

The LLM4APR research is constrained by computing resources and model size, with the emergence of billion-parameter models posing a significant challenge. Therefore, it is crucial to investigate the impact of Parameter-Efficient Fine-Tuning techniques. This study aims to explore how PEFT techniques affect the repair capabilities of large-scale LLMs. The goal is to provide guidance on selecting appropriate PEFT strategies for LLM4APR studies, especially in resource-constrained scenarios. Specifically, we investigate the repair ability of the 5 billion-level LLMs (InCoder-6B,

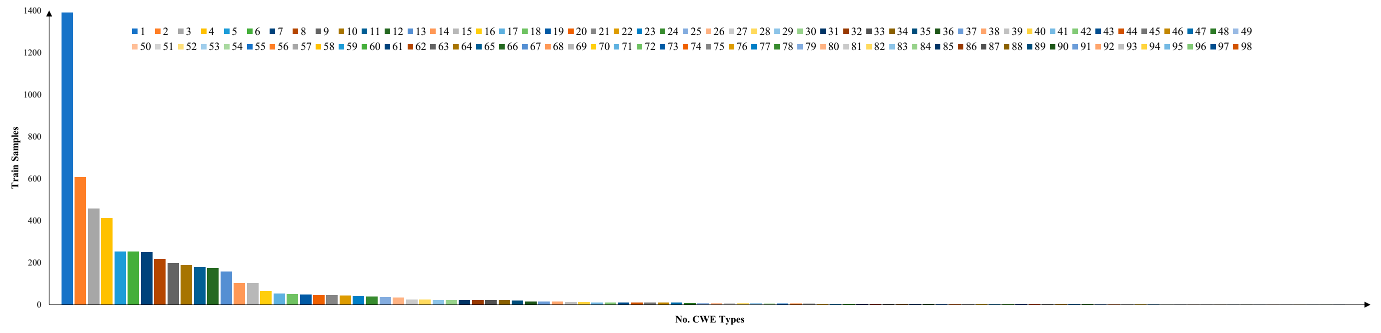


Fig. 7. Distribution of Long-tailed VulRepair dataset.

TABLE XVI
IMPACT OF INPUT/OUTPUT LENGTHS ON LLMS FOR VULNERABILITY REPAIR

CodeBERT		Input Length					
		0-100	101-200	201-300	301-400	401-500	500+
Output Length	0-10	69%	65%	75%	65%	86%	63%
	11-20	53%	60%	75%	57%	80%	52%
	21-30	46%	57%	65%	57%	50%	65%
	31-40	44%	61%	67%	47%	59%	72%
	41-50	37%	59%	39%	50%	58%	44%
	50+	29%	28%	32%	27%	14%	24%

GCodeBERT		Input Length					
		0-100	101-200	201-300	301-400	401-500	500+
Output Length	0-10	76%	72%	77%	71%	76%	65%
	11-20	57%	64%	77%	57%	80%	52%
	21-30	49%	59%	65%	57%	50%	67%
	31-40	52%	61%	68%	53%	59%	74%
	41-50	37%	59%	39%	50%	58%	50%
	50+	32%	28%	33%	29%	18%	25%

PLBART		Input Length					
		0-100	101-200	201-300	301-400	401-500	500+
Output Length	0-10	77%	85%	82%	88%	86%	71%
	11-20	74%	74%	84%	64%	83%	56%
	21-30	78%	74%	70%	57%	50%	67%
	31-40	63%	70%	71%	53%	59%	70%
	41-50	63%	74%	39%	50%	58%	50%
	50+	54%	35%	41%	27%	18%	27%

CodeT5		Input Length					
		0-100	101-200	201-300	301-400	401-500	500+
Output Length	0-10	81%	89%	82%	94%	95%	72%
	11-20	81%	70%	88%	75%	83%	61%
	21-30	83%	81%	73%	71%	50%	68%
	31-40	63%	76%	79%	60%	59%	68%
	41-50	63%	74%	50%	70%	58%	58%
	50+	57%	37%	42%	29%	21%	21%

UniXcoder		Input Length					
		0-100	101-200	201-300	301-400	401-500	500+
Output Length	0-10	83%	83%	86%	88%	95%	72%
	11-20	82%	79%	88%	71%	83%	60%
	21-30	83%	74%	73%	63%	50%	71%
	31-40	63%	73%	79%	60%	59%	68%
	41-50	63%	74%	50%	60%	58%	58%
	50+	54%	32%	38%	31%	21%	27%

CodeGeeX2-6B, CodeGen25-7B, StarCoder-15B, CodeLlama-34B) under 4 PEFT strategies (LoRA, AdaLoRA, IA3, FPFT) on the Transfer dataset. Considering the huge cost of model fine-tuning, we only randomly select approximately 45k data from the Transfer dataset for fine-tuning and testing (train/val/test=45508/948/949), and we use 8-Bit Quantization to load LLMs. As shown in Table XVII, we present the specific parameter settings for the PEFT4LLM study. These settings are based on previous research and our initial experiments to achieve optimal performance for large-scale LLMs.

B. Empirical Results

Table XVIII-XX show the repair results, memory cost, and time cost of 5 billion-level LLMs under 4 fine-tuning straties.

C. Research Questions

1) RQ5: How do different PEFT options affect LLMs' repair capability?

a) Repair Results: As shown in Table XVIII, we explore the impact of different fine-tuning strategies on the repair effectiveness of LLMs to provide guidance for maximizing the repair capability of LLMs. Specifically, LoRA techniques excel in maximizing repair capabilities, outperforming even full parameter fine-tuning. Interestingly, some advanced techniques, such as AdaLoRA and IA3, which claim to be improvements over LoRA, performed worse in the LLM4APR study. Overall, fine-tuning large-scale billion-parameter LLMs using LoRA proves to be the best option for maximizing repair capabilities.

Finding 13: In maximizing the repair capability of LLMs, LoRA is superior to FPFT as well as AdaLoRA and IA3.

b) Memory Cost: As shown in Table XIX, we present the memory cost when using different fine-tuning strategies for LLMs. Specifically, PEFT techniques (LoRA, AdaLoRA, IA3) save almost half of the memory compared to full parameter fine tuning. In particular, IA3 has the lowest memory cost due to involving fewer fine-tuned parameters, which is consistent with what its paper claims. Overall, PEFT techniques offer significant advantages in reducing memory costs for fine-tuning, with IA3 being especially beneficial due to its lower memory requirements.

Finding 14: In saving the memory cost required for fine-tuning LLMs, PEFT techniques have a significant advantage, and IA3 maintains the lowest memory cost.

c) Time Cost: As shown in Table XX, we present the time costs associated with different fine-tuning strategies for LLMs. We analyzed the average time taken to process one BFP during LLM fine-tuning. Notably, PEFT techniques incur significantly higher time costs compared to full-parameter fine-tuning. This indicates that while PEFT techniques offer good repair capabilities and lower memory costs, they do so at the expense of increased time costs. Among PEFT techniques, IA3 maintains the lowest time cost, consistent with the claims made in its paper.

Finding 15: In saving the time cost required for fine-tuning LLMs, FPFT has a significant advantage, and IA3 is one of the PEFT techniques with the lowest time cost.

TABLE XVII
PARAMETER SETTINGS FOR PEFT4LLM STUDY. (LR: LORA_R; LA: LORA_ALPHA; LD: LORA_DROPOUT; IR: INIT_R; TR: TARGET_R; B1: BETA1; B2: BETA2; DT: DELTAT; I.O.: MAX INPUT/OUTPUT LENGTH; L.R.: LEARNING RATE; T.E.: TRAINING EPOCH; B.W.: BEAM WIDTH; P.N.: PATCH NUMBER)

LLM	LoRA					AdaLoRA									IA3			FPFT	General Parameter Setting			
	L.R.	LR	LA	LD	target_modules	L.R.	IR	TR	B1	B2	DT	LA	LD	target_modules	L.R.	target_modules	feedforward_modules	L.R.	I.O.	T.E.	B.W.	P.N.
InCoder-6B	1e-4	16	32	0.05	["k_proj", "v_proj", "q_proj", "out_proj", "fc1", "fc2"]	1e-3	12	8	0.85	0.85	10	32	0.05	["k_proj", "v_proj", "q_proj", "out_proj", "fc1", "fc2"]	1e-3	["k_proj", "v_proj", "q_proj", "out_proj", "fc1", "fc2"]	["fc1", "fc2"]	5e-5	2048	1	1	1
CodeGeeX2-6B	5e-5	16	32	0.05	default	1e-3	12	8	0.85	0.85	10	32	0.05	["query_key_value"]	1e-3	["query_key_value", "mlp.dense_4h_to_h"]	["mlp.dense_4h_to_h"]	5e-6	2048	1	1	1
CodeGen25-7B	5e-5	16	32	0.05	default	5e-5	12	8	0.85	0.85	10	32	0.05	default	5e-5	default	default	5e-5	2048	1	1	1
CodeLlama-34B	5e-5	16	32	0.05	["q_proj", "k_proj", "v_proj", "o_proj"]	1e-4	12	8	0.85	0.85	10	32	0.05	default	1e-3	default	default	5e-5	2048	1	1	1
StarCoder-15B	5e-5	16	32	0.05	["c_proj", "c_attn", "q_attn"]	5e-5	12	8	0.85	0.85	10	32	0.05	default	5e-5	default	default	5e-6	2048	1	1	1

TABLE XVIII
REPAIR EFFECTIVENESS OF 5 BILLION-LEVEL LLMS UNDER 4 FINE-TUNING STRATIES. (Z%: REPAIR ACCURACY)

LLM	Fine-Tuning Strategy			
	LoRA	AdaLoRA	IA3	FPFT
InCoder-6B	**13.17%**	12.96%	12.01%	12.64%
CodeGeeX2-6B	14.23%	13.17%	12.86%	**16.23%**
CodeGen25-7B	**16.75%**	14.23%	8.11%	11.70%
StarCoder-15B	**15.49%**	14.23%	10.96%	13.07%
CodeLlama-34B	**21.39%**	12.01%	6.95%	16.02%
#Average	**16.21%**	13.32%	10.18%	13.93%

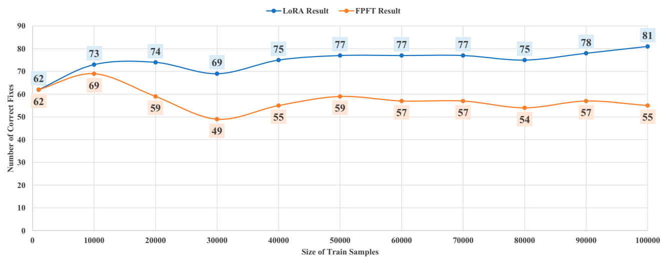


Fig. 8. Correct fixes for HumanEval-Java generated by InCoder-6B that are fine-tuned with different-sized data.

TABLE XIX
MEMORY COST OF 5 BILLION-LEVEL LLMS UNDER 4 FINE-TUNING STRATIES

LLM	Fine-Tuning Strategy			
	LoRA	AdaLoRA	IA3	FPFT
InCoder-6B	11.37G	11.41G	**11.21G**	27.99G
CodeGeeX2-6B	12.11G	**12.10G**	12.55G	33.91G
CodeGen25-7B	13.16G	**13.04G**	13.15G	27.91G
StarCoder-15B	23.44G	23.58G	**23.17G**	44.42G
CodeLlama-34B	41.51G	41.41G	**40.75G**	79.13G
#Average	20.32G	20.31G	**20.17G**	42.67G

TABLE XX
TIME COST OF 5 BILLION-LEVEL LLMS UNDER 4 FINE-TUNING STRATIES

LLM	Fine-Tuning Strategy			
	LoRA	AdaLoRA	IA3	FPFT
InCoder-6B	0.94s/BFP	1.08s/BFP	0.85s/BFP	**0.40s/BFP**
CodeGeeX2-6B	0.73s/BFP	0.75s/BFP	0.72s/BFP	**0.57s/BFP**
CodeGen25-7B	0.96s/BFP	1.04s/BFP	0.97s/BFP	**0.47s/BFP**
StarCoder-15B	1.59s/BFP	1.55s/BFP	1.53s/BFP	**0.68s/BFP**
CodeLlama-34B	3.62s/BFP	3.57s/BFP	3.49s/BFP	**1.36s/BFP**
#Average	1.57s/BFP	1.60s/BFP	1.51s/BFP	**0.70s/BFP**

d) Overall Performance: Here, we analyze the overall performance of different fine-tuning strategies to provide guidance for practical PEFT4LLM research. Firstly, among the PEFT techniques, LoRA demonstrates a clear advantage in repair effectiveness compared to the other two PEFT techniques and the FPFT strategy. Additionally, LoRA's memory and time costs are not significantly different from those of AdaLoRA and IA3. While FPFT maintains an advantage in terms of time cost, it incurs higher memory costs and does not significantly outperform LoRA in repair effectiveness. To conclude, in memory-constrained scenarios, LoRA is the more appropriate option for practical use.

Finding 16: Overall, the LoRA technique is an optimal option for resource-constrained scenarios.

e) LoRA vs. FPFT: Previous repair results have demonstrated the superiority of LoRA over FPFT in maximizing the repair capability. To further reveal the reason behind this, we follow the previous work [13] to explore the impact of the model's repair capability under different fine-tuning data sizes when using LoRA and FPFT. Here, we chose the HumanEval-Java [13] with no risk of data leakage as the test benchmark and implement fine-tuning with 100K training samples randomly selected from the Transfer dataset [48]. For model selection, we chose InCoder-6B and StarCoder2-15B, representing the best and worst performance on HEV from previous experiments (Table XII). In addition, we used 8-Bit Quantization in both LoRA and FPFT experiments to save memory.

As shown in Figs. 8 and 9, we show the repair results for each checkpoint using LoRA and FPFT strategies under different fine-tuning data sizes. These results clearly show that the model's repair capability does not always improve with a larger fine-tuning data size. For example, when using the FPFT strategy, InCoder's repair performance after fine-tuning with 100K samples is worse than after fine-tuning with 10K samples (55 vs. 69). Similarly, StarCoder's performance after fine-tuning with 100K samples is worse than after fine-tuning with 20K samples (98 vs. 104). Additionally, with the LoRA strategy, InCoder's repair results after fine-tuning with 30K samples are worse than after fine-tuning with 20K samples (69 vs. 74), and StarCoder's results follow a similar trend, with performance dropping from 129 (20K samples) to 126 (30K

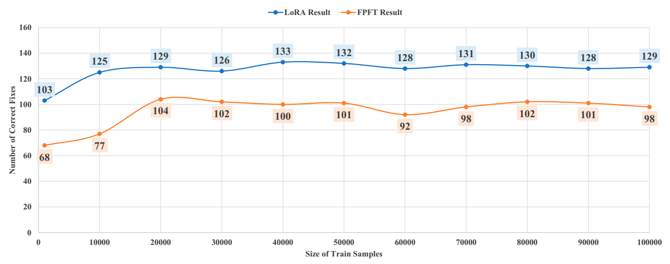
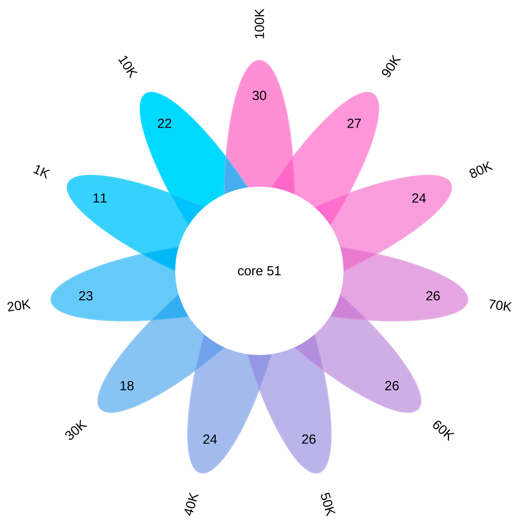


Fig. 9. Correct fixes for HumanEval-Java generated by StarCoder2-15B that are fine-tuned with different-sized data.

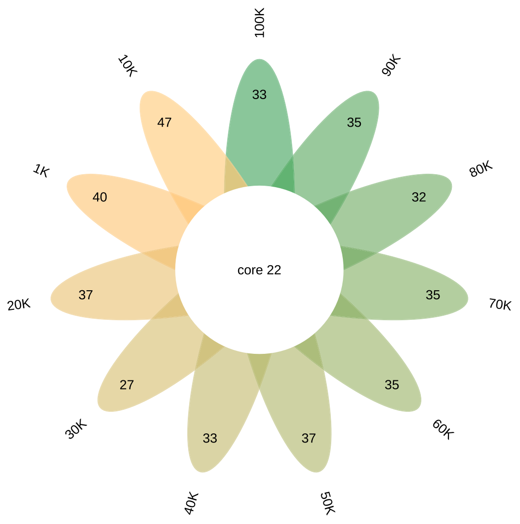
samples). This indicate that LLMs experience varying degrees of catastrophic forgetting when using LoRA and FPFT.

To investigate the extent to which LoRA and FPFT suffer from the catastrophic forgetting problem, we use flower plots [110] to visualize the commonalities in the model's repair capabilities across different fine-tuning data sizes. As shown in Figs. 10 and 11, In particular, we observe that LoRA consistently maintains more common fixes than FPFT. For instance, all InCoder checkpoints achieved 29 more common fixes with LoRA than with FPFT (51 vs. 22), and all StarCoder checkpoints achieved 30 more common fixes with LoRA than with FPFT (84 vs. 54). These results suggest that LLMs experience a milder catastrophic forgetting problem when using LoRA, as it retains more prior repair knowledge. In contrast, the FPFT strategy suffers from more severe forgetting, as indicated by the lower number of common fixes, suggesting a greater loss of previously learned repair knowledge during the continual learning process.

In summary, the experimental results demonstrate that both LoRA and FPFT fine-tuning strategies suffer from the catastrophic forgetting problem to varying degrees. However, compared to FPFT, the LoRA technique better mitigates catastrophic forgetting [111], retaining more of the repair knowledge acquired during the continual learning process, which results in improved repair capabilities.



(a) Flower diagram of InCoder-6B when using LoRA.



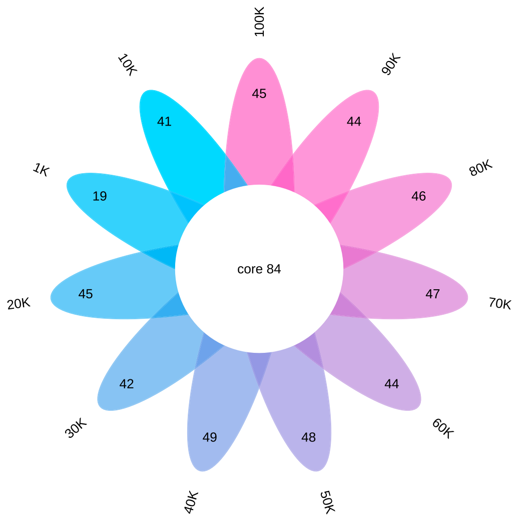
(b) Flower diagram of InCoder-6B when using FPFT.

Fig. 10. Flower diagram for 11 checkpoints of InCoder-6B when using LoRA and FPFT.

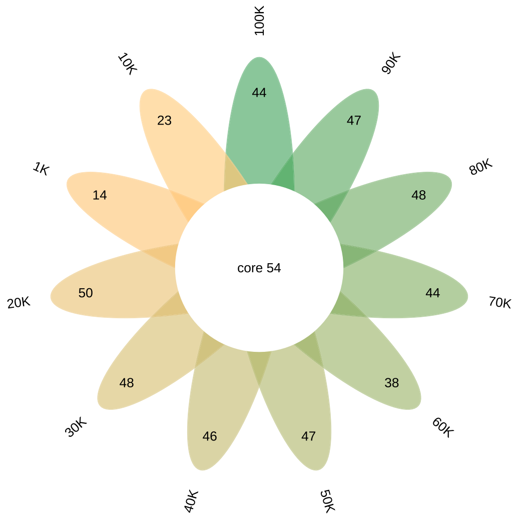
VI. APR$_4$LLM Study

A. Study Setup

Above experiments only explored the repair ability of LLMs under the basic NMT fine-tuning strategy, and it remains to be explored how to further enhance the repair ability of LLMs. In recent learning-based APR work, various novel repair strategies have been proposed. Therefore, we will explore the generalizability of these repair strategies on LLMs, and further analyze their potential in enpowering the repair ability of LLMs. Specifically, we reproduce 4 repair strategies (NMT, ITER, TENURE, KATANA) on 5 LLMs (InCoder-6B, CodeGeeX2-6B, CodeGen25-7B, CodeLlama-13B, StarCoder-15B) and choose the Transfer dataset as the training data and Defects4J V1.2 as the test benchmark. In the fine-tuning process, we followed the experience of the PEFT4APR study (Section V) in fine-tuning all models using the 8-Bit Quantization and LoRA techniques. In the parameter settings, we follow the settings of Task ❼ in the LLM4APR study (Section IV). In particular, ITER employs an iterative patch generation strategy, which is different from other repair strategies that generate all the patches at once, which can extend and explore more deeper patch space. Note that when applying the ITER strategy to LLMs, the substantial computational cost makes it impractical to use a large beam size. Therefore, we follow the default settings of the original paper of ITER and set the beam size to 4 and iter size to 3. This means that 4 patches are generated in each iteration, with iteration rounds of 3, and the final patch space contains at most 86 ($4+4^2+4^3$) patches.



(a) Flower diagram of StarCoder2-15B when using LoRA.



(b) Flower diagram of StarCoder2-15B when using FPFT.

Fig. 11. Flower diagram for 11 checkpoints of StarCoder2-15B when using LoRA and FPFT.

B. Empirical results

Table XXI shows the impact of using different repair strategies on the repair capability of LLMs. The first row of the table indicates the selected model and the second row indicates the used repair strategy. For example, the StarCoder fine-tuning implementation using the NMT strategy (StarCoder$_{NMT}$) produced 93 plausible fixes on Defects4J V1.2, of which 74 were correct patches.

C. Research Questions

1) RQ6: **How well do recent repair strategies work on the LLM basis?**

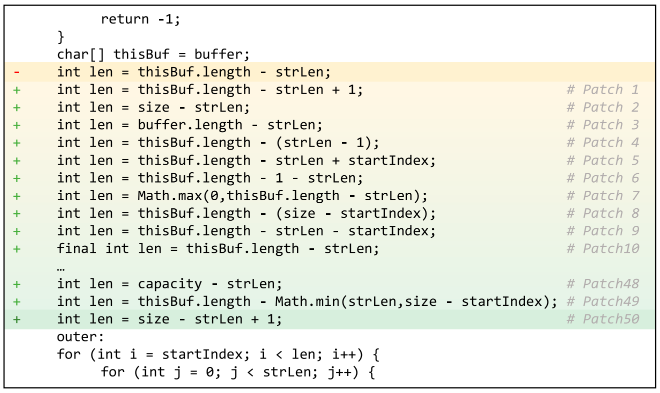
a) NMT: Although the basic NMT strategy works effectively on these LLMs, the patch generation process, which relies solely on learning the mapping between buggy and fixed code pairs, has its limitations. Specifically, while fine-tuning LLMs with NMT enables the identification of defective code hunks and the generation of potential patches, the traditional NMT workflow lacks additional domain knowledge during the patch generation phase. This deficiency can result in a suboptimal patch space. For example, as shown in Fig. 12, the irrelevant variable `thisBuf.length` predominantly appears in the patch space, whereas the key repair ingredient, the variable `size`, is rarely present. As we continued to expand the patch space, it was not until the Top-50 patch that LLM synthesized the correct patch using this crucial repair ingredient. This case suggests that the patch synthesis process for the NMT strategy is unguided, leading to blind exploration of the patch space and resulting in suboptimal positions for the correct patches. Ideally, the patch space should contain more relevant ingredients, allowing correct patches to appear earlier. To alleviate this problem, enabling the model to better understand code elements and decide which key ingredients to employ to constrain its patch synthesis process is a viable path to optimizing the patch space.

Finding 17: The basic NMT strategy still works effectively on decoder-only LLMs, but its lack of guidance on the patch generation process may result in a patch space that is sub-optimal.

b) TENURE: The TENURE strategy aims to enhance patch generation by providing additional template guidance to the model [49]. Theoretically, this guidance helps the model synthesize repair behaviors that align with the target template. However, surprisingly, the TENURE strategy appears to slightly reduce the repair capability of the model. In specific, as shown in Table XXI, the TENURE implementation on top of LLM performs even worse than the basic NMT fine-tuning. We analyzed the reasons behind this and found that LLMs tend to predict repeated templates consistently. Here, we analyze a case where the NMT strategy successfully repairs a bug, while the TENURE strategy fails, highlighting the issue. As shown in Fig. 13, the fourth candidate patch (Patch 4) generated by StarCoder$_{NMT}$ agrees with the developer's patch and successfully fixes Chart-12. However, the top-10 candidates generated by StarCoder$_{TENURE}$ all fail to fix it. We observe that the correct repair template for Chart-12 should be $MutateSingleLine$ (i.e., mutating an entire line of code, see TENURE's template settings [49] for details). Unfortunately, StarCoder$_{TENURE}$ always predicts incorrect templates, which leads to synthesizing incorrect repair behavior. Moreover, we observe that the TENURE strategy tends to predict duplicate templates (e.g., the repair templates for StarCoder$_{TENURE}$'s Patch 2-10 are all $MutateVariable$), which leads to the possibility that it misses the chance of correct templates, thus limiting the model's repair capability. Overall, the challenge with the TENURE strategy and LLMs lies in covering a diverse distribution of templates with the model's predictions. To address this, incorporating a specialized template ranking step could help cover the full range of template classes and prioritize templates, potentially improving the strategy's effectiveness.

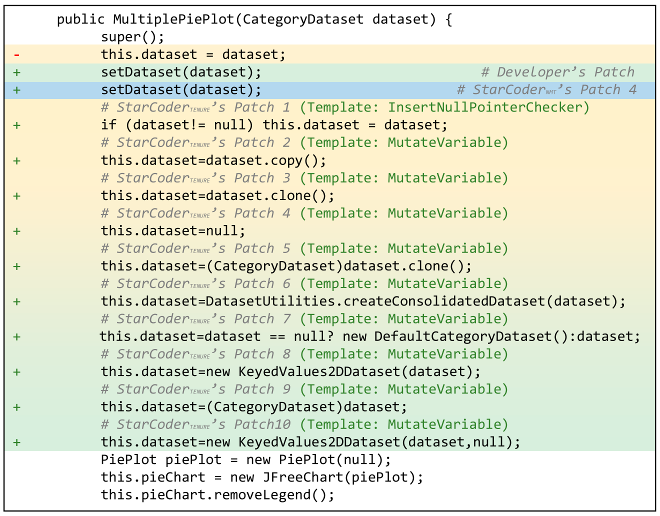
TABLE XXI
REPAIR RESULTS OF DIFFERENT REPAIR STRATEGIES FOR BILLION-LEVEL LLMS ON DEFECTS4J V1.2

LLM	**InCoder-6B**				**CodeGeeX2-6B**				**CodeGen25-7B**				**CodeLlama-13B**				**StarCoder-15B**			
Strategy	NMT	TENURE	KATANA	ITER	NMT	TENURE	KATANA	ITER	NMT	TENURE	KATANA	ITER	NMT	TENURE	KATANA	ITER	NMT	TENURE	KATANA	ITER
Beam Size	10	10	10	4	10	10	10	4	10	10	10	4	10	10	10	4	10	10	10	4
Patch Size	10	10	10	84	10	10	10	84	10	10	10	84	10	10	10	84	10	10	10	84
Chat	11/14	12/15	10/12	11/13	12/14	9/10	9/10	12/13	11/14	10/12	9/11	12/14	12/14	11/12	11/11	13/14	12/15	11/13	10/12	12/15
Closure	22/29	22/30	21/26	20/29	19/23	18/20	17/20	15/19	26/30	22/29	24/27	25/32	27/32	22/26	28/34	23/32	34/37	28/32	31/36	25/30
Lang	13/19	13/21	14/23	12/20	14/19	10/16	12/19	12/18	15/23	11/20	17/22	18/28	19/25	19/23	17/25	21/27	16/23	14/22	14/23	16/23
Math	21/24	20/24	20/26	18/26	24/29	19/20	23/26	20/22	28/35	26/32	30/36	26/39	29/32	24/28	31/38	29/35	30/36	27/33	30/37	31/37
Mockito	4/4	3/4	4/4	5/6	4/5	4/5	3/3	5/6	5/6	5/6	5/6	5/5	5/5	3/4	4/5	8/9	6/7	4/5	5/5	6/8
Time	3/3	2/3	2/2	3/4	2/3	2/3	3/4	2/4	3/3	3/3	3/3	2/6	3/4	3/5	3/4	3/5	3/4	3/4	3/5	4/6
#Total	74/93	72/97	71/93	69/98	75/93	62/74	67/82	66/82	88/111	77/102	88/105	88/124	95/112	82/98	94/117	97/122	101/122	87/109	93/118	94/119



```
        return -1;
    }
    char[] thisBuf = buffer;
-   int len = thisBuf.length - strLen;
+   int len = thisBuf.length - strLen + 1;                          # Patch 1
+   int len = size - strLen;                                        # Patch 2
+   int len = buffer.length - strLen;                               # Patch 3
+   int len = thisBuf.length - (strLen - 1);                        # Patch 4
+   int len = thisBuf.length - strLen + startIndex;                 # Patch 5
+   int len = thisBuf.length - 1 - strLen;                          # Patch 6
+   int len = Math.max(0,thisBuf.length - strLen);                  # Patch 7
+   int len = thisBuf.length - (size - startIndex);                 # Patch 8
+   int len = thisBuf.length - strLen - startIndex;                 # Patch 9
+   final int len = thisBuf.length - strLen;                        # Patch10
    …
+   int len = capacity - strLen;                                    # Patch48
+   int len = thisBuf.length - Math.min(strLen,size - startIndex); # Patch49
+   int len = size - strLen + 1;                                    # Patch50
    outer:
    for (int i = startIndex; i < len; i++) {
        for (int j = 0; j < strLen; j++) {
```

Fig. 12. Candidate patches generated by StarCoder$_{NMT}$ for Lang-61 (Patch 1-49 are incorrect patches, Patch 50 is the correct patch).

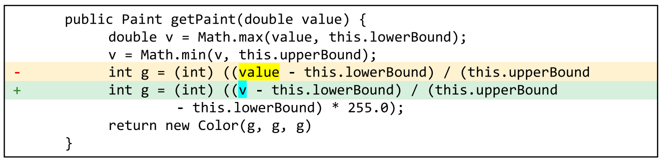


```
    public MultiplePiePlot(CategoryDataset dataset) {
        super();
-       this.dataset = dataset;
+       setDataset(dataset);                               # Developer's Patch
+       setDataset(dataset);                            # StarCoderNMT's Patch 4
        # StarCoderTENURE's Patch 1 (Template: InsertNullPointerChecker)
+       if (dataset!= null) this.dataset = dataset;
        # StarCoderTENURE's Patch 2 (Template: MutateVariable)
+       this.dataset=dataset.copy();
        # StarCoderTENURE's Patch 3 (Template: MutateVariable)
+       this.dataset=dataset.clone();
        # StarCoderTENURE's Patch 4 (Template: MutateVariable)
+       this.dataset=null;
        # StarCoderTENURE's Patch 5 (Template: MutateVariable)
+       this.dataset=(CategoryDataset)dataset.clone();
        # StarCoderTENURE's Patch 6 (Template: MutateVariable)
+       this.dataset=DatasetUtilities.createConsolidatedDataset(dataset);
        # StarCoderTENURE's Patch 7 (Template: MutateVariable)
+       this.dataset=dataset == null? new DefaultCategoryDataset():dataset;
        # StarCoderTENURE's Patch 8 (Template: MutateVariable)
+       this.dataset=new KeyedValues2DDataset(dataset);
        # StarCoderTENURE's Patch 9 (Template: MutateVariable)
+       this.dataset=(CategoryDataset)dataset;
        # StarCoderTENURE's Patch10 (Template: MutateVariable)
+       this.dataset=new KeyedValues2DDataset(dataset,null);
        PiePlot piePlot = new PiePlot(null);
        this.pieChart = new JFreeChart(piePlot);
        this.pieChart.removeLegend();
```

Fig. 13. Candidate patches generated by StarCoder$_{NMT}$ and StarCoder$_{TENURE}$ for Chart-12 (StarCoder$_{NMT}$'s Patch 4 is correct patch, StarCoder$_{TENURE}$'s Patch 1-10 are incorrect patches).

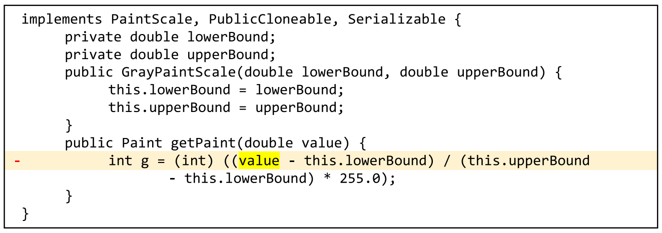
Finding 18: The TENURE repair strategy shows limited effectiveness on LLMs, primarily due to the model's inability to generate a diverse distribution of templates for generating correct patches.

c) KATANA: The KATANA strategy aims to simplify the context and extract relevant elements (repair ingredients) by slicing the input bug code [54]. Briefly, KATANA reduces noise and captures relevant repair ingredients by analyzing statements that have a control or data dependency on the buggy statement.



```
    public Paint getPaint(double value) {
        double v = Math.max(value, this.lowerBound);
        v = Math.min(v, this.upperBound);
-       int g = (int) ((value - this.lowerBound) / (this.upperBound
+       int g = (int) ((v - this.lowerBound) / (this.upperBound
            - this.lowerBound) * 255.0);
        return new Color(g, g, g)
    }
```

Fig. 14. Chart-24 and its patch.



```
implements PaintScale, PublicCloneable, Serializable {
    private double lowerBound;
    private double upperBound;
    public GrayPaintScale(double lowerBound, double upperBound) {
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
    }
    public Paint getPaint(double value) {
-       int g = (int) ((value - this.lowerBound) / (this.upperBound
            - this.lowerBound) * 255.0);
    }
}
```

Fig. 15. Chart-24 with KATANA's slicing strategy.

As shown in Table XXI, KATANA's slicing strategy does not seem to achieve a significant improvement compared to the NMT strategy of inputting a complete method-level context. Here, we find that blind slicing may remove the necessary repair ingredients, resulting in the model's failure to synthesize the correct patch. For example, we take Chart-24 as an example that was successfully fixed by all LLMs' NMT implementation, but failed to be fixed by the KATANA implementation. Specifically, as shown in Fig. 14, the NMT strategy uses the full method-level input, and the correct patch for the bug is to replace the variable $value$ with v, where v appears in the context as a potential repair ingredient. However, as shown in Fig. 15, KATANA performs slicing by retaining only the relevant code elements in the bug statement and removing the irrelevant content in the context, which results in missing the repair ingredient v and thus fails to synthesize the correct patch. Overall, this is due to the fact that KATANA's slicing strategy may ignore the root cause of the bug in order to retain the necessary relevant ingredients. It also shows that simply slicing code elements using their dependencies is not enough and that it is still necessary to retain the necessary context to fully understand the root cause of the bug.

Finding 19: KATANA's slicing strategy, while helpful in simplifying contextual input and capturing relevant code elements, may also result in the loss of necessary repair ingredients.

TABLE XXII
REPAIR RESULTS OF ITER'S STRATEGY FOR BILLION-LEVEL LLMS ON DEFECTS4J V1.2

LLM	**InCoder-6B**			**CodeGeeX2-6B**			**CodeGen25-7B**			**CodeLlama-13B**			**StarCoder-15B**		
ITER Size	ITER 1	ITER 2	ITER 3	ITER 1	ITER 2	ITER 3	ITER 1	ITER 2	ITER 3	ITER 1	ITER 2	ITER 3	ITER 1	ITER 2	ITER 3
Beam Size	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Patch Size	4	20	84	4	20	84	4	20	84	4	20	84	4	20	84
Chat	10/12	11/13	11/13	11/12	12/13	12/13	11/13	12/14	12/14	12/13	13/14	13/14	12/14	12/15	12/15
Closure	17/23	20/29	20/29	14/17	15/18	15/19	17/21	24/30	25/32	18/22	19/26	23/32	19/22	24/29	25/30
Lang	9/12	10/17	12/20	12/17	12/18	12/18	10/15	14/24	18/28	16/21	19/25	21/27	13/17	16/22	16/23
Math	14/17	15/23	18/26	19/20	19/20	20/22	24/28	26/37	26/39	22/26	28/34	29/35	20/25	26/32	31/37
Mockito	3/4	5/6	5/6	4/5	5/6	5/6	3/4	4/4	5/5	3/3	6/6	8/9	4/5	6/8	6/8
Time	3/4	3/4	3/4	2/2	2/3	2/4	2/4	2/5	2/6	2/3	2/4	3/5	2/3	3/5	4/6
#Total(Corr./Plaus.)	56/72	64/92	69/98	62/73	65/78	66/82	67/85	82/114	88/124	73/88	87/109	97/122	70/86	87/111	94/119

```
static boolean mayBeString(Node n, boolean recurse) {
    if (recurse) {
-       return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
+       return allResultsMatch(n,MAY_BE_STRING_PREDICATE,true);  # Patch 1
+       return allResultsMatch(n,MAY_BE_STRING_PREDICATE,false); # Patch 2
+       return allResultsMatch(n,MAY_BE_STRING_PREDICATE,true);  # Patch 3
    } else {
        return mayBeStringHelper(n);
}
```

Fig. 16. StarCoder$_{ITER}$'s patches for Clourse-10.

d) ITER: The iterative repair paradigm of the ITER strategy is helpful in extending and exploring the patch space, which further alleviates the limitation of computing resources on the limited patch space. The cleverness of the ITER strategy lies in its ability to iteratively generate subsequent patches by leveraging previous patches, thereby greatly expanding and exploring deeper patch spaces. For example, as shown in Table XXI, the ITER strategy extends the patch space to 86 patches through multiple iterations with a smaller beam size (beam size is 4), which rivals or even exceeds the repair results of other non-iterative generation strategies (e.g., CodeGen$_{ITER}$ vs. CodeGen$_{NMT}$ and CodeLlama$_{ITER}$ vs. CodeLlama$_{NMT}$) with a larger beam size (beam size is 10). Meanwhile, as shown in Table XXII, the ITER strategy can explore the patch space more deeply and improve the repair results as the iteration number increases. However, the ITER strategy also carries limitations in patch generation. We observe that ITER's patch generation may get stuck in a loop of iterations. As shown in Fig. 16, the first iteration of ITER generates Patch 1 is an incorrect patch, and Patch 2 is generated after implementing second iterative fixes on top of Patch 1, and yet the third iteration's Patch 3 remains consistent with the previous Patch 1. This results in numerous duplicate incorrect patches in the patch space, which affects the repair efficiency.

Finding 20: ITER's iterative repair paradigm is an effective measure to alleviate patch space constraints in the era of LLMs, however it may have duplicate items in the patch space.

e) Repair Overlap: In order to present a clearer picture of the impact of different repair strategies on the repair capability of the LLM, we analyze the overlap in repair capability of LLM implementations based on different repair strategies. Fig. 17 illustrates that each strategy successfully fixes a significant portion of bugs, indicating that different repair strategies make unique contributions to the repair capability of LLMs. In real-world scenarios, these LLMs, implemented based on different strategies, can be ensembled as distinct expert models to further enhance overall repair results.

Finding 21: Different repair strategies have unique contributions to the repair capabilities of the LLM, and there is potential to ensemble them to build powerful repair tools.

VII. DISCUSSION

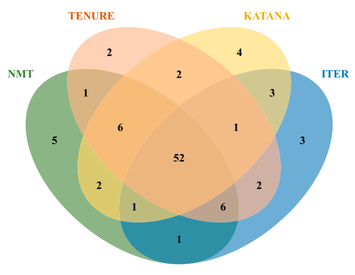
This section discusses the limitations identified in our study that affect the repair capability of LLMs and seeks directions for improvement.

1) Loss of Pre-Trained Knowledge: As described in **Finding 9**, after fine-tuning, LLMs may lose some of the knowledge learned from the pre-training phase compared to zero-shot learning [17]. Furthermore, we noticed that AlphaRepair [17] converts the repair task into a *cloze* task (MLM) rather than a *translation* task (NMT). The *cloze* task could better fit the model's pre-training task (i.e., MLM). That is, it predicts the token at the mask location based on the contextual tokens. However, it is unclear how the repair ability differs using the two paradigms (NMT and MLM). Therefore, we suggest exploring the following two directions.

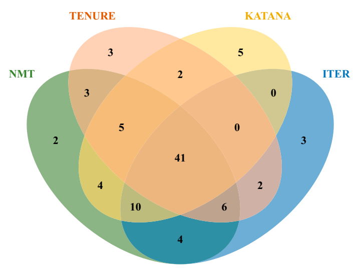
D1: Mitigation of catastrophic forgetting. There have been various mitigation measures towards this problem [108], and it is meaningful to introduce these techniques into APR.
D2: NMT vs. MLM. Fine-tuning the LLM through both NMT and MLM tasks allows us to explore the differences in repair capabilities between the two learning paradigms.

2) Lack of Repair Ingredients and Long Sequence Problem: As described in **Finding 10**, the input/output length limit of the model make LLMs can not cover sufficient repair ingredients, which in turn constraints the repair capability. Furthermore, **Finding 12** also points out that LLMs suffer from the long sequence problem.

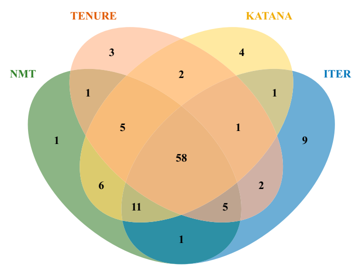
D1: Precise context extraction. Through data/control flow analysis, we can trim irrelevant context [56], aid in pinpointing defect locations and guide repairs.
D2: Essential repair ingredients. We can integrate traditional APR techniques based on redundancy assumptions [18] with LLMs to introduce additional repair elements into the model input.
D3: Breaking the length limit. One way to model long sequences for covering more repair ingredients is MegaByte [112]. In addition, adopting sliding-encoder and decoder (SLED) [113] to partition the input into overlapping chunks may also help accept long and/or dependent methods.



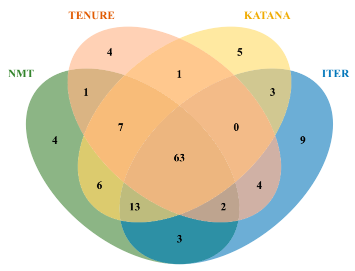
(a) Implementation of Different Repair Strategies on InCoder-6B.



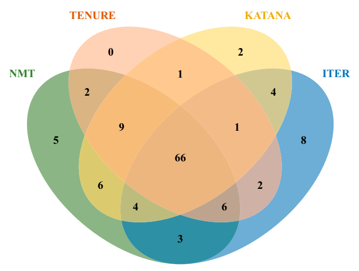
(b) Implementation of Different Repair Strategies on CodeGeeX2-6B.



(c) Implementation of Different Repair Strategies on CodeGen25-7B.



(d) Implementation of Different Repair Strategies on CodeLlama-13B.



(e) Implementation of Different Repair Strategies on StarCoder-15B.

Fig. 17. Bug fix Venn diagram on Defects4J V1.2.

3) Long-Tail Problem: As described in **Finding 11**, the long-tail problem caused by imbalanced type distribution in vulnerability repair tasks remains a key challenge. Also, there is the small sample problem. Previous works [33], [56] have proposed using transfer learning to alleviate this problem.

D1: Data augmentation. Designing data augmentation schemes [114] can expand the number of training samples and mitigate the challenge of small sample sizes.
D2: Combining multiple PLs. Using meta learning [115] that integrates multiple PLs to enhance the multilingual repair capability of LLMs.

4) Multi-Hunk Fixes: As described in **Finding 6**, LLMs still have difficulty dealing with complex multi-hunk fixes.

D1: Capturing complex code dependencies. Tree [31] or graph [116] structures that capture global dependencies can enhance the model's ability to understand and handle complex repair tasks.
D2: Extracting in-depth semantic information. Leveraging high-level semantic information (such as bytecode [117] and intermediate representation [118]) can aid the model in comprehending the root cause of defects, thereby enhancing its repair capability.

VIII. Threats to Validity

Internal. Existing approaches typically use different training datasets, patch space sizes, post-processing strategies, and other details, and it would be unfair to compare these APR tools directly [119]. To mitigate this threat, we used the same dataset and beam size as baselines. Note that when comparing with DEAR [31], their paper did not specify a specific patch space size, so we followed the practice of previous works [41], [46] and chose a minimum beam size of 100 [46] for million-level LLMs. Besides, we set the beam size is 10 when fine-tuning billion-level LLMs, which is due to our memory limitations. Also, our work did not use patch filtering and re-ranking strategies, whereas some baselines like DEAR adopted the post-processing for improvement. Therefore, our results could be further improved and our comparison is fair for baselines.

External. Although we have conducted a comprehensive study of 10 LLM families for APR, with a variety of scenarios (e.g., 3 defect types, 8 test benchmarks, and 3 PLs), our results may still not generalize well to other LLMs and PLs [120]. For example, we did not include extremely LLMs (e.g., Falcon-180B) for APR, mainly because of the limited computing resource. However, we believe our results are representative for a relatively wide range of conditions. We will enhance it with more advanced resources and expect researchers of following work could improve our study.

IX. Related Work

A. LLM4APR

1) Technique: Nowadays, APR research has entered the era of LLM. Depending on the use of different learning paradigms, current LLM-based APR techniques can be divided into two types: prompt-learning and fine-tuning. The prompt learning paradigm based LLM4APR work implements repair by designing and constructing suitable prompt templates to utilize the language understanding and generation capabilities of the LLM itself. Typical works include AlphaRepair [17], ChatRepair [19], ContrastRepair [28], SRepair [121], RepairAgent [29], FixAgent [122], Repilot [20], GAMMA [30], TypeFix [22], D4C [123], Toggle [124], and so on. For example, AlphaRepair, GAMMA, and TypeFix are inspired by the template-based APR technique [101] that treats the repair task as the infilling task, thereby constructing prompted templates to mask predictions of fault locations. ChatRepair, ContrastRepair, and SRepair allow guiding the LLM to better generate fixes by providing additional information (e.g., test case test execution information, project-level repair ingredients, etc.) to the LLM by conversation. Recently, LLM4APR research has entered the era of agent, where FixAgent and RepairAgent enable repair workflows by building agent applications on top of LLM to allow LLM to act as multiple roles and execute external tools. In summary, current LLM-based APR research focuses on how to design novel strategies to better utilize (or stimulate) the repair capability (or potential) of LLMs.

2) Study: At the same time, researchers have conducted empirical studies to reveal the potential of LLM in APR tasks. Fan et al. [10] investigated whether APR techniques can improve the reliability of code produced by LLMs (Codex) and provide suggestions for enhancing APR with the help of LLMs. Xia et al. [16] and Pearce et al. [11] comprehensively explored the performance of LLMs in bug and vulnerability repair tasks using the zero/few-shot learning paradigm. Besides, some studies [58], [125] systematically compared LLMCs on various tasks, which partially include APR. Unlike their work, we focus on the repair capabilities of LLMs under the NMT fine-tuning paradigm. One similar work is Jiang et al. [13], which also used the fine-tuning paradigm. However, they focused on the comparison between zero-shot and fine-tuning for APR, and only explored the single-hunk Java bug repair task. In contrast, we make a comprehensive exploration of fine-tuning LLMs for APR across multiple languages, various defect types, and different levels of bug/vulnerability complexity. We also provide guidance on selecting the appropriate designs to enhance the repair capabilities of LLMs, and achieve the new SOTA results. Another similar work is Zhang et al. [12], which also used the fine-tuning paradigm. The difference is that they focus only on vulnerability repair, while our study covers more defect types. In addition, previous studies [12], [13], [14] have rarely fine-tuned on the basis of LLM at scales above 6B, and our extended study bridges this gap.

B. PEFT4LLM

1) Technique: With the increase in the size of LLMs, fine-tuning them becomes a resource-intensive task. For researchers with limited computing resources, fine-tuning LLMs is increasingly expensive. Therefore, PEFT technique research aims to further lower the threshold of model fine-tuning to address this challenge. Numerous PEFT techniques have been proposed and recent PEFT surveys [37], [126] have categorized them into four main types: 1) Additive Fine-tuning. These methods involve introducing new extra trainable parameters for task-specific fine-tuning, e.g., Sequential Adapter [127], P-tuning [95], IA3 [53], etc. 2) Partial Fine-tuning. These methods aim to reduce the number of fine-tuned parameters by selecting a subset of pre-trained parameters that are critical to downstream tasks while discarding unimportant ones, e.g., BitFit [128], FISH MASK [129], LT-SFT [130], etc. 3) Reparameterized Fine-tuning. These methods utilize low-rank transformation to reduce the number of trainable parameters while allowing operating with high-dimensional matrices, e.g., LoRA [51], AdaLoRA [52], QLoRA [102], etc. 4) Hybrid Fine-Tuning. These methods aim to combine various PEFT techniques to leverage the strengths of each method and mitigate their weaknesses, e.g., MAM Adapter [131], UniPELT [132], AutoPEFT [133], etc.

2) Study: Recent studies have begun to explore how to select the appropriate PEFT technique for the specific downstream task in order to maximize the model's capabilities. Liu et al. [134] examined the performance of Adapter Tuning and LoRA on 5 million-level LLMs processing code change related tasks. Liu et al. [135] explored the effectiveness of Adapter Tuning, Prefix Tuning, LoRA, and MHM techniques for 4 million-level LLMs on code summarize and generation tasks. Zou et al. [136] evaluated the effectiveness of 5 PEFT methods on 8 million-level LLMs for four software engineering tasks. Pu et al. [136] evaluating model performance across different data scales of classification and generation dataset under 4 PEFT techniques. However, current research has not explored the impact of PEFT on the base of billion-level LLMs for the code repair task, and our study bridges this gap to provide guidance on how to select appropriate PEFT techniques for billion-level LLMs to maximize the repair capability of LLMs.

X. Conclusion

This study provides a comprehensive exploration of APR research in the era LLMs, which we summarize into 3 areas. **1) LLM4APR**: We comprehensively investigate the repair capability of million/billion-level LLMs on APR tasks in order to present the significant impact of LLMs on APR research. **2) PEFT4LLM**: We thoroughly explore the impact of PEFT techniques on LLM4APR research in order to provide guidance on how to select appropriate PEFT techniques. **3) APR4LLM**: We systematically analyze the potential of novel APR techniques (repair strategies) in further enhancing the repair capability of LLMs in order to reveal the generalizability of recent repair strategies in the era of LLMs.

Acknowledgment

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[2] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17: 1–17:24, 2018.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, Jan. 2019.

[4] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 1–69, 2023.

[5] K. Huang et al., "Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities," *ACM Comput. Surv.*, vol. 57, no. 2, pp. 1–43, 2024.

[6] M. Monperrus, "The living review on automated program repair," *HAL Arch. Ouvertes*, 2018.

[7] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[8] W. X. Zhao et al., "A survey of large language models," 2023, *arXiv:2303.18223*.

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, Apr. 2024.

[10] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proc. 45th Int. Conf. Softw. Eng., ICSE*, 2023, pp. 1469–1481.

[11] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proc. 44th Symp. Secur. Privacy, SP*, 2023, pp. 2339–2356.

[12] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 4, pp. 2507–2525, Jul./Aug. 2024.

[13] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1430–1442.

[14] Y. Wu et al., "How effective are neural networks for fixing security vulnerabilities," in *Proc. 32nd Int. Symp. Softw. Testing Anal. (ISSTA)*, 2023, pp. 1282–1294.

[15] K. Huang et al., "An empirical study on fine-tuning large language models of code for automated program repair," in *Proc. 38th Int. Conf. Automated Softw. Eng.* (ASE), 2023, pp. 1162–1174.

[16] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1482–1494.

[17] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proc. 30th Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2022, pp. 959–971.

[18] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *Proc. 38th Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 522–534.

[19] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in *Proc. 33rd Int. Symp. Softw. Testing Anal. (ISSTA)*, 2024, pp. 819–831.

[20] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proc. 31st Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2023, pp. 172–184.

[21] W. Wang, Y. Wang, S. Joty, and S. C. H. Hoi, "RAP-Gen: Retrieval-augmented patch generation with CodeT5 for automatic program repair," in *Proc. 31st Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2023, pp. 146–158.

[22] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing Python type errors," in *Proc. 46th Int. Conf. Softw. Eng. (ICSE)*, 2024, pp. 1–13.

[23] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "INFERFIX: End-to-end program repair with LLMS," in *Proc. 31st Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2023, pp. 1646–1656.

[24] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A t5-based automated software vulnerability repair," in *Proc. 30th Joint Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng. (ESEC/FSE)*, 2022, pp. 935–947.

[25] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proc. 46th Int. Conf. Softw. Eng. (ICSE)*, 2024, pp. 872–872.

[26] A. Silva, S. Fang, and M. Monperrus, "RepairLLaMA: Efficient representations and fine-tuned adapters for program repair," 2023, *arXiv:2312.15698*.

[27] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, "CIGAR: Cost-efficient program repair with LLMS," 2024, *arXiv:2402.06598*.

[28] J. Kong, M. Cheng, X. Xie, S. Liu, X. Du, and Q. Guo, "ContrastRepair: Enhancing conversation-based automated program repair via contrastive test case pairs," 2024, *arXiv:2403.01971*.

[29] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in *Proc. 47th Int. Conf. Softw. Eng. (ICSE)*, 2024.

[30] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "GAMMA: Revisiting template-based automated program repair via mask prediction," in *Proc. 38th Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 535–547.

[31] Y. Li, S. Wang, and T. N. Nguyen, "DEAR: A novel deep learning-based approach for automated program repair," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 511–523.

[32] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFIX: Learning to fix coding errors with a text-to-text transformer," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2021, pp. 780–791.

[33] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in C code," *IEEE Trans. Softw. Eng.*, vol. 49, no. 1, pp. 147–165, Jan. 2023.

[34] E. Mashhadi and H. Hemmati, "Applying codeBERT for automated program repair of Java simple bugs," in *Proc. 18th Int. Conf. Mining Softw. Repositories (MSR)*, 2021, pp. 505–509.

[35] D. Yang et al., "Where were the repair ingredients for defects4j bugs?" *Empir. Softw. Eng.*, vol. 26, no. 6, 2021, Art. no. 122.

[36] K. Huang, J. Zhang, X. Meng, and Y. Liu, "Template-guided program repair in the era of large language models," in *Proc. 47th Int. Conf. Softw. Eng. (ICSE)*, 2024, pp. 367–379.

[37] Z. Han, C. Gao, J. Liu, S. Q. Zhang et al., "Parameter-efficient fine-tuning for large models: A comprehensive survey," 2024, *arXiv:2403.14608*.

[38] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021.

[39] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proc. 29th Int. Symp. Softw. Testing Anal. (ISSTA)*, 2020, pp. 101–114.

[40] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proc. 43rd Int. Conf. Softw. Eng. (ICSE)*, 2021, pp. 1161–1173.

[41] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "KNOD: Domain knowledge distilled tree decoder for automated program repair," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1251–1263.

[42] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-based code transformation learning for automated program repair," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 602–614.

[43] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "SelfAPR: Self-supervised program repair with test execution diagnostics," in *Proc. 37th Int. Conf. Automated Softw. Eng. (ASE)*, 2022, pp. 1–13.

[44] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1506–1518.

[45] H. Ye and M. Monperrus, "ITER: Iterative neural repair for multi-location patches," in *Proc. 46th Int. Conf. Softw. Eng. (ICSE)*, 2024, pp. 1–13.

[46] Q. Zhu et al., "A syntax-guided edit decoder for neural program repair," in *Proc. 29th Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2021, pp. 341–353.

[47] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1443–1455.

[48] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 1169–1180.

[49] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1456–1468.

[50] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, 2014, pp. 437–440.

[51] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2021.

[52] Q. Zhang et al., "Adaptive budget allocation for parameter-efficient fine-tuning," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2023.

[53] H. Liu et al., "Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 35, 2022, pp. 1950–1965.
[54] M. Sintaha, N. Nashid, and A. Mesbah, "KATANA: Dual slicing based context for learning bug fixes," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 100:1–100:27, 2023.
[55] "LLM4APR study." 2024. [Online]. Available: https://github.com/LLMC-APR/STUDY
[56] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "SeqTrans: Automatic vulnerability fix via sequence to sequence learning," *IEEE Trans. Softw. Eng.*, vol. 49, no. 2, pp. 564–585, Feb. 2023.
[57] W. Yuan et al., "CIRCLE: continual repair across programming languages," in *Proc. 31st Int. Symp. Softw. Testing Anal. (ISSTA)*, 2022, pp. 678–690.
[58] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2136–2148.
[59] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "CodeScore: Evaluating code generation by learning code execution," 2023, *arXiv:2301.09043*.
[60] K. Huang, S. Yang, H. Sun, C. Sun, X. Li, and Y. Zhang, "Repairing security vulnerabilities using pre-trained programming language models," in *Proc. 52nd Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, 2022, pp. 111–116.
[61] I. Nimah, M. Fang, V. Menkovski, and M. Pechenizkiy, "NLG evaluation metrics beyond correlation analysis: An empirical metric preference checklist," in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2023, pp. 1240–1266.
[62] X. Li et al., "TransRepair: Context-aware program repair for compilation errors," in *Proc. 37th Int. Conf. Automated Softw. Eng. (ASE)*, 2022, pp. 1–13.
[63] "CodeBERT-base." 2024. [Online]. Available: https://huggingface.co/microsoft/codebert-base
[64] "CodeSearchNet." 2024. [Online]. Available: https://huggingface.co/datasets/code_search_net
[65] "StarCoderBase-15b." 2024. [Online]. Available: https://huggingface.co/bigcode/starcoderbase
[66] "StarCoder2-15B." 2024. [Online]. Available: https://huggingface.co/bigcode/starcoder2-15b
[67] "GraphCodeBERT-base." 2024. [Online]. Available: https://huggingface.co/microsoft/graphcodebert-base
[68] "CodeLlama-13B." 2024. [Online]. Available: https://huggingface.co/codellama/CodeLlama-13b-hf
[69] "CodeLlama–34B." 2024. [Online]. Available: https://huggingface.co/codellama/CodeLlama-34b-hf
[70] "CodeLlama-70B." 2024. [Online]. Available: https://huggingface.co/codellama/CodeLlama-70b-hf
[71] "PLBart-base." 2024. [Online]. Available: https://huggingface.co/uclanlp/plbart-base
[72] "CodeGen25-7B." 2024. [Online]. Available: https://huggingface.co/Salesforce/codegen25-7b-multi_P
[73] "CodeT5-base." 2024. [Online]. Available: https://huggingface.co/Salesforce/codet5-base
[74] "CodeGeeX2–6B." 2024. [Online]. Available: https://huggingface.co/THUDM/codegeex2-6b
[75] "UniXcoder-base." 2024. [Online]. Available: https://huggingface.co/microsoft/unixcoder-base
[76] "InCoder-6B." 2024. [Online]. Available: https://huggingface.co/facebook/incoder-6B
[77] "Big code models leaderboard." 2024. [Online]. Available: https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard
[78] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 1433–1443.
[79] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Proc. 35th Conf. Neural Inf. Process. Syst. Datasets Benchmarks Track*, 2021.
[80] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Findings Assoc. Comput. Linguistics: EMNLP(ACL)*, 2020, pp. 1536–1547.
[81] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2021.
[82] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics (NAACL)*, 2021, pp. 2655–2668.
[83] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. Empirical Methods Natural Lang. Process. (EMNLP)*, 2021, pp. 8696–8708.
[84] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2022, pp. 7212–7225.
[85] B. Roziere et al., "Code Llama: Open foundation models for code," 2023, *arXiv:2308.12950*.
[86] R. Li et al., "StarCoder: May the source be with you!" 2023, *arXiv:2305.06161*.
[87] Q. Zheng et al., "CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X," in *Proc. 29th Conf. Knowl. Discovery Data Mining (KDD)*, 2023, pp. 5673–5684.
[88] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," 2022, *arXiv:2203.13474*.
[89] D. Fried et al., "InCoder: A generative model for code infilling and synthesis," 2022, *arXiv:2204.05999*.
[90] M. Sourab, G. Sylvain, D. Lysandre, B. Younes, and P. Sayak, "PEFT: State-of-the-art parameter-efficient fine-tuning methods," 2024. [Online]. Available: https://github.com/huggingface/peft
[91] "PEFT." 2024. [Online]. Available: https://huggingface.co/docs/peft
[92] "Prompt-based methods." 2024. [Online]. Available: https://huggingface.co/docs/peft/conceptual_guides/prompting
[93] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proc. Conf. Empirical Methods in Natural Lang. Process. (EMNLP)*, 2021, pp. 3045–3059.
[94] X. L. Li and P. Liang, "Prefix-Tuning: Optimizing continuous prompts for generation," in *Proc. 11th Int. Joint Conf. Natural Lang. Process. (IJCNLP)*, 2021, pp. 4582–4597.
[95] X. Liu et al., "GPT understands, too," *AI Open*, vol. 5, pp. 208–215, 2024.
[96] "Adapter-based methods." 2024. [Online]. Available: https://huggingface.co/docs/peft/conceptual_guides/adapter
[97] N. Hyeon-Woo, M. Ye-Bin, and T.-H. Oh, "FedPara: Low-rank Hadamard product for communication-efficient federated learning," in *Proc. Int. Conf. Learn. Representations, ICLR*, 2021.
[98] S.-Y. Yeh, Y.-G. Hsieh, Z. Gao, B. B. Yang, G. Oh, and Y. Gong, "Navigating text-to-image customization: From Lycoris fine-tuning to model evaluation," 2023, *arXiv:2309.14859*.
[99] "IA3." 2024. [Online]. Available: https://huggingface.co/docs/peft/conceptual_guides/ia3
[100] G. Pu, A. Jain, J. Yin, and R. Kaplan, "Empirical analysis of the strengths and weaknesses of PEFT techniques for LLMS," in *Proc. Workshop Math. Empirical Understanding Found. Models*, 2023.
[101] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proc. 28th Int. Symp. Softw. Testing Anal. (ISSTA)*, 2019, pp. 31–42.
[102] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMS," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 36, 2024, pp. 10088–10115.
[103] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR)*, 2020, pp. 508–512.
[104] G. P. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software," in *Proc. 17th Int. Conf. Predictive Models Data Analytics Softw. Eng. (PROMISE)*, 2021, pp. 30–39.
[105] A. Lozhkov et al., "StarCoder 2 and the Stack v2: The next generation," 2024, *arXiv:2402.19173*.
[106] Z. Chen and M. Monperrus, "The CodRep machine learning on source code competition," 2018, *arXiv:1807.03200*.
[107] B. Baudry et al., "A software-repair robot based on continual learning," *IEEE Softw.*, vol. 38, no. 4, pp. 28–35, Jul./Aug. 2021.
[108] C. Shao and Y. Feng, "Overcoming catastrophic forgetting beyond continual learning: Balanced training for neural machine translation," 2022, *arXiv:2203.03910*.
[109] Y. Zhang, B. Kang, B. Hooi, S. Yan, and J. Feng, "Deep long-tailed learning: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 9, pp. 10795–10816, Sep. 2023.
[110] "Flower plot," 2024. [Online]. Available: http://www.ehbio.com/test/venn/VennDoc/EVennDoc/flowerplot.html

[111] "PEFT: Parameter-efficient fine-tuning of billion-scale models on low-resource hardware," 2023. [Online]. Available: https://huggingface.co/blog/peft
[112] L. Yu, D. Simig, C. Flaherty, A. Aghajanyan, L. Zettlemoyer, and M. Lewis, "MEGABYTE: Predicting million-byte sequences with multiscale transformers," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, 2023, pp. 78808–78823.
[113] M. Ivgi, U. Shaham, and J. Berant, "Efficient long-text understanding with short-text models," *Trans. Assoc. Comput. Linguistics*, vol. 11, pp. 284–299, 2023.
[114] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "VULGEN: Realistic vulnerability generation via pattern mining and deep learning," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2527–2539.
[115] C. Park, Y. Tae, T. Kim, S. Yang, M. A. Khan, L. Park, and J. Choo, "Unsupervised neural machine translation for low-resource domains via meta-learning," in *Proc. 11th Int. Joint Conf. Natural Lang. Process. (IJCNLP)*, 2021, pp. 2888–2901.
[116] S. Liu, X. Xie, J. K. Siow, L. Ma, G. Meng, and Y. Liu, "GraphSearchNet: Enhancing GNNs via capturing global dependencies for semantic code search," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2839–2855, Apr. 2023.
[117] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proc. 28th Int. Symp. Softw. Testing Anal. (ISSTA)*, 2019, pp. 19–30.
[118] Z. Li et al., "Unleashing the power of compiler intermediate representation to enhance neural program embeddings," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, 2022, pp. 2253–2265.
[119] W. Zhong et al., "StandUp4NPR: Standardizing setup for empirically comparing neural program repair systems," in *Proc. 37th Int. Conf. Automated Softw. Eng. (ASE)*, 2022, pp. 1–13.
[120] R. Widyasari et al., "BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 1556–1560.
[121] J. Xiang, X. Xu, F. Kong, M. Wu, H. Zhang, and Y. Zhang, "How far can we go with practical function-level program repair?" 2024, *arXiv:2404.12833*.
[122] C. Lee, C. S. Xia, J. Huang, Z. Zhu, L. Zhang, and M. RLyu, "A unified debugging approach via LLM-based multi-agent synergy," 2024, *arXiv:2404.17153*.
[123] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning LLMs for FL-free program repair," 2024, *arXiv:2404.08877*.
[124] S. B. Hossain et al., "A deep dive into large language models for automated bug localization and repair," 2024*, arXiv:2404.11595*.
[125] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proc. 31st Int. Symp. Softw. Testing Anal. (ISSTA)*, 2022, pp. 39–51.
[126] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, "Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment," 2023, *arXiv:2312.12148*.
[127] N. Houlsby et al., "Parameter-efficient transfer learning for NLP," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2019, pp. 2790–2799.
[128] E. B. Zaken, Y. Goldberg, and S. Ravfogel, "BitFit: Simple parameter-efficient fine-tuning for transformer-based masked language-models," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2022, pp. 1–9.
[129] Y.-L. Sung, V. Nair, and C. A. Raffel, "Training neural networks with fixed sparse masks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 34, 2021, pp. 24193–24205.
[130] A. Ansell, E. Ponti, A. Korhonen, and I. Vulić, "Composable sparse fine-tuning for cross-lingual transfer," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (ACL)*, 2022, pp. 1778–1796.
[131] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, "Towards a unified view of parameter-efficient transfer learning," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2021.
[132] Y. Mao et al., "UniPELT: A unified framework for parameter-efficient language model tuning," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics*, 2022, pp. 6253–6264.
[133] H. Zhou, X. Wan, I. Vulić, and A. Korhonen, "AutoPEFT: Automatic configuration search for parameter-efficient fine-tuning," *Trans. Assoc. Comput. Linguistics*, vol. 11, pp. 525–542, 2024.
[134] S. Liu, J. Keung, Z. Yang, F. Liu, Q. Zhou, and Y. Liao, "Delving into parameter-efficient fine-tuning in code change learning: An empirical study," 2024, *arXiv:2402.06247*.
[135] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *Proc. 38th Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 397–408.
[136] W. Zou et al., "A comprehensive evaluation of parameter-efficient fine-tuning on software engineering tasks," 2023, *arXiv:2312.15614*.

Kai Huang is currently working toward the Ph.D. degree with the Technical University of Munich. His research interests include automated program repair, large language model. His research has won the ACM SIGSOFT Distinguished Paper Award (ASE 23).

Jian Zhang received the B.S. degree from Beihang University, and the Ph.D. degree in computer science from Beihang University. Currently, he is a Research Fellow with Nanyang Technological University. His research interests mainly focus on intelligent software engineering, particularly using deep learning techniques for source code maintenance to improve software quality and security.

Xinlei Bao is currently working toward the bachelor's degree in software engineering with Beihang University, China. Her research interests include software engineering, program analysis, and deep learning.

Xu Wang received the B.Eng. and Ph.D. degrees in computer science from Beihang University, in 2008 and 2015, respectively, where he is currently an Assistant Professor with the School of Computer Science and Engineering. He was a Visiting Scholar with the Department of Computer Science, University of Chicago, from 2016 to 2017. His research interests focus on how to improve software development efficiency and software quality through AI techniques, program analysis, and algorithm optimization.

Yang Liu (Senior Member, IEEE) received the B.Comp. (Hons.) degree from the National University of Singapore (NUS), in 2005, and the Ph.D. degree from NUS and MIT, in 2010. He started his Postdoctoral work with NUS and MIT. In 2012, he joined Nanyang Technological University (NTU). Currently, he is a Full Professor and the Director of the Cybersecurity Laboratory, NTU. He specializes in software verification, security, and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. He has more than 400 publications in top tier conferences and journals. He received a number of prestigious awards, including MSRA Fellowship, TRF Fellowship, Tan Chin Tuan Fellowship, Nanyang Research Award 2019, ACM Distinguished Speaker, NRF Investigatorship, and 15 Best Paper Awards.