Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair

Guochang Li
Zhejiang University
Hangzhou, China
gcli@zju.edu.cn

Chen Zhi*
Zhejiang University
Hangzhou, China
zjuzhichen@zju.edu.cn

Jialiang Chen
Zhejiang University
Hangzhou, China
cjl99@zju.edu.cn

Junxiao Han
Hangzhou City University
Hangzhou, China
hanjx@hzcu.edu.cn

Shuiguang Deng*
Zhejiang University
Hangzhou, China
dengsg@zju.edu.cn

ABSTRACT

Automated Program Repair (APR) aims to fix bugs by generating patches. And existing work has demonstrated that "pre-training and fine-tuning" paradigm enables Large Language Models (LLMs) improve fixing capabilities on APR. However, existing work mainly focuses on Full-Model Fine-Tuning (FMFT) for APR and limited research has been conducted on the execution-based evaluation of Parameter-Efficient Fine-Tuning (PEFT) for APR. Comparing to FMFT, PEFT can reduce computing resource consumption without compromising performance and has been widely adopted to other software engineering tasks.

To fill this gap, we enhance the existing APR dataset by employing prompt engineering to create an instruction dataset, APR-Instruction, at first. Secondly, we fine-tune four pre-trained LLMs using four different PEFT methods with APR-Instruction. The best fine-tuned model fixes 58% more bugs than the state-of-the-art LLM-based APR techniques. The results also show that $(IA)^3$ improves the creativity of LLMs more effectively through fine-tuning and achieves the highest fixing capability compared to the other three PEFT methods. Thirdly, we explore the optimal configuration of PEFT hyperparameters, and assess the impact of instruction dataset size, showing that a larger number of parameters and a larger training dataset do not necessarily result in better performance for PEFT. Lastly, we analyze peak memory usage and trainable parameters to show the efficiency of PEFT.

This work provides a comprehensive exploration of PEFT on APR and suggests potentially promising directions for extension to other software engineering downstream tasks. APR-Instruction, PEFT weights, and the fine-tuning code are publicly available as open-source resources.

*Corresponding authors.

KEYWORDS

Automated Program Repair, Parameter-Effective Fine-Tuning, Large Language Model, Execution-based Evaluation

ACM Reference Format:
Guochang Li, Chen Zhi, Jialiang Chen, Junxiao Han, and Shuiguang Deng. 2024. Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695066

1 INTRODUCTION

Automated Program Repair (APR) aims to generate patches for program bugs automatically, without human intervention. Based on existing studies [4, 18, 35, 57, 69], traditional APR techniques include search-based [19, 28], constraint-based [44, 67] and template-based [27, 34], which require experts to provide prior knowledge manually. Learning-based APR techniques [8, 41], such as Neural Machine Translation (NMT) techniques [55], enhance fixing capabilities through translating bug lines into fix lines. Nowadays, LLM-based APR techniques [66] have gained significant attention and are regarded as highly promising APR techniques.

Some existing studies have employed LLMs directly on APR. For instance, Alpharepair [66] enhances fixing capabilities by directly predicting the fix line based on the provided bug line context. While zero-shot can already aids in completing certain APR tasks, Full-Model Fine-Tuning (FMFT) LLMs with domain-specific datasets can fully unlock its potential [17, 49, 60]. Recently, researchers have undertaken FMFT with APR datasets, which shows significant improvements on fixing capabilities of LLMs, enabling them to fix 46% to 164% more bugs compared to existing deep learning (DL)-based APR techniques [20].

Instruction-tuning [59] enables LLMs to align the training objectives with downstream tasks. However, instruction datasets [5, 61] of software engineering (SE) mainly consist of general code generation tasks, such nature language to program language. There is a lack of APR-specific instruction datasets for APR tasks. Although some APR datasets are available for DL-based APR techniques [72], which cannot be used for instruction-tuning due to the limited variety of instruction types. To adapt to instruction-tuning, it is necessary to enrich the instruction types, such as descriptive information, of existing datasets to build an APR instruction dataset.

FMFT demands significant computing resources which may be not practical for all users [36]. In contrast, PEFT methods [16, 30, 32, 37] have emerged as a more promising approach for limited computing resources [62]. PEFT guarantees the preservation of the core generation capability of LLMs by freezing the initial parameters and fine-tuning additional parameters, which do not exceed 1%-5% of the initial parameters [36]. Previous studies [9, 73] have demonstrated the superiority of PEFT in specific software engineering (SE) downstream tasks. However, there is currently no research investigating PEFT methods on APR.

Several SE tasks that employ PEFT, such as code completion [7], docstring generation [39], and code clone detection [73], are evaluated by Natural Language Processing (NLP) metrics like BLEU [47]. These metrics do not reliably measure the functional correctness of the generated code [33, 65]. Unlike other SE tasks, APR requires generated patches that can be merged into the original code and executed correctly. Therefore, while generated code segments may exhibit high similarity, it does not ensure that executing the two code segments will produce similar or correct results. Evaluation of the generated patches needs to be performed using execution-based methods, such as test cases. Limited research has focused on the execution-based evaluation of PEFT results, although execution-based evaluation plays a crucial role in numerous code generation tasks involving APR.

To address the above issues, this work comprehensively explores PEFT on APR, with a focus on the construction of APR-Instruction, supervised fine-tuning (SFT) with PEFT, and patch validation. To begin with, we combine prompt engineering with existing APR datasets [72] to enrich additional descriptive information, such as problem descriptions and bug causes, resulting APR-Instruction, which consists of 30,000 instructions and has been made publicly available [1]. In order to assess the effectiveness, a comparative analysis is conducted with two widely used SE instruction datasets, Code Alpaca [5] and OSS-Instruction [61]. The results of experiments provide evidence that APR-Instruction [1] holds better potential for improving fixing capabilities of LLMs, which fixes 20.1% more than the other two datasets at most.

Secondly, we conduct SFT using *LoRA* [16], *prefix-tuning* [30], *p-tuing v2* [37], $(IA)^3$ [32] on CodeLlama-7B [50], CodeLlama-13B [50], Llama-2-7B [54], DeepSeek-Coder-Base-6.7B [14] with APR-Instruction, and evaluating on Defects4J v2.0 [24], QuixBugs [31], and HumanEval-Java [20] to showcase the effectiveness and efficiency of PEFT in the single-hunk APR task. Initially, we compare fixing capabilities of PEFT with the state-of-the-art (SOTA) APR techniques, which demonstrates that PEFT can at most fix 58% more bugs than the SOTA APR techniques. Subsequently, we investigate fixing capabilities of no fine-tuning as the baseline, *LoRA* [16], *prefix-tuning* [30], *p-tuing v2* [37], $(IA)^3$ [32], and FMFT on CodeLlama-7B. The results suggest that, compared to no fine-tuning, PEFT methods enhance the fixing capabilities of LLMs, fixing 7.2%-31.3% more bugs.

Among four LLMs and four PEFT methods in this work, DeepSeek-Coder-Base-6.7B achieve the highest fixing capability following $(IA)^3$. Furthermore, we try to analyze why $(IA)^3$ achieve the best performance among four PEFT methods. Combining *pass@k* and an instance, we demonstrate that $(IA)^3$ improves the creativity of LLMs more effectively through fine-tuning compared to the other three PEFT methods, which accounts for its superior performance. PEFT methods alsoreduce around 45.6%-50.0% peak GPU memory usage compared to FMFT, which shows the efficiency of PEFT.

Finally, we conduct further investigations into two aspects aimed at improving the efficiency of PEFT: hyperparameters and the size of instruction datasets. We examine the impact of hyperparameters, such as *rank* of *LoRA* on APR, since a lower *rank* implies a reduction of trainable parameters, resulting in less memory usage. The results indicate that setting *rank* of *LoRA* to 16 already enables LLMs to achieve a relatively strong fixing capability. Increasing the *rank* blindly does not necessarily yield further improvements in fixing capabilities. We examine the impact of instruction dataset size, considering that a smaller dataset implies reduced computational resource costs. To investigate this, we employ $(IA)^3$ to fine-tune different sizes of APR-Instruction dataset, revealing $(IA)^3$ only fixes 3.67% less bugs when reducing 50% instructions. This suggests the feasibility of appropriately reducing the instruction dataset when resources are limited.

To sum up, the main contributions of this work are as follows:

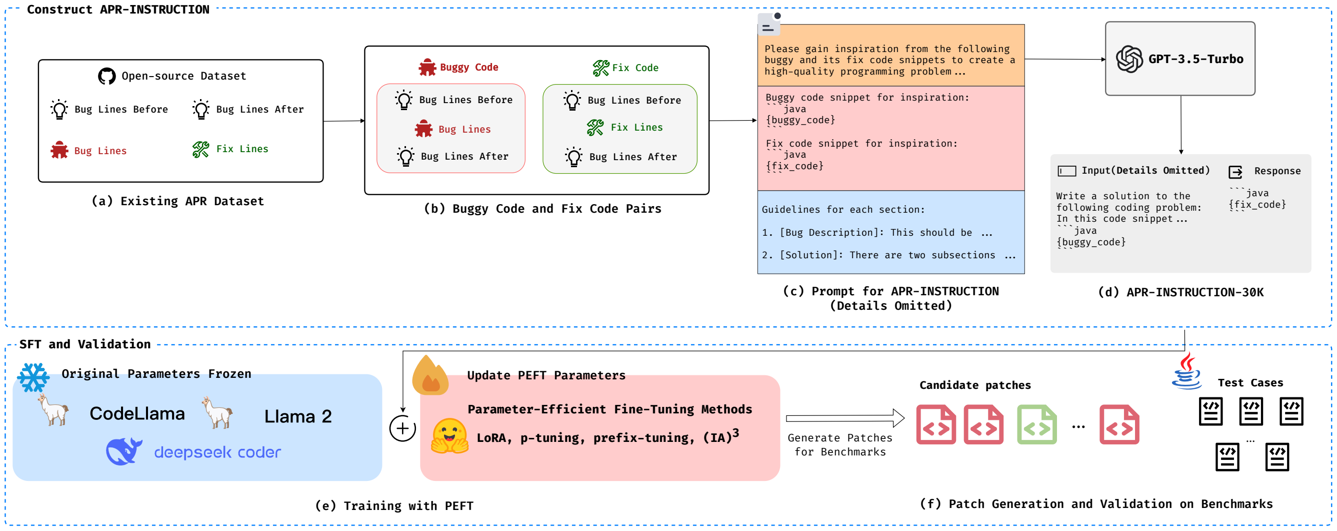
- This work constructs APR-Instruction by enhancing the existing APR training dataset by adding extra descriptions, which indeed improves fixing capabilities of LLMs on APR.
- This work shows CodeLlama-7B with all PEFT methods fixes 7.2%-31.3% more bugs than no fine-tuning, and reduce around around 45.6%-50.0% peak GPU memory usage compared to FMFT, showing the effectiveness and efficiency of PEFT on APR.
- This work shows DeepSeek-Coder-Base-6.7B with $(IA)^3$ achieves the best fixing capability, fixing 58% more bugs than the SOTA technique.
- This work investigates multiple factors that potentially influence the fixing capability of PEFT methods, including scales of LLMs, type of pre-training data, and base models.
- This work demonstrates that $(IA)^3$ achieve the best fixing capability, benifiting from its superior creativity improvement through SFT compared to the other three PEFT methods, possibly.
- This work further explores the impact of hyperparameters of PEFT and the size of instruction dataset to improve the efficiency of PEFT.
- This work, including APR-Instruction, code, and weights, has been publicly released on [1] to allow researchers to further extend it to other software engineering tasks.

Overall, this work primarily focuses on investigating the effectiveness and efficiency of PEFT on APR, which presents a comprehensive roadmap, offering complete and reproducible data. By applying the same workflow, PEFT methods have the potential to be extended to other SE tasks by leveraging existing datasets in order to enhance performance of downstream tasks with LLMs.

2 EXPERIMENT DESIGN

2.1 Overview

Figure 1 illustrates the overview of this work, which consists of three main stages: constructing APR-Instruction, supervised fine-tuning (SFT) with PEFT, and generating and validating patches.



~~~
Please gain inspiration from the following
buggy and its fix code snippets to create a
high-quality programming problem ...

Buggy code snippet for inspiration:
```java
{buggy_code}
~~~
Fix code snippet for inspiration:
```java
{fix_code}
```

Guidelines for each section:
1. [Bug Description]: This should be ...
2. [Solution]: There are two subsections ...
```

```
Write a solution to the
following coding problem:
In this code snippet ...
~~~java
{buggy_code}

```java
{fix_code}
~~~

**Figure 1: An overview of APR-Instruction constructed, supervised fine-tuning on four LLMs with four PEFT Methods, and Evaluation on three benchmarks**

In the initial stage of constructing APR-Instruction, we utilize an APR dataset previously released [72] as Figure 1 (a) shown. This dataset contains pairs of bug code and fix code as Figure 1 (b) shown, served as seed fragments. To enhance the diversity of APR-Instruction, we employ prompt engineering with ChatGPT (i.e., the GPT-3.5-turbo model with its default setting [46]) as Figure 1 (c) shown to incorporate additional descriptive information as Figure 1 (d) shown. In the second stage, we perform SFT on four LLMs with four PEFT methods based on APR-Instruction as Figure 1 (e) shown. The training weights are saved for inference on benchmarks, and for each bug code, we generate 10 candidate patches. Finally, we assess the correctness of the generated patches by test cases in benchmarks as Figure 1 (f) shown.

## 2.2 Studied Large Language Models

We select LLMs for this work on the following criteria: (1) since the currently available open-source LLMs are predominantly built with a decoder-only architecture [70], we specifically focus on selecting open-source LLMs that adhere to this structure. (2) LLMs such as Llama 2 [54] typically have at least 7 billion (7B) parameters. Hence, all LLMs in this work have sizes around 7B or larger. (3) LLMs are obtained from *Hugging Face* [2], which are publicly accessible. Finally, we select four LLMs: CodeLlama-7B [50], CodeLlama-13B [50], Llama-2-7B [54] and DeepSeek-Coder-Base-6.7B [14]

Llama 2 [54] is pre-trained on the next token prediction task and incorporates several architectural advancements compared to Llama [53], including an increased context length from 2K to 4K. Pre-training data for Llama 2 comprises a new mixture of publicly available online sources, resulting in a corpus size of 2 trillion tokens. In this work, we focus on Llama-2-7B. CodeLlama [50] is a family of LLMs designed for code generation and infilling. It is achieved by further training Llama 2 on code-specific datasets, using a dataset that comprises 500 billion tokens. CodeLlama is specifically trained on the fill-in-the-middle task, aimed at generating code that closely matches an existing prefix and suffix. The context window of CodeLlama is 16K. In this work, we focus on analyzing CodeLlama-7B and CodeLlama-13B. DeepSeek-Coder [14] undergoes pre-training with two main objectives: next token prediction and fill-in-the-middle. The training dataset for DeepSeek-Coder consists of 87% source code, encompassing 603 million files from 87 programming languages and comprising 2 trillion tokens. In this work, we concentrate on DeepSeek-Coder-Base-6.7B.

## 2.3 APR-Instruction Construction

*2.3.1 Data Source.* APR-Instruction is derived from the APR dataset shared in the previous research [72], which is obtained from commits of open-source Java projects on GitHub. Each individual fix in the dataset is treated as a separate instance, resulting in a total of 143,666 instances [72]. Previous studies on APR have predominantly utilized FMFT, and this dataset has been employed to create pairs of bug code and fix code for training [20, 72]. The format of the dataset is illustrated in Figure 1 (a). However, since instruction-tuning typically requires a diverse range of instruction types to achieve optimal results, this dataset cannot be directly used for instruction-tuning. Additional methods are necessary to construct a more varied instruction dataset.

*2.3.2 Data Construction.* Considering the cost, we employed *gpt-3.5-turbo* [46] for generating APR-Instruction. To avoid the attention of LLMs dispersed by long context, we firstly tokenize buggy codes by CodeLlama tokenizer. We only retain buggy codes with tokenized vector lengths not exceed 200. By utilizing the prompt template shown in Figure 1 (c), we employ *gpt-3.5-turbo* to generate extra descriptions of the original code, including problem description, bug description and so on. The overall process results in a dataset, APR-Instruction containing 30,000 APR instructions [1].

#### 2.3.3 *Impact of APR-Instruction.*

To demonstrate the effectiveness of APR-Instruction, a comparison is conducted on two datasets, Code Alpaca [5] and OSS-Instruction [61], which are widely used in instruction-tuning for software engineering tasks. SFT is performed on two base models, CodeLlama-7B and DeepSeek-Coder-Base-6.7B, to eliminate the influence of the base model on fixing capabilities. Additionally, in order to exclude the impact of the PEFT method on fixing capabilities, two PEFT methods, *LoRA* and *p-tuning*, are applied to DeepSeek-Coder-Base-6.7B. OSS-Instruction consists of 75,000 instructions. To mitigate the impact of data volume for SFT, a random selection of 30,000 instructions from OSS-Instruction was executed to construct the OSS-Instruction-30K for instruction-tuning. On the other hand, Code Alpaca comprises a total of 20,000 instructions, which were totally used for instruction-tuning. After SFT, LLMs generate patches on three benchmarks, which are assessed correctness by test cases. Results are presented in Table 1.

**Table 1: Number of fixed bugs by three instruction datasets, two LLMs and two PEFT methods. The best results are highlighted in red.**

| Base Model with PEFT Method | Instruction Dataset | Benchmark | | | Total Fixed Bugs |
|---|---|---|---|---|---|
| | | HumanEval-Java (pass@10) | Defects4J v2.0 (pass@10) | QuixBugs (pass@10) | |
| CodeLlama-7B *LoRA* | OSS-Instruction-30K | 97/163 | 46/217 | 28/40 | 171/420 |
| | Code Alpaca | 95/163 | 56/217 | 33/40 | 184/420 |
| | APR-Instruction | 98/163 | 75/217 | 26/40 | 199/420 |
| DeepSeek-Coder Base 6.7B *LoRA* | OSS-Instruction-30K | 109/163 | 45/217 | 34/40 | 188/420 |
| | Code Alpaca | 101/163 | 59/217 | 32/40 | 192/420 |
| | APR-Instruction | 109/163 | 92/217 | 30/40 | 231/420 |
| DeepSeek-Coder Base 6.7B *p-tuning* | OSS-Instruction-30K | 102/163 | 50/217 | 34/40 | 186/420 |
| | Code Alpaca | 96/163 | 56/217 | 30/40 | 182/420 |
| | APR-Instruction | 108/163 | 84/217 | 33/40 | 225/420 |

Table 1 demonstrates that APR-Instruction outperforms OSS-Instruction and Code Alpaca in terms of fixed bugs across two LLMs and two PEFT methods, indicating its improved fixing capabilities with instruction-tuning. Specifically, CodeLlama-7B with LoRA fixed an additional 28 bugs with APR-Instruction than OSS-Instruction with the lowest performance, and addressed 15 more bugs than Code Alpaca. Similarly, after fine-tuning, DeepSeek-Coder-Base-6.7B using LoRA with APR-Instruction achieves the highest fixing capability, which eliminate the influence of base models on the improvement of fixing capabilities. Furthermore, based on DeepSeek-Coder-Base-6.7B, instruction-tuning with p-tuning, along with APR-Instruction, demonstrates the highest fixing capability, which eliminates the influence of PEFT methods. In conclusion, APR-Instruction significantly enhances fixing capabilities through fine-tuning. To promote wider adoption by researchers, we have open-sourced APR-Instruction [1].

#### 2.3.4 *Evaluation of overlap between APR-Instruction and benchmarks.*

In conclusion, there is no overlapping between the instruction dataset APR-Instruction and three benchmarks, namely HumanEval-Java, Quixbugs, and Defects4J. On the one hand, the collection of APR-Instruction has excluded these benchmarks. APR-Instruction is constructed on an existing dataset [72], which claims the dataset is collected from Java projects created on GitHub between March 2011 and March 2018 and excludes projects that are clones of Defects4J projects or use Defects4J, ensuring no overlap with the Defects4J. Humaneval-Java project, with its first commit in 2023, is not included in the collection period of the existing dataset [72], ensuring no overlap with the existing APR dataset [72]. APR-Instruction uses GPT to add some additional description to this existing APR dataset [72] without more code added. On the other hand, prior work [20, 72] uses this existing APR dataset [72] for training and also conducts evaluation on Defects4J, QuixBugs, and Humaneval-Java, further indicating no data overlap between APR-Instruction and these three benchmarks.

#### 2.3.5 *Evaluation of APR-Instruction validity.*

APR-Instruction contains some information generated by GPT-3.5, so we conduct the evaluation for the validity of APR-Instruction. In the process of generating data, we initially produce nearly 100 samples and conduct a manual inspection, finding no anomalies. Subsequently, we generate the final set of 30k instructions. Given the large size of APR-Instruction, it is impractical to verify each instruction individually. Even if there is some inaccurate data, prior studies [15, 61] suggest that not removing noisy data can actually improve performance. The study [42] posits that low-quality data can still positively contribute to the diversity of the dataset. Therefore, we do not necessarily need to remove it. Overall, the impact of data validity is manageable and does not significantly affect the results.

## 2.4 PEFT Methods and Implementation Details

PEFT library [43] developed by *Hugging Face* team, encompasses multiple widely used PEFT methods. Additionally, open-source LLMs released on *Hugging Face*, can also be easily used through Transformers library [63]. Therefore, combining PEFT library with Transformers library provides a convenient approach for importing and then fine-tuning LLMs. As a result, this work primarily focuses on selecting four PEFT methods from PEFT library. Since all LLMs used in this work are casual language models. Following the guidance provided by PEFT library, there are three categories of PEFT methods to choose from: LoRA, prompt-based tuning, and $(IA)^3$. Finally, we select four PEFT methods, LoRA, p-tuning, prefix-tuning, and $(IA)^3$ to conduct the experiments. Figure 2 illustrates the working principles of these four PEFT methods on LLMs. Although Llama 2 uses Group Query Attention [3], to simplify the illustration, only Multi-head Attention is depicted in Figure 2.

LoRA [16] focuses on $Query$ and $Key$ matrices within attention layers, as depicted in Figure 2. LoRA incorporates additional trainable low-rank matrices to enable parameters updating. Specifically, for the initial weight matrix $W_0$, modification is performed through an additive operation $W_0 + BA$, where the rank of matrices $B$ and $A$ is considerably smaller than that of $W_0$, thereby reducing the number of trainable parameters.

Figure 2 demonstrates the architecture of two prompt-based tuning methods, p-tuning v2 [37] and prefix-tuning [30]. Unlike static prompts commonly used in traditional prompt engineering, these prompts are dynamic and learnable, allowing for greater adaptability across various tasks. In p-tuning, trainable parameters are added to the input embeddings, while in prefix tuning, they are added to all layers, represented as $P_k$ in Figure 2.

$(IA)^3$ [32] stands for "Infused Adapter by Inhibiting and Amplifying Inner Activation". $(IA)^3$ introduces three learnable vectors: $l_k$, $l_v$, and $l_{ff}$, as shwon in Figure 2. The vectors $l_k$ (for keys) and $l_v$

(for values) are used to rescale the attention keys and values, respectively, before applying the softmax. On the other hand, the vector $l_{ff}$ is employed to adjust the inner activations of the position-wise feed-forward networks. The attention mechanism with $(IA)^3$ is represented by the following formula:

$$\text{softmax}\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V)$$

Where $\odot$ denotes the element-wise multiplication, $Q$ and $K$ are the $Query$ and $Key$ matrices, $V$ is the $Value$ matrix, and $d_k$ is the dimension of $Key$.

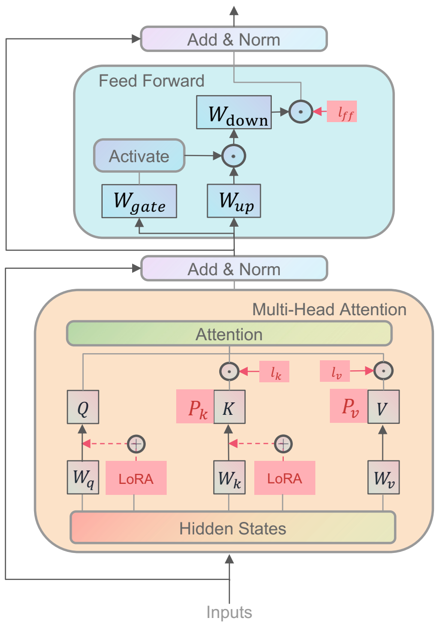


**Figure 2: Principles of LoRA, p-tuning, prefix-tuning, and $(IA)^3$ on LLMs. Red parts represents trainable modules of PEFT methods.**

To ensure the fairness in the comparison of different PEFT methods on the same base model, we have maintained consistent magnitudes of trainable parameters among different PEFT methods by adjusting hyperparameters of different PEFT methods, shown in Table 2. Several challenges prevent prevent us from ensuring an exact match of trainable parameters across different PEFT methods in practice. Firstly, for $(IA)^3$, the number of trainable parameters is only determined by the structure of the base model, indicating $(IA)^3$ is impossible to have exactly the same trainable parameters as the other PEFT methods. Secondly, achieving a specific and exact number of trainable parameters is challenging. According to LoRA documentation, the rank should be set as an integer exponent of 2, which implies that adjusting the rank does not enable precise control over the desired number of trainable parameters. Overall, except for $(IA)^3$, we ensure that the number of trainable parameters in LoRA, p-tuning, and prefix-tuning remains within the same order of magnitude. Table 2 also shows the setting of hyperparameters for different PEFT methods, taking CodeLlama-7B as an example. The setting of PEFT hyperparameters for other LLMs can be found in the source code [1].

**Table 2: PEFT Hyperparameters setting of different PEFT Methods, taking CodeLlama-7B as an example.**

| PEFT Methods | Hyper-parameters | value | Trainable parameters |
|---|---|---|---|
| $(IA)^3$ | None | None | $6.14 \times 10^5$ |
| LoRA | rank | 32 | $1.68 \times 10^7$ |
| | lora alpha | 16 | |
| | lora dropout | 0.05 | |
| p-tuning | virtual tokens | 100 | $2.14 \times 10^7$ |
| | encoder hidden size | 2048 | |
| | encoder reparameterization type | MLP | |
| prefix-tuning | encoder hidden size | 256 | $6.88 \times 10^7$ |
| | virtula tokens | 100 | |

Indeed, there are several adapter-based PEFT methods available. However, the main focus of this work is to assess the effectiveness of PEFT methods, rather than comparing and determining the best method among all PEFT methods. Additionally, this work aims to simplify the application of PEFT methods by utilizing the Transformers and PEFT libraries for SFT, thereby enhancing the reusability of these methods for other software engineering tasks.

## 2.5 Benchmarks and Evaluation Metrics of APR

*2.5.1 Three existing benchmarks.* Since APR-Instruction exclusively consists of Java instances, we specifically select APR benchmarks in Java for evaluation. In order to enable convenient comparison of fixing capabilities with previous work [20], this work specifically evaluates single-hunk bugs and utilizes the same three benchmarks as used in prior work: Defects4J [24], which comprises a collection of 217 single-hunk bugs carefully chosen by previous work [20], QuixBugs [31], which includes 40 bugs serves as a benchmark for well-known algorithm-related issues, and HumanEval-Java [20], which covers 163 single-hunk Java bugs varying from simple errors to complex logical bugs. In all benchmarks, the location of the bug liens is known and label. Importantly, HumanEval-Java demonstrates a reduction of data leakage risks between pre-training data and benchmarks. All three benchmarks provide corresponding test suites for each bug, which are used to evaluate the correctness of the generated patches.

*2.5.2 Patch generation and validation.* In experiments, LLMs generate 10 candidate patches for each example in the benchmark. Recent studies [45] on developer preferences have shown that high-quality patches ranked within the top 10, generated within an acceptable time frame of 1 hour, are considered more valuable. Consequently, we individually validate the generated patches using the corresponding test suite to assess their correctness. A large number of experiments have been conducted in this work, generating over 7,000 plausible correct patches. Manually checking all of them would be excessively labor-intensive. Then, we manually checked a sample of the patches and all generated patch files have been published and are available for further examination.

Following other studies of execution-based evaluation, we used the *pass@k* [26] to evaluate fixing capabilities. This metric generates $k$ code samples for each problem, and if any of the samples pass the unit tests, the problem is considered solved. In the subsequent experiments, we report the *pass*@10 metric for each benchmark.

## 2.6 Implementation Details

This work also employs FMFT to explore the effectiveness and efficiency of PEFT methods. Since it is not practical to directly obtain the fixing capability of no instruction-following LLMs, the benchmarks are transformed into an infilling task by employing the fill-in-the-middle task provided by CodeLlamato complete APR. In this infilling task, bug lines serve as lines that need to be completed by CodeLlama. As a result, we convert the buggy code from the three benchmarks into the infilling format depicted in Figure 3 and feed it into CodeLlamafor inference to generate patches for valuation. Additionally, during SFT, we record the usage of peak GPU memory and the number of trainable parameters in order to analyze the efficiency of PEFT in subsequent sections. Setting of trainning hyperparameters could be found in our source code [1].

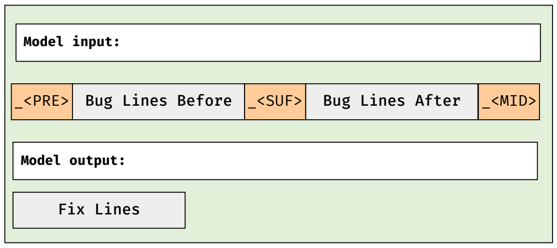


**Figure 3: Infilling template of CodeLlama-7B**

All pre-trained LLMs are downloaded from *Hugging Face*. We conduct all experiments on a server equipped with Intel (R) Xeon (R) Platinum 8358 CPU, 1TB RAM, and 80G A800 GPU running Ubuntu 18.04. Additionally, we have released SFT and inference code to facilitate reproducibility by researchers [1].

## 3 EXPERIMENT RESULTS

Firstly, we apply LoRA, p-tuning, prefix-tuning, $(IA)^3$, and FMFT on CodeLlama-7B to evaluate effectiveness of PEFT methods compared to no fine-tuning and other State-of-the-Art (SOTA) APR techniques. Secondly, we evaluate the impact of different PEFT methods and different LLMs on fixing capabilities. Lastly, we explore the influence of hyperparameters and the size of instruction dataset on the efficiency of the PEFT methods, aiming to address the following research question:

**RQ1:** How do PEFT methods effectively and efficiently improve the fixing capabilities of LLMs compared to no fine-tuning and SOTA APR techniques?

**RQ2:** How do different PEFT methods improve the fixing capabilities of LLMs fine-tuned on different base LLMs?

**RQ3:** How do PEFT hyperparameters and the size of instruction dataset affect fixing capabilities of LLMs on APR?

## 3.1 RQ1: Fixing Capabilities and Computing Resource Consumption of PEFT

Firstly, since our objective is to demonstrate the effectiveness of PEFT methods, we conduct a comparison between PEFT methods and the SOTA APR techniques provided in prior work [20]. Secondly, we present PEFT methods improve fixing capabilities of CodeLlama-7B through fine-tuning, compared to the baseline, no fine-tuning. Lastly, as PEFT emphasizes the reduction of computational resource consumption, we conduct an analysis and comparison of computing resources consumption among different fine-tuning methods, including FMFT.

*3.1.1 Experiments Setup.* In experiments, we compare the computating resource consumption of different fine-tuning methods using specific hyperparameter settings to keep external environment settings consistent. We set *epoch* to 5, *maximum sequence length* to 1000, and *learning rate* to $1e^{-4}$, while further details can be found in our source code [1]. Following the guidance of PEFT methods, we assign *rank* to 32 and *lora alpha* to 16 for LoRA, *virtual tokens* to 100, *encoder hidden size* to 2048, and *encoder reparameterization type* to *Multiple Layer Perceptron (MLP)* for p-tuning, and *virtual tokens* to 100 and *encoder hidden size* to 256 for prefix-tuning. After SFT, we transform buggy code in benchmarks into instructions to guide LLMs in inferring and generating patches, which are validated by test cases, and *pass@k* values are recorded, shown on Figure 1 (f).

**Table 3: Number of fixed bugs of DL-based[20], LLM-based[20] and PEFT methods on HumanEval-Java**

| Base Models | Fine-tuning Method | HumanEval-Java pass@1 | HumanEval-Java pass@5 | HumanEval-Java pass@10 |
|---|---|---|---|---|
| CURE [21] | FMFT | 7/163 | 15/163 | 18/163 |
| RewardRepair [68] | FMFT | 4/163 | 17/163 | 22/163 |
| Recorder [72] | FMFT | 5/163 | 7/163 | 11/163 |
| INCODER-1B [11] | FMFT | 38/163 | 54/163 | 64/163 |
| INCODER-6B [11] | FMFT | 43/163 | 62/163 | 70/163 |
| CodeLlama-7B | $(IA)^3$ | 37/163 | 87/163 | 99/163 |
| CodeLlama-13B | $(IA)^3$ | 55/163 | 89/163 | 97/163 |
| DeepSeek-Coder-Base-6.7B | $(IA)^3$ | 76/163 | 98/163 | 111/163 |

*3.1.2 Analysis of Fixing Capabilities.* We conduct a comparison of fixing capabilities between PEFT methods and the SOTA APR techniques based on previous work findings. It has been previously noted that Defects4J and QuixBugs present potential risks of data leakage [20]. Therefore, we will compare the fixing capability on HumanEval-Java with previous APR techniques. To ensure consistency, we directly utilize the experimental results from previous research [20] and present the comparison in Table 3. We select three DL-based methods, CURE [21], RewardRepair [68], and Recorder [72]. Among these, RewardRepair demonstrates the highest fixing capability by successfully addressing 22 bugs out of 163 instances. Additionally, we evaluate the two most effective LLM-based APR technique, INCODER-1B [11] and INCODER-6B [11], both fine-tuned with the APR dataset. Indeed, PEFT methods do not introduce any additional knowledge during fine-tuning compared to INCODER, as the dataset used for fine-tuning on INCODER-6B is the same one used to construct APR-Instruction.

According to the results in Table 3, INCODER-6B currently stands as the SOTA APR technique before this work, successfully fixing 70 bugs. LLMs with more parameters demonstrates enhanced fixing capability compared to DL-based methods. Additionally, Table 3 displays fixing capabilities of the three LLMs employed in this work, fine-tuned with $(IA)^3$, on Humaneval-java. With $(IA)^3$, CodeLlama-7B achieves a higher fixing capability and resolves 99 bugs. The best model in this work, DeepSeek-Coder-Base-6.7B with $(IA)^3$ fine-tuning, successfully fixes 111 bugs, which outperforms the previous SOTA, INCODER-6B, by fixing an additional 41 bugs, which is a 58% improvement. Overall, PEFT methods indeed leads to a improvement in fixing capability, enhancing it up to 58% when compared to previous SOTA APR techniques.

**Finding 1:** As Table 3 shown, the best model in this work, DeepSeek-Coder-Base-6.7B with $(IA)^3$, fixes 58% more bugs than the SOTA LLM-based APR technique, INCODER-6B with FMFT, on HumanEval-Java.

**Table 4: Number of fixed bugs on different fine-tuning methods on CodeLlama-7B. (%) represents the percentage of performance improvement compared to no fine-tuning(baseline).**

| Fine-tuning Method | Benchmark | | | Total Fixed Bugs |
|---|---|---|---|---|
| | Humaneval-java (pass@10) | Defects4j v2.0 (pass@10) | Quixbugs (pass@10) | |
| No Fine-tuning | 73/163 | 75/217 | 18/40 | 166/420(0%) |
| FMFT | 52/163 | 38/217 | 15/40 | 105/420(-36.7%) |
| LoRA | 98/163 | 75/217 | 26/40 | 199/420(+19.9%) |
| p-tuning | 86/163 | 85/217 | 25/40 | 196/420(+18.1%) |
| prefix-tuning | 81/163 | 73/217 | 24/40 | 178/420(+7.2%) |
| $(IA)^3$ | 99/163 | 95/217 | 24/40 | 218/420(+31.3%) |

Table 4 is evident that PEFT methods have significantly improved the fixing capabilities compared to the baseline, no fine-tuning on CodeLlama-7B. $(IA)^3$ outperform the baseline by fixing 26 additional bugs on HumanEval-Java, 20 additional bugs on Defects4J, and 6 additional bugs on QuixBugs. This clearly demonstrates the effectiveness of PEFT methods in enhancing fixing capabilities on APR. However, it is important to note that FMFT performed inferior to both no fine-tuning and PEFT methods. This disparity can be attributed to the limited size of APR-Instruction, which contributes to overfitting issues with only 5 epochs. In an attempt to address this, we reduce the number of epochs to 3, but the improvement in fixing capabilities remained insufficient. Otherwise, during fine-tuning, we conduct *Adam* optimizer to dynamically adjust the learning rate, but do not observe any benefits. Since FMFT requires over 300 times more trainable parameters than PEFT, the larger parameter size in FMFT may increase the risk of overfitting, which is consistent with the challenges observed in other studies when working with limited datasets. Overall, compared to no fine-tuning, the fixing capabilities of CodeLlama-7B with four PEFT methods have indeed improved, confirming the effectiveness of PEFT methods on APR.

*3.1.3 Analysis of Computing Resource Consumption.* PEFT methods prioritize achieving performance improvement that is comparable to FMFT while aiming to reduce computational resource consumption, including peak GPU memory usage and the number of trainable parameters. Figure 4 illustrates the relationship between computation resource consumption and fixing capabilities for different fine-tuning methods on CodeLlama-7B. FMFT consumes the largest number of trainable parameters and the highest peak GPU memory, 126.56GB where $(IA)^3$ consumes 68.91GB and LoRA consumes 63.30GB, 45.6%-50.0% less than FMFT, demonstrating the efficiency of PEFT methods in enhancing fixing capabilities on APR. Intuitively, having fewer trainable parameters results in lower peak

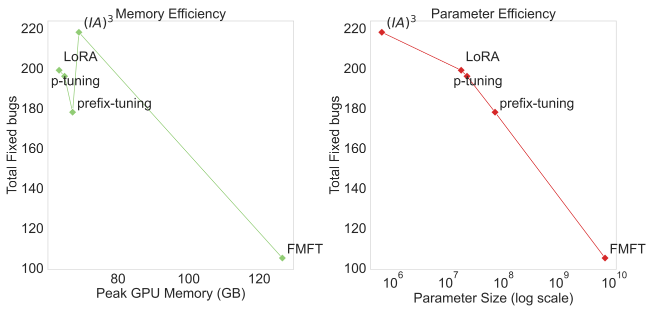


**Figure 4: Efficiency of memory and parameters with different fine-tuning methods on CodeLlama-7B**

GPU memory consumption, as observed in the LoRA, p-tuning, and prefix-tuning methods, as Fiure 4 shown. However, $(IA)^3$ has the fewest trainable parameters but consumes the most memory. This can be attributed to the principles of $(IA)^3$ discussed in the previous section 2, where it introduces three additional scaling vectors, necessitating the use of three separate loss functions to update these vectors. This can lead to the need for storing more intermediate gradients, thus increasing memory usage. Nevertheless, $(IA)^3$ only requires approximately 8.86% more memory compared to LoRA, while achieving 9.55% more fixed bugs, showing the effectiveness and efficiency.

**Finding 2:** CodeLlama-7B fine-tuned with four PEFT methods has shown significantly improved fixing capabilities compared with no fine-tuning (fixing 7.2%-31.3% more fixed bugs) and reduces consumption of computational resources (45.6%-50.0% less peak GPU memory usage and around 97-10,00 times less trainable parameters) compared to FMFT.

## 3.2 RQ2: Fix Capabilities in Different PEFT Methods Based on Different LLMs

We conduct extensive experiments to investigate the factors that contribute to the improvement of fixing capabilities on CodeLlama-7B, CodeLlama-13B, Llama-2-7B, and DeepSeek-Coder-Base-6.7B, using four PEFT methods, including LoRA, p-tuning, prefix-tuning, and $(IA)^3$. Results are presented in Table 5.

*3.2.1 CodeLlama-7B vs CodeLlama-13B.* In terms of the number of total fixed bugs, both CodeLlama-7B and CodeLlama-13B demonstrate the highest fixing capability using $(IA)^3$. Notably, CodeLlama-13B fixes 4 more additional bugs than CodeLlama-7B, summarized in Table 5. This finding aligns with the expectation that larger LLMs contribute to stronger fixing capabilities. However, we observe that, after applying p-tuning and prefix-tuning, the fixing capability of CodeLlama-13B is not as effective as that of CodeLlama-7B when using the same PEFT methods. Given the number of trainable parameters for p-tuning on CodeLlama-13B is 1.2 times that of CodeLlama-7B, while for prefix-tuning, it is 1.5 times. Hence, we consider that the decline in the fixing capability of p-tuning and prefix-tuning on CodeLlama-13B can be attributed to the increase in trainable parameters without a corresponding

**Table 5: Number of fixed bugs on CodeLlama-7B, CodeLlama-13B, Llama-2-7B, and DeepSeek-Coder-Base-6.7B, fine-tuned with LoRA, p-tuning, prefix-tuning, and $(IA)^3$. Red parts the highest fixing capability of different PEFT methods in the same base models, whicle blod indicates the highest fixing capability of the same PEFT method across distinct base models.**

| Base Model | PEFT Method | Benchmark | | | Total Fixed Bugs |
|---|---|---|---|---|---|
| | | HumanEval-Java (pass@10) | Defects4J v2.0 (pass@10) | QuixBugs (pass@10) | |
| CodeLlama-7B | LoRA | 98/163 | 75/217 | 26/40 | 199/420 |
| | p-tuning | 86/163 | 85/217 | 25/40 | 196/420 |
| | prefix-tuning | 81/163 | 73/217 | 24/40 | 178/420 |
| | $(IA)^3$ | 99/163 | 95/217 | 24/40 | 218/420 |
| CodeLlama-13B | LoRA | 94/163 | 86/217 | 26/40 | 206/420 |
| | p-tuning | 86/163 | 73/217 | 22/40 | 181/420 |
| | prefix-tuning | 16/163 | 25/217 | 11/40 | 52/420 |
| | $(IA)^3$ | 97/163 | 97/217 | 28/40 | 222/420 |
| DeepSeek-Coder Base 6.7B | LoRA | 109/163 | 92/217 | 30/40 | **231/420** |
| | p-tuning | 108/163 | 84/217 | 33/40 | **225/420** |
| | prefix-tuning | 98/163 | 73/217 | 30/40 | **201/420** |
| | $(IA)^3$ | 111/163 | 98/217 | 34/40 | **243/420** |
| Llama-2-7B | LoRA | 24/163 | 12/217 | 2/40 | 38/420 |
| | p-tuning | 10/163 | 10/217 | 1/40 | 21/420 |
| | prefix-tuning | 12/163 | 3/217 | 2/40 | 17/420 |
| | $(IA)^3$ | 17/163 | 3/217 | 2/40 | 22/420 |

increase epoch, still 5 , which means LLMs may not be trained sufficiently. To further investigate this, additional experiments are conducted, and the results are presented in Table 6.

Table 6 illustrates that increasing the epoch to 8 actually results in a decrease in the fixing capability for LoRA, while we will not continue to increase epoch. This indicates that 5 epochs may be sufficient, given that the parameter size of LoRA is smaller than that of p-tuning and prefix-tuning. In the case of p-tuning, when the epoch is increased to 10, CodeLlama-13B fixes 8 more bugs in HumanEval-Java, 11 more bugs in Defects4J, and 14 more bugs in QuixBugs, totaling 33 more bugs. On the other hand, prefix-tuning fixes 57 more bugs in HumanEval-Java, 26 more bugs in Defects4J, and 7 more bugs in QuixBugs, for a total of 100 additional bugs. However, the number of fixed bugs with prefix-tuning on CodeLlama-13B is still fewer than on CodeLlama-7B. It is possible that an epoch of 10 may still be insufficient, considering that prefix-tuning utilizes a greater number of trainable parameters compared to other PEFT methods. Due to limited computing resources, we will not delve further into this problem, although we have shown epoch affects the fixing capability of larger scale LLMs. Another possible factor is that the size of APR-Instruction is too small for such a large LLM, which implies that we need more data. Therefore, when employing a larger LLM for fine-tuning with p-tuning and prefix-tuning, it is advisable to appropriately increase the epoch to ensure adequate learning from the dataset.

**Finding 3:** CodeLlama-13B with LoRA, $(IA)^3$ and p-tuning fixes 7, 4, 18 more bugs than CodeLlama-7B with the same PEFT mthods. This demonstrates that increasing the scale of base models with PEFT can improve the fixing capability on APR. However, when using PEFT methods that require more parameters, such as p-tuning and prefix-tuning, we suggest appropriately increasing epoch ensures LLMs undergo sufficient training.

**Table 6: Number of fixed bugs by increasing epoch on CodeLlama-13B**

| PEFT Methods | epoch | Benchmarks | | |
|---|---|---|---|---|
| | | HumanEval-Java (pass@10) | Defects4J v2.0 (pass@10) | QuixBugs (pass@10) |
| LoRA | 5 | 94/163 | 86/217 | 26/40 |
| LoRA | 8 | 89/163(-5) | 82/217(-4) | 27/40(+1) |
| p-tuning | 5 | 86/163 | 73/217 | 22/40 |
| p-tuning | 10 | 94/163(+8) | 84/217(+11) | 36/40(+14) |
| prefix-tuning | 5 | 16/163 | 25/217 | 11/40 |
| prefix-tuning | 10 | 73/163(+57) | 61/217(+26) | 18/40(+7) |

### 3.2.2 *CodeLlama-7B vs Llama-2-7B.*

Table 5 clearly demonstrates that, even after fine-tuning, Llama-2-7B continues to exhibit the lowest fixing capability. This outcome is unexpected, given that both Llama-2-7B and CodeLlama-7B have the same parameter size. The primary contributing factor is the significant discrepancy between the text data used for pre-training Llama-2-7B and the code data utilized for instruction-tuning. This finding emphasizes fine-tuning datasets of PEFT should be align with the pre-training data of LLMs in order to effectively enhance fixing capabilities on APR, as well as other code generation tasks.

**Finding 4:** Llama-2-7B demonstrates the lowest fixing capability. We suggest to employ code language models when utilizing PEFT for software engineering tasks.

### 3.2.3 *CodeLlama-7B vs DeepSeek-Coder-Base-6.7B.*

According to Table 5, DeepSeek-Coder-Base-6.7B with $(IA)^3$ demonstrates superior fixing capability, outperforming CodeLlama-7B with $(IA)^3$ by fixing additional 25 bugs, which is also the highest fixing capability among all experiments conducted in this work. It is worth noting that both CodeLlama-7B and DeepSeek-Coder-Base-6.7B have similar parameter sizes and the same model architecture. Hence, with the same parameter size, different LLMs exhibit varying degrees of improvement in the fixing capability through PEFT on APR. Furthermore, DeepSeek-Coder-Base-6.7B fixes 21 more bugs than CodeLlama-13B, while the scale of DeepSeek-Coder-Base-6.7B is smaller than that of CodeLlama-13B. In fact, four PEFT methods in this work achieved the highest fixing capability on DeepSeek-Coder-Base-6.7B, as Table 5 **Bold Text** shows. This finding emphasizes the importance of selecting the appropriate base model when implementing the PEFT method. With the right choice, LLMs with a small scale can preform better than a larger-scale LLM.

**Finding 5:** DeepSeek-Coder-Base-6.7B with $(IA)^3$ exhibits the highest fixing capability, successfully fixing 243 bugs totally, which demonstrates the fixing capability of PEFT methods relies on the selection of a suitable base model. We suggest utilizing DeepSeek-Coder-Base-6.7B for APR and other SE tasks when implementing PEFT, may be a good starting point.

### 3.2.4 *Impact of PEFT Methods.*

CodeLlama-7B, CodeLlama-13B, and DeepSeek-Coder-Base-6.7B all demonstrate the highest fixing capability with $(IA)^3$ from Table 5. $(IA)^3$ also costs minimal parameter among four PEFT methods. Furthermore, $(IA)^3$ exhibits robustness as it does not require the configuration of any hyperparameters. Additionally, our experiments show that LoRA, closely following $(IA)^3$, not only enhances the fixing capability but also involves a relatively lower number of trainable parameters, which also contributes to the popularity of LoRA.

Furthermore, we wonder to gain a deeper understanding of the significant enhancement of the fixing capability by $(IA)^3$. However, the original papers on PEFT methods do not explicitly provide proof of the performance improvement. Fortunately, this work focuses on a specific scenario, APR, which enables us to analyze the potential factors contributing to improving fixing capability by $(IA)^3$, at least for APR. By plotting the relationship between *pass@k* and the number of fixed bugs for different PEFT methods, as shown in Figure 5, we have made interesting observations. Notably, $(IA)^3$ consistently exhibits lower fixing capabilities at *pass*@1 compared to other PEFT methods. For example, on Defects4J, we observe that $(IA)^3$ performs the lowest fixing capability at *pass*@1 when using CodeLlama-7B. However, starting at *pass*@5, $(IA)^3$ surpasses the other three PEFT methods and continues to improve until *pass*@10, ultimately achieving the best performance. This same trend is also observed on CodeLlama-13B and DeepSeek-Coder-Base-6.7B.

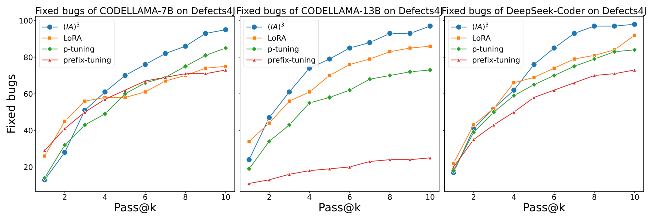
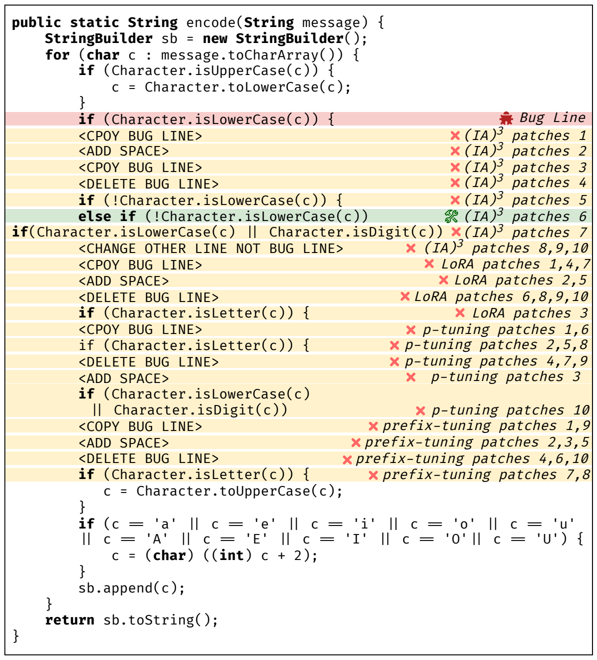


**Figure 5: *pass@k* on Defects4J with different PEFT methods**



```
public static String encode(String message) {
    StringBuilder sb = new StringBuilder();
    for (char c : message.toCharArray()) {
        if (Character.isUpperCase(c)) {
            c = Character.toLowerCase(c);
        }
        if (Character.isLowerCase(c)) {                        Bug Line
        <CPOY BUG LINE>                                  ✗ (IA)^3 patches 1
        <ADD SPACE>                                      ✗ (IA)^3 patches 2
        <CPOY BUG LINE>                                  ✗ (IA)^3 patches 3
        <DELETE BUG LINE>                                ✗ (IA)^3 patches 4
        if (!Character.isLowerCase(c)) {                 ✗ (IA)^3 patches 5
        else if (!Character.isLowerCase(c))                (IA)^3 patches 6
if(Character.isLowerCase(c) || Character.isDigit(c)) ✗ (IA)^3 patches 7
        <CHANGE OTHER LINE NOT BUG LINE>             ✗ (IA)^3 patches 8,9,10
        <CPOY BUG LINE>                                 ✗ LoRA patches 1,4,7
        <ADD SPACE>                                       ✗ LoRA patches 2,5
        <DELETE BUG LINE>                            ✗ LoRA patches 6,8,9,10
        if (Character.isLetter(c)) {                        ✗ LoRA patches 3
        <CPOY BUG LINE>                               ✗ p-tuning patches 1,6
        if (Character.isLetter(c)) {                ✗ p-tuning patches 2,5,8
        <DELETE BUG LINE>                           ✗ p-tuning patches 4,7,9
        <ADD SPACE>                                   ✗  p-tuning patches 3
        if (Character.isLowerCase(c)
          || Character.isDigit(c))                    ✗ p-tuning patches 10
        <COPY BUG LINE>                          ✗ prefix-tuning patches 1,9
        <ADD SPACE>                            ✗ prefix-tuning patches 2,3,5
        <DELETE BUG LINE>                     ✗ prefix-tuning patches 4,6,10
        if (Character.isLetter(c)) {             ✗ prefix-tuning patches 7,8
           c = Character.toUpperCase(c);
        }
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'
        || c == 'A' || c == 'E' || c == 'I' || c == 'O'|| c == 'U') {
            c = (char) ((int) c + 2);
        }
        sb.append(c);
    }
    return sb.toString();
}
```

Figure 6: Instance of HumanEval-Java with different PEFT methods. This function represents an encoding method that involves swapping the case of all letters and replacing vowels in the message with the letter that appears two places ahead in the English alphabet.

Figure 6 presents an instance, *ENCODE* of HumanEval-Java where only $(IA)^3$ successfully fixes the bug, while the other three PEFT methods failed to fix it on DeepSeek-Coder-Base-6.7B. The bug line is highlighted in red in Figure 6. Figure 6 illustrates ten candidate patches from each PEFT method, To aid understanding, some patches are represented as fixing behaviors such as <COPY BUG LINE>, <DELETE BUG LINE>, <ADD SPACE> and so on. Figure 6 shows that $(IA)^3$ adopts a more conservative approach in the initial four patches, making minimal changes to the bug line, such as <COPY BUG LINE>. This also explains why $(IA)^3$ does not fix a significant number of bugs from pass@1 to pass@5. From the fifth patch, $(IA)^3$ employs a more extreme strategy by making substantial modifications to the bug line to explore the correct patch. The correct fix is accomplished in the sixth patch, and in the seventh patch, $(IA)^3$ continues to creatively modify the bug line. Starting from the eighth patch, $(IA)^3$ gradually extends modifications to lines outside of the bug line to complete the fix.

For LoRA shown in Figure 6, most patches tend to make minimal changes to the bug line. It only modifies the bug line in the third patch and remains relatively conservative in the subsequent fixes. This aligns with the curve of Figure 5, demonstrating that the improvement of LoRA in the number of fixed bugs from pass@6 to pass@10 is not as significant as from pass@1 to pass@6. Given that this study primarily addresses single-hunk bugs, a more conservative fixing strategy may yield relatively fine results. P-tuning introduces significant modifications to the bug line in the 2nd, 3rd, 5th, and 8th patch, while adopting a conservative approach in the other patches. Although p-tuning enhances the creativity of LLMs, the improvement remains limited, resulting in the similarity of the 2nd, 5th, and 8th patch. Prefix-tuning only modifies the bug line in the 7th and 8th patch, but makes the same changes. The other patches generally leave the bug line unchanged. As a result, previous experiments indicate that prefix-tuning has the lowest fixing capability. From Figure 5 and the instance in Figure 6, we hypothesize that $(IA)^3$ enhances the creativity of LLMs during fine-tuning which aims to generate more diverse outputs to explore more results as much as possible, thereby increasing the likelihood of generating the correct patch. However, it is essential to note that this conclusion is based on experimental observations and may not necessarily hold theoretical validity.

In Figure 7, we present peak GPU memory usage of the four models during the training of the four PEFT methods. Among the applied fine-tuning methods, the peak GPU memory usage for the 7B model consistently remained below 70G, while the peak GPU memory consumption for fine-tuning the 13B model reached approximately 120G. Among the four evaluated PEFT methods, LoRA demonstrated the lowest memory consumption, followed by prefix-tuning and p-tuning, with $(IA)^3$ tending to exhibit the highest memory usage. Among 7B-scale LLMs, the difference in

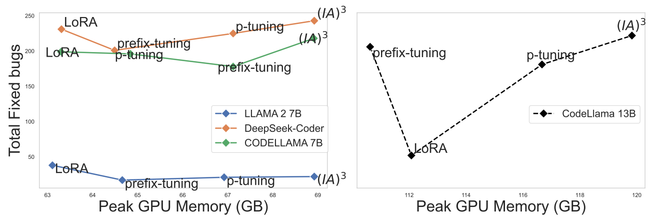


Figure 7: Peek GPU Memory on CodeLlama-7B, CodeLlama-13B, Llama-2-7B and DeepSeek-Coder-Base-6.7B

memory consumption between PEFT methods with the lowest and highest usage is about 4GB.

Finding 6: $(IA)^3$ achieves the highest fixing capability among all LLMs by improving the diversity of outputs, followed by LoRA, p-tuning, and prefix-tuning. We suggest leverage $(IA)^3$ and LoRA for fine-tuning as the first attempt on APR.

3.3 RQ3: Exploration of Hyperparameters Setting and Sizes Setting of Instruction Dataset

3.3.1 Impact of Hyperparameters Setting. When using PEFT methods, it is often necessary to configure specific hyperparameters to adjust the number of trainable parameters or the PEFT architecture. Examples of hyperparameters include *rank* of LoRA, *virtual tokens* of prefix-tuning, *encoder hidden size* of p-tuning and so on. While $(IA)^3$ is the most effective PEFT method, but it lacks adjustable hyperparameters. However, LoRA, another popular PEFT method, exhibits the second fixing capability only to $(IA)^3$. Therefore, due to limited computational resources, we conduct experiments solely on CodeLlama-7B to explore *rank* of LoRA.

Table 7: Number of fixed bugs with different hyperparameters setting of LoRA *rank* on CodeLlama-7B

LoRA (rank)	Benchmark			Total Fixed Bugs
	HumanEval-java (pass@10)	Defects4J v2.0 (pass@10)	QuixBugs (pass@10)	
8	90/163	74/217	25/40	189/420
16	99/163	87/217	26/40	212/420
32	98/163	75/217	26/40	199/420
64	96/163	78/217	23/40	197/420
128	98/163	78/217	26/40	202/420

The guidelines of LoRA [16] suggest that the maximum *rank* should not exceed 256. As a result, we set the maximum *rank* to 128 and examined four different rank values: 8, 16, 32, and 64. We fix the other hyperparameters of LoRA, *LoRA alpha* and *LoRA dropout* at 16 and 0.05, respectively. The results are presented in Table 7. The *rank* setting of 16 demonstrates the highest fixing capability, fixing 13 more bugs compared to the experiment where the *rank* is set to 32, as conducted previously in this work. As shown in 7, the fixing capability initially increases and then decreases as the *rank* increases. Thus, solely increasing the number of trainable parameters for PEFT does not necessarily enhance effectiveness.

Finding 7: Adjusting hyperparameters to increase the number of trainable parameters does not guarantee improvement. We suggest it maybe a good choice to set *rank* to 16 of LoRA as the initial configuration.

3.3.2 Impact of Instruction Dataset Size. Using PEFT methods always entails limited computing resources. Consequently, we conduct an analysis to determine the impact of reducing the size of the training data on the fixing capabilities of PEFT methods, which is a way to reduce computing resource consumption. We select $(IA)^3$ and perform fine-tuning on CodeLlama-7B, which does not require the configuration of hyperparameters. APR-Instruction consists of 30K instructions. So, we conduct fine-tuning using 5K, 10K, 15K, 20K, 25K, and 30K instructions, respectively. The experimental results shown in Table 8 clearly indicate that the largest size, 30K, corresponds to the largest number of fixed bugs, validating the intuition that a larger dataset allows LLMs to acquire more knowledge and improve its fixing capability. This validation also suggests the effectiveness of APR-Instruction.

Table 8: Number of fixed bugs with different sizes of APR-Instruction on CodeLlama-7B with $(IA)^3$. (%) represents the percentage of performance degradation compared to the best result, highlighted by red.

Size of Dataset	Benchmark			Total Fixed Bugs
	HumanEval-java (pass@10)	Defects4J v2.0 (pass@10)	QuixBugs (pass@10)	
5K	96/163	85/217	27/40	208/420 (-4.59%)
10K	96/163	90/217	26/40	212/420 (-2.75%)
15K	93/163	93/217	24/40	210/420 (-3.67%)
20K	94/163	92/217	24/40	210/420 (-3.67%)
25K	100/163	90/217	24/40	214/420 (-1.83%)
30K	99/163	95/217	24/40	218/420 (0%)

Additionally, we found that when we reduced the dataset size to 25K, 16.7% less than the full-size 30K, the fixing capability of LLMs only degraded by 1.83%. Even when we further reduced the dataset size to 5K, 83.3% less than the full-size 30K, the model fixes only 4.59% fewer bugs. In situations where computational resources are limited, fine-tuning with $(IA)^3$ on a 5K dataset can still yield reasonably good results. This highlights the advantage of PEFT in resource-constrained scenarios, as it allows for minimizing training parameters and data while maintaining a fine performance.

Finding 8: The improvement in the fix capability of $(IA)^3$ does not significantly increase with a substantial expansion of the instruction dataset size. To balance computational resource consumption and the enhancement of fixing capability, we suggest researchers to start fine-tuning from a smaller-sized dataset and gradually increasing its size until the improvement in fix capability becomes insignificant.

4 DISCUSSION AND THREATS TO VALIDITY

This work mainly focuses the APR task in Java. Regrettably, no attempts have been made to extend these tasks to multiple programming languages, thereby limiting the applicability of the current methods to a broader range of software engineering challenges.

Moreover, our concentration primarily lies on single-hunk APR tasks, neglecting to explore the fixing capabilities of PEFT in the multi-hunk APR task, which consequently leaves gaps in covering the entirety of APR scenarios. The hyperparameter settings employed for p-tuning and prefix-tuning in this work cannot guarantee optimality, suggesting a potential for improvement in certain experimental results. However, finding the optimal hyperparameters necessitates extensive experiments, rendering it challenging to determine the best configuration due to resource limitations. The patches passing test cases of benchmarks are "plausible" instead of "correct", caused by not ensuring complete line coverage of test cases, a common issue in software testing. Additionally, there exist quantization methods for PEFT, such as QLoRA [10], that can further reduce computational resource consumption. Nevertheless, these quantization methods often require additional environmental configurations, which have not been explored in this work. Nonetheless, quantization remains an exceptionally promising technique for further reducing resource consumption.

5 RELATED WORK

5.1 Parameter-Efficient Fine-Tuning for Software Engineering

"Pre-training and fine-tuning" [71] enables LLMs to quickly adapt to SE tasks and improve the performance of LLMs. However, the parameter size of current LLMs is much larger, often reaching hundreds of billions, compared to previous models, which leads to significant computational resource consumption with FMFT. To reduce the resource consumption in fine-tuning, various techniques have been proposed, and one popular strategy is PEFT. Existing works have employed PEFT on SE tasks. The work [73] proves the effectiveness PEFT methods by evaluating five PEFT methods on eight PTMs and four SE downstream tasks. Another work [62] focuses on the effectiveness of PEFT methods on code generation, which illustrates the practicality of PEFT under a limited resources scenario, effectively mitigating the reliance on large and expensive computational infrastructures. The work [56] explores parameter-efficient fine-tuning techniques for specializing LLMs for code search and code summarization, finding that parameter-efficient fine-tuning outperforms in-context learning. CodePrompt [9] further propose a new PEFT method for SE tasks, to boosts code generation performance. The work [51] explores the impact of selectively freezing layers of the model. The evaluation of these studies primarily relies on similarity-based Exact Match and CodeBLEU metrics, which fail to capture the code actual executability.

5.2 LLM-based Automated Program Repair

With the advancements in pre-trained LLMs, recent research has begun exploring their usage on APR. An earlier study [65] comprehensively evaluates the performance of nine LLMs on APR tasks using various input forms. Compared to traditional APR techniques [6, 12, 13, 23, 25, 38, 64], LLMs have demonstrated distinct advantages, particularly in their ability to fix bugs without the need for manual provision of prior knowledge. Prior to the emergence of GPT, several works [22, 48, 52, 58] have proposed fine-tuning LLMs for APR, although the size of these LLMs does not exceed 7B. The study [65] further explains that employing models with larger parameters can enhance their repair ability. In a different research work [20], aside from directly employing LLMs for APR, ten Code Language Models (CLMs) were fine-tuned using APR datasets. The evaluation conducted on Defects4J, QuixBugs, and Humaneval-Java benchmarks demonstrates that CLMs, after fine-tuning with FMFT, exhibit improved fixing capabilities. These two studies emphasize the advantages of LLMs in APR tasks. However, LLMs used in these works were models released before 2022, with a maximum parameter size of 6B. The introduction of *gpt3.5* in 2022 brought forth newly released LLMs that demonstrate even stronger performance. Several LLMs specifically designed for code-related tasks, such as StarCoder [29], CodeLlama [50], WizardCoder [40], and DeepSeek-coder [14], have showcased impressive capabilities.

6 CONCLUSION

This work demonstrates the effectiveness and efficiency of PEFT in enhancing the fixing capability of LLMs on APR. Compared to the SOTA APR techniques, the best model presented in this work successfully fixes 58% more bugs. Four LLMs and four PEFT methods are examined, resulting in sixteen sets of PEFT weights released soon. The impact of different base models and PEFT methods on the fixing capability is evaluated using three benchmarks. Specifically, an instance is provided to illustrate how $(IA)^3$ enhancing the creativity of LLM, outperformed the other three PEFT methods. Additionally, this work delves into the investigates of configuring hyperparameters of PEFT and reducing training data size to minimize computational resource consumption, while preserving the fixing capability of LLMs. Given the increasing scale of LLMs, we suggest researchers to consider employing PEFT for fine-tuning with limited computing resources. All datasets, code, and weights in this work will be available on [1]. Researchers can refer to the workflow outlined in this work and apply PEFT to other software engineering tasks, leveraging existing datasets, conveniently. In conclusion, we present a comprehensive roadmap for the application of Parameter-Effective Fine-Tuning on Automated Program Repair, highlighting its potential extension to other domains.

ACKNOWLEDGMENTS

This research is supported by Open Project of Key Laboratory of Industrial Software Engineering and Application Technology, Ministry of Industry and Information Technology (HK202300064), the National Science Foundation of China under Grants U20A20173, 62125206 and 62402433, the National Key R&D Program of China (2022YFF0902702) and the Key R&D Program of Ningbo (2023Z235). This research is also supported in part by the Zhejiang Provincial Natural Science Foundation of China under Grant LQ24F020019.

REFERENCES

[1] 2024. https://github.com/zjulgc/llmpeft4apr.

[2] 2024. https://huggingface.co.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).

[4] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 907–918.

[5] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

[6] Liushan Chen, Yu Pei, and Carlo A Furia. 2020. Contract-based program repair without the contracts: An extended study. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2841–2857.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[9] YunSeok Choi and Jee-Hyong Lee. 2023. CodePrompt: Task-agnostic prefix tuning for program and language generation. In *Findings of the Association for Computational Linguistics: ACL 2023*. 5282–5297.

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).

[11] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[12] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[14] Daya Guo, Qihao Zhu, and Dejian Yang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. https://arxiv.org/abs/2401.14196

[15] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2023. Unnatural Instructions: Tuning Language Models with (Almost) No Human Labor. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

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

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

[18] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *arXiv preprint arXiv:2303.18184* (2023).

[19] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

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

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

[22] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[23] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*. Springer, 226–238.

[24] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[25] Alireza Khalilian, Ahmad Baraani-Dastjerdi, and Bahman Zamani. 2021. CGenProg: Adaptation of cartesian genetic programming with migration and opposite guesses for automatic repair of software regression faults. *Expert Systems with Applications* 169 (2021), 114503.

[26] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).

[27] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[28] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[29] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[30] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).

[31] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[32] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.

[33] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[34] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 102–113.

[35] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 615–627.

[36] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, and Yihan Liao. 2024. Delving into Parameter-Efficient Fine-Tuning in Code Change Learning: An Empirical Study. *arXiv preprint arXiv:2402.06247* (2024).

[37] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602* (2021).

[38] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[39] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[40] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=UnUwSIgK5W

[41] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[42] Adyasha Maharana, Prateek Yadav, and Mohit Bansal. 2024. $\mathbb{D}^2$ Pruning: Message Passing for Balancing Diversity & Difficulty in Data Pruning. In *The Twelfth International Conference on Learning Representations*.

[43] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[44] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[45] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 2228–2240.

[46] OpenAI. 2023. https://openai.com/blog/chatgpt.

[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[48] Rishov Paul, Md Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna CS Santos. 2023. Enhancing Automated Program Repair through Fine-tuning and Prompt Engineering. *arXiv preprint arXiv:2304.07840* (2023).

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[50] Baptiste Rozière, Jonas Gehring, and Fabian Gloeckle. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[51] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.

[52] Atsushi Shirafuji, Md Mostafizer Rahman, Md Faizul Ibne Amin, and Yutaka Watanobe. 2023. Program repair with minimal edits using codet5. In *2023 12th International Conference on Awareness Science and Technology (iCAST)*. IEEE, 178–184.

[53] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[54] Hugo Touvron, Louis Martin, and Kevin Stone. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[55] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

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

[57] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 968–980.

[58] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[59] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *ACL*.

[60] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[61] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. (2024).

[62] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462* (2023).

[63] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[64] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 354–366.

[65] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

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

[67] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[68] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*. 1506–1518.

[69] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.

[70] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, and Yun Yang Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *arXiv preprint arXiv:2405.01466* (2024).

[71] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[72] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 341–353.

[73] Wentao Zou, Qi Li, Jidong Ge, Chuanyi Li, Xiaoyu Shen, Liguo Huang, and Bin Luo. 2023. A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Software Engineering Tasks. *arXiv preprint arXiv:2312.15614* (2023).