The Impact of Fine-tuning Large Language Models on Automated Program Repair

Roman Macháček*
University of Bern
Bern, Switzerland
roman.machacek@unibe.ch

Anastasiia Grishina
Simula Research Laboratory
Oslo, Norway
anastasiia@simula.no

Max Hort
Simula Research Laboratory
Oslo, Norway
maxh@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
leon.moonen@computer.org

***Abstract*—Automated Program Repair (APR) uses various tools and techniques to help developers achieve functional and error-free code faster. In recent years, Large Language Models (LLMs) have gained popularity as components in APR tool chains because of their performance and flexibility. However, training such models requires a significant amount of resources. Fine-tuning techniques have been developed to adapt pre-trained LLMs to specific tasks, such as APR, and enhance their performance at far lower computational costs than training from scratch.**

In this study, we empirically investigate the impact of various fine-tuning techniques on the performance of LLMs used for APR. Our experiments provide insights into the performance of a selection of state-of-the-art LLMs pre-trained on code. The evaluation is done on three popular APR benchmarks (i.e., QuixBugs, Defects4J and HumanEval-Java) and considers six different LLMs with varying parameter sizes (resp. CodeGen, CodeT5, StarCoder, DeepSeekCoder, Bloom, and CodeLlama-2). We consider three training regimens: no fine-tuning, full fine-tuning, and parameter-efficient fine-tuning (PEFT) using LoRA and IA3. We observe that full fine-tuning techniques decrease the benchmarking performance of various models due to different data distributions and overfitting. By using parameter-efficient fine-tuning methods, we restrict models in the amount of trainable parameters and achieve better results.

***Index Terms*—large language models, automated program repair, parameter-efficient fine-tuning, AI4Code, AI4SE, ML4SE.**

I. Introduction

Software development, maintenance, and evolution are expensive processes, both in terms of money and time [1]. Supporting the efficiency of software engineers responsible for these workflows can save significant resources and enable companies to speed up the production and delivery of products without loss of quality. One of the key challenges that software engineers face is the occurrence of software defects, or bugs, which are unintended errors in the code that cause deviations from expected behavior. These defects vary in complexity, from simple one-line syntax errors to intricate multi-line logic bugs that can span multiple files and components.

Automated Program Repair (APR) aims to support developers with the software maintenance and evolution process, by helping them to fix any bugs they encounter and achieve their goals faster. Many techniques with varying characteristics and performance exist, depending mainly on the type and complexity of the bug [2]. Some methods specialize in specific programming languages [3–5], while others use specific patterns found in common bugs [6, 7]. A relatively new direction in APR methods is based on using Large Language Models (LLMs) [8–10], which have made significant progress in text-based tasks, including, for instance, text summarization, translation, and chatbots. When trained on programming languages, such models can learn to translate from buggy to fixed code rather than between languages.

While LLMs that are pre-trained on code already have considerable capabilities, fine-tuning them on a specific task such as APR can further improve their performance. However, the fine-tuning of LLMs is not without its challenges, such as computational overhead due to the large datasets and models, and overfitting of the models to the datasets [11]. Nevertheless, LLMs for APR remain a very active research topic due to the quality of the solutions (or patches) they generate compared to those generated by existing APR methods. One promising strategy to combat the high costs of fine-tuning, is the use of parameter efficient fine-tuning (PEFT) techniques [12]. These techniques can freeze parts of the original LLMs and enable fine-tuning on a reduced subset of parameters. However, a potential disadvantage of limiting the number of trainable parameters is that the model loses power and performance decreases. The investigation described in this paper aims to better understand the trade-off between efficiency gains and power loss in the context of fine-tuning models for APR.

Contributions: We conduct an extensive empirical study on the impact of fine-tuning LLMs on their APR performance. We provide a detailed comparison of different fine-tuning strategies for APR and offer practical insights into leveraging LLMs for automated software maintenance and evolution. Specifically, we make the following contributions:

⋆ *Establish a baseline of pre-trained LLMs for the program repair task.* We select six state-of-the-art LLMs pre-trained on code (resp. CodeGen [13], CodeT5 [14], Bloom [15], CodeLlama-2 [16], StarCoder [17], and DeepSeek-Coder [18]), and evaluate their out-of-the-box performance on a selection of APR benchmarks without applying any fine-tuning. Following earlier work by Jiang et al. [8], we use three widely adopted APR benchmarks (i.e., QuixBugs, Defects4J, and HumanEval-Java) and count the number of fixed examples in each benchmark.

[*] Work carried out as a master student at Simula Research Laboratory.

2025 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3315-9587-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSME64153.2025.00042

2576-3148/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSME64153.2025.00042

Finally, to help find the preferred input format for the selected LLMs, we analyze how including or omitting the buggy line(s) affects the LLM's performance on the benchmarks. Our findings provide insights into the suitability of non-fine-tuned models for APR and establish a baseline for the next steps of our study.

- ⋆ *Assess the impact of full-model fine-tuning on APR performance.* We empirically examine the effects of full-model fine-tuning of the selected LLMs on an APR-specific dataset of bug-fix pairs on their APR capabilities. This helps us to understand whether allowing adjustment of all parameters during fine-tuning improves or degrades the repair performance. The fine-tuning dataset was collected by Zhu et al. [19] and used earlier in the study by Jiang et al. [8]. We measure model effectiveness using established APR metrics such as exact match and CodeBLEU [20], along with model loss on the training and validation sets, and make a final evaluation of the model performance on the three APR benchmarks. Since the datasets for fine-tuning and benchmarking do not have the same bug-fix distribution, this simulates scenarios closer to the real world.
- ⋆ *Explore the effects of parameter-efficient fine-tuning (PEFT) on APR performance.* We examine adapter-based PEFT techniques, focusing on LoRA [21] and IA3 [22], which optimize training efficiency by fine-tuning fewer parameters. We compare these approaches to full fine-tuning, analyzing trade-offs between computational efficiency and performance. Our results highlight the potential of overcoming resource constraints using PEFT.
- ⋆ *Analyze the impact of LoRA hyperparameters on APR performance.* We systematically vary LoRA hyperparameters to assess their impact on APR performance. This part of the study provides a deeper understanding of how parameter-efficient fine-tuning configurations affect LLM effectiveness for APR. The insights from this experiment help to optimize PEFT strategies in future APR research.
- ⋆ *Provide a comprehensive replication package for experimental setup and evaluation.*[1] We provide a replication package with code and results for our study to enable others to build on our work. Our code builds on the code provided by Jiang et al. [8], adapting it to accommodate the benchmarking of additional models and parameter-efficient fine-tuning. Our extensions were developed in a modular fashion to facilitate the easy integration of additional models, techniques, and benchmarks [23].

II. Background and Related Work

A. Large Language Models

The use of LLMs has gained a lot of popularity in recent years, due to the increase of computational resources and breakthroughs in deep learning, such as transformer models. The transformer architecture [24] allows models to efficiently learn long-range dependencies in text data. Contrary to previous approaches towards sequence processing, transformers do not use recurrent sequential structures to process sequences, instead, they process sequences in parallel through multiple layers to produce output sequences, thus improving upon problems faced by RNN-based architectures. A core component of transformers is self-attention, allowing them to focus on different parts of the sequence to capture token dependencies.

Training of LLMs is typically done using semi-supervised learning approaches. For example, during pre-training, we may mask some of the tokens in a sentence and make the model predict the masked or missing token in the sequence. Another approach is to predict the next token in sequence, which is achieved by masking tokens after the predicted token. Currently, the fill-in-the-middle (FIM) approach is used for decoder models, where the whole sentence is used to predict the next token in sequence by re-structuring the input sentence using special tokens [25].

By pre-training LLMs we obtain a model with knowledge about various domains, problems, and grammatical structure. Further fine-tuning this model then improves the model's performance and understanding of the specific task at hand, which has been done for a diverse set of software engineering tasks [26–29].

Nowadays, research and university clusters cannot compete with industry giants in training the biggest LLMs due to scarce resources, thus restraining researchers with limited computational assets from the state-of-the-art models. One approach for reducing training time and memory requirements is to use only part of a model's parameters or layers, such as training only last layers of the model, while keeping the rest of the parameters *frozen* to decrease the amount of trainable parameters. Another common alternative to freezing is *pruning* [30], in which weights close to zero are clipped to zero, thus making them sparse.

B. Parameter Efficient Fine-tuning

Parameter efficient fine-tuning (PEFT) [12] is another technique to address the challenge of having an LLM with a large amount of trainable parameters. PEFT techniques address this challenge by freezing the whole model and adding an additional trainable structure, called an adapter.

A potential disadvantage of limiting the number of trainable parameters is that the model loses its power and performance decreases. The investigation described in this paper aims to better understand the trade-off between efficiency gains and power loss in the context of fine-tuning models for APR.

In particular, we investigate the impact of the LoRA [21] and IA3 [22] adapters on APR because of their promising results in other applications [31].

1) LoRA: Low-Rank Adaptation (LoRA) is a method developed by Hu et al. [21]. The main idea behind LoRA is to decompose large-weight matrices from Large Language Models into smaller low-ranking factors, thus reducing memory and increasing the computational efficiency of the model during training and inference. The inspiration comes from the

[1] Replication package: https://doi.org/10.5281/zenodo.16359186.

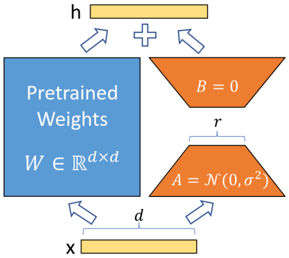

Fig. 1. Illustration of LoRA decomposition [21].

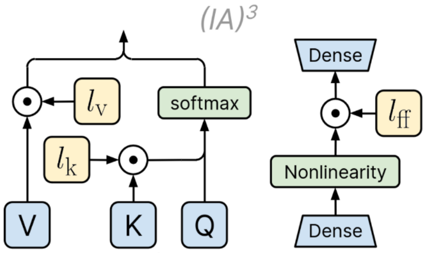

Fig. 2. Illustration of IA3 adapter [22].

observation that pre-trained models with a large number of parameters inherently exhibit a low-dimensional structure [32]. Furthermore, pre-trained LLMs learn efficiently even after being projected into a subspace of lower dimension [21].

The goal is to project the weight matrix $W_0 \in \mathbb{R}^{d\times k}$ to low-rank decomposition $W_0 + \Delta W = W_0 + BA$ as can be seen in Figure 1, where $B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k}$ and the rank $r < \min(d, k)$, resulting in $d \cdot r$ trainable parameters.

The pre-trained weights are frozen during training, ΔW is approximated by the parameterized matrices A, B. Forward pass, or inference, then corresponds to the use of pre-trained weight W_0 together with an approximation of ΔW during fine-tuning:

$$f(x) = W_0 x + \Delta W x = W_0 x + BAx$$

LoRA has been adopted in many frameworks and serves as an efficient method of training LLMs. The advantage of LoRA lies in the ease of use for applications of Large Language Models to new domains with limited resources, where instead of fine-tuning the whole model for small problems, we approximate the update on a subspace, saving time and resources.

2) IA3: Infused Adapter by Inhibiting and Amplifying Inner Activations (IA3) [22] is a parameter-efficient fine-tuning technique that promises improvement over LoRA. The idea of IA3 lies in optimizing the transformer architecture directly, more specifically the attention mechanism, by adding three vectors in order to re-scale key and value matrices. Similarly to LoRA, pre-trained weights remain frozen but instead injected scaling vectors are trained, as illustrated in Figure 2. Scaling vectors (as opposed to low-rank matrices) reduce the parameters to save time and resources during training and inference. Consider the vectors $l_k \in \mathbb{R}^{d_k}, l_v \in \mathbb{R}^{d_v}$, and $l_{ff} \in \mathbb{R}^{d_{ff}}$. Attention A mechanism of the transformer then becomes [22]:

$$A = \sigma\left(\frac{Q(l_k \cdot K^T)}{\sqrt{d_k}}\right)(l_v \cdot V)$$

Here, σ corresponds to the softmax function. The IA3 method adds $L(d_k + d_v + d_{ff})$ parameters for the L-layer encoder part of the transformer and $L \cdot (2d_k + 2d_v + d_{ff})$ for a L-layer decoder. The forward pass in the position feed-forward network of the transformer can be written as [22]:

$$f(x) = l_{ff} \cdot f(W_1 x))W_2$$

Similarly as in LoRA, we can have multiple IA3 adapters on top of a single pre-trained model, each one trained and serving a different task. Because the base pre-trained model can be merged into the IA3 adapter, there is no overhead during inference. According to experiments by Liu et al. [22], IA3 outperformed full fine-tuning and performs better than LoRA while using fewer trainable parameters.

Before focusing on program repair in the next section, we briefly review studies that consider PEFT for other software engineering (SE) tasks, such as code generation, clone detection, defect detection, and code summarization. Zhuo et al. conducted one of the first studies of PEFT for code LLMs and found that while complete fine-tuning generally performs best on code generation and code comprehension benchmarks, LoRA offered the most competitive trade-offs between cost and performance, particularly for the largest (16B parameter) models in their study [33]. Weyssow et al. empirically investigated PEFT for code generation and compare it to in-context learning (ICL) and retrieval-augmented generation (RAG) to guide the synthesis with task-specific examples [34]. They found that PEFT performs superior to ICL and RAG across a diverse set of LLMs and three representative Python code generation datasets. Afrin et al. [35] explored the impact of PEFT, in particular Quantized LoRA (QLoRA) on code summarization. They found that QLoRA enables efficient fine-tuning of code LLMs for summarization and requires minimal parameter adjustment compared to full model fine-tuning. Finally, Haque et al. conducted a recent systematic literature review on PEFT for code LLMs [31].

C. Automated Program Repair

Automated Program Repair (APR) [9] is a field of software engineering that deals with the identification and repair of bugs. Bugs are flaws, mistakes, or errors, in the source code. APR can be useful for the maintenance of already existing repositories and the development of new projects, increasing code quality, and decreasing the time spent on debugging. To accomplish this goal, several techniques and algorithms were created in order to analyze the code, detect bugs in it, and generate the corresponding patches, thus fixing the bug.

In the following, we are going to discuss the learning-based APR methods and techniques, surveyed by Huang et al. [2] and Zhang et al. [9]. Their goal is to use machine learning algorithms to learn and repair bugs and leverage huge amounts of software engineering data. With the increasing amount of data, this approach gained significant popularity over the last few years and was used to repair errors [8, 36–38] and vulnerabilities [39–41] in code. However, learning-based methods suffer from a number of limitations, including internal interpretation of models and the repair of long sequences [2].

Many methods for automatic repair of software bugs were developed, using abstract syntax trees [42], code similarity and automatic matching [43], convolutional neural networks [44], code reviews [45], grammar rules [46] along with syntax [5], code aware beam-search [47] and hybrid techniques combining ML techniques with fault localization [48]. The focus has also been on having models capable of repairing programs in multiple languages [49] with multiple lines and locations [48], called multi-hunk [2], repair capabilities. Repair of easy programming errors, usually one-line semantic and syntactic errors, utilized RNNs [50], graph neural networks [51], data perturbation [37], error messages [52], transformer-based models [36], parameter-efficient methods for code generation [34, 53] along with prompt engineering [54].

With LLMs, we are able to use datasets and pre-train models to gain an understanding of code syntax and logic before fine-tuning models for a specific code repair task and language. By providing models with context, the code around the bug, we allow models to make context-aware repairs. Pre-training models on vast amounts of code alleviates a problem of limited patch variety, since the models do not need to be trained on relatively small datasets consisting of bug-fix pairs. Recently, LLMs were used for APR with [8] and without fine-tuning [38, 55] and achieved promising results. Moreover, there are some recent works which observed the benefit of using PEFT techniques for APR [53, 56, 57]. For instance, Silva et al. [53] presented an approach for APR which fine-tuned a CodeLlama-7b model with LoRA. The performance of LoRA was better than full fine-tuning.

Jiang et al. [8] carried out APR experiments with full fine-tuning on different LLMs and share a framework. We build upon their framework but use a different set of models and add PEFT methods on top of these. The closest work to ours is the recent study by Li et al. [58], who also investigated PEFT techniques for APR, and the work by Huang et al. [59], who carried out a study on the fine-tuning of LLMs for APR concurrently with our investigation. In contrast to our experiments, Li et al. considered instruction-tuning datasets while we perform regular fine-tuning. We compare our findings where possible, i.e., where the same models and hyperparameters are used, and thus investigate how instruction-based fine-tuning [58] affects the results compared to NMT-style fine-tuning as is also done by Jiang et al. [8].

III. Experimental Design

In this section, we describe the experimental details (e.g., datasets, models, and evaluation metrics) to answer the following four research questions:

- **RQ1:** What is the benchmarking performance of Large Language Models pre-trained using code on the program repair task, without further fine-tuning?
- **RQ2:** How does full fine-tuning of LLMs on a dataset of bug-fix pairs affect the resulting APR performance?
- **RQ3:** What is the effect of adapter-based parameter-efficient fine-tuning of LLMs on APR benchmarks?
- **RQ4:** How do hyperparameters of LoRA affect the performance of selected LLMs?

A. Datasets

Here we outline the three benchmarking datasets, to evaluate LLMs on the APR task, and one dataset for fine-tuning LLMs. First, we introduce the three benchmarking datasets: Defects4J, QuixBugs, HumanEval-Java. Among these three benchmark datasets, Defects4J contains real-world projects with long context, while HumanEval-Java and QuixBugs include smaller programming problems with shorter context.

Defects4J [60] is a collection of real-world Java bugs collected from multiple open-source projects. The amount of bugs in the database varies depending on the Defects4J version, in our case we work with Defects4J 2.0.1 consisting of 835 active bugs and 29 deprecated bugs, of which 219 are used. The infrastructure provided contains information about the bug, along with tests and a fixed code.

QuixBugs [61] is a multilingual collection of 40 programs translated into Python and Java collected from the Quixey Challenge, where we use only Java programs. Test cases are provided with an infrastructure for the validation of each of the programs. The defects found in each program are further classified.

HumanEval-Java [8] is a manually created dataset based on the HumanEval dataset from which 163 bugs are used. The reason for doing so is to eliminate the threat that models have already seen the dataset during pre-training [8]. Each of the programs in the HumanEval dataset, together with their test cases, was converted from Python to Java. HumanEval-Java follows a similar structure as QuixBugs, containing both one-line and multi-line bugs together with the fixed program files and their test cases.

Finetuning: For fine-tuning our LLMs, we use the same dataset as Jiang et al. [8], introduced by Zhu et al. [19]. As the dataset has no formal name, we refer to it as **CLM**, based on the GitHub repository name for the work of Jiang et al. [8], where we obtained it.[2] The CLM dataset was created from Java projects on GitHub, consisting of 1,083,185 commits from March 2011 to March 2018. Filtering of commits was done to obtain only those patches that correspond to either an update or an insertion of a single-hunk statement, in line with the three test datasets. Moreover, to avoid benchmark contamination through data leakage, all patches related to projects in Defects4J were removed based on a comparison using abstract syntax trees [19]. The resulting dataset contains 143,666 instances, of which 129,300 (80%) are in the training

[2] https://github.com/lin-tan/clm

dataset and 14,366 (20%) in the test dataset. By using this cleaned CLM dataset for fine-tuning, we can leave the three benchmark datasets intact to enable a fair comparison with other papers. However, if there is a domain shift between the patches included in the fine-tuning and benchmark datasets, the resulting APR performance can be negatively affected.

B. Models

In this section, we present the LLMs used for our experiments. We chose them based on their performance on the HumanEval dataset [62], popularity and availability on Hugging Face, with the intention of improving over APR performance observed by Jiang et al. [8]. Moreover, we consider CodeGen and CodeT5 models to partially replicate the results in the related literature [8]. DeepSeekCoder and CodeLlama-2 allow us to additionally explore the newer models and compare with the results of Li et al. [58]. In total, we selected six different LLMs with different sizes:

CodeGen [13] is a family of decoder models based on transformer architecture. Today there are 3 available versions: CodeGen-1, CodeGen-2, and CodeGen-2.5 from which CodeGen-1, or CodeGen, will be used. Models were pre-trained on the next token prediction task on three datasets: ThePile, BigQuery, and BigPython, resulting in CodeGen-NL, CodeGen-Multi, and CodeGen-Mono versions. In our experiment, we use CodeGen-Multi with three sizes: 350M, 2B to 6B.

CodeT5 [14] is a family of encoder-decoder models based on transformer architecture. Nowadays, there is a newer version is available, CodeT5+, but for our experiment, CodeT5 is used to replicate the model selection and results by Jiang et al. [8]. The models were pre-trained using the Masked Identifier Prediction task on the CodeSearchNet dataset, resulting in CodeT5-small (60M), CodeT5-base (220M) and CodeT5-large (770M) versions.

StarCoder [17] is a family of decoder models based on transformer architecture. Models were pre-trained using the Fill-in-the-Middle (FIM) task on 80 programming languages to produce StarCoderBase and further fine-tuned on Python code to produce StarCoder. We are using StarCoderBase and its 1B and 3B versions.

DeepSeekCoder [18] is a family of transformer-based decoder models. Models were pre-trained using a Fill-in-the-Middle approach on the custom dataset consisting of 87% code, 10% code-related language and 3% noncode-related Chinese language [18] producing DeepSeekCoder-Base and further tuned through instruction-based data to produce DeepSeekCoder-Instruct. We use the 1.3B and 6.7B versions of DeepSeekCoder-Base. Note that at the time of writing, we used the latest DeepSeekCoder model available, i.e., v1.

Bloom [15] is a family of decoder models based on transformer architecture. Models were pre-trained using next token prediction on the ROOTS dataset consisting of 46 natural and 13 programming languages. We use versions 560M, 1B7 (1B700M) and 7B1 (7B100M).

CodeLlama2 [16] is a family of decoder models based on transformer architecture built on the Llama2 family of models [63]. Models were pre-trained using Fill-in-the-Middle task on custom datasets starting from versions of Llama2 resulting in CodeLlama2 models, which were further tuned on Python to produce CodeLlama2-Python along with instruction-based dataset resulting in CodeLlama2-Instruct. We use the 7B version of CodeLlama2.

C. Evaluation

To measure the ability of LLMs to fix one of the benchmarking problems, we let them generate 10 patches for each problem, as done in related works [8, 53, 56–58]. Each patch is then evaluated with the unit tests for the corresponding problem. The results of the tests determine the success of the patch, more specifically they provide detailed information about the execution of the patch. The results of the patch execution can be either:

- **Plausible (P):** Patch was compiled and the project successfully passed through all of its test(s).
- **Timeout (T):** Patch compilation and execution exceeded the time limit.
- **Uncompilable (U):** Patch was not compilable and the project, therefore, could not be further executed.
- **Wrong (W):** Patch was compiled successfully but the project did not pass its test(s).
- **Unknown (UNK):** Unique error due to the empty patch, due to the difference between the benchmarks and the Java version. This leads to the unavailability of execution for the generation of patches.

For the proceeding experiments, we focus on plausible patches (i.e., we count if any of the 10 patches is plausible), as an indication of their correctness.

In addition to the performance on tests, we investigate the quality of generated patches based on their closeness to the ground truth, human patches. This is in particular helpful for patches that do not solve all the tests. To compute the similarity between two code snippets, we use CodeBLEU. CodeBLEU [20] extends the BLEU metric [64] to be suitable for use in the quality assessment of code generation models. Similarly to BLEU, CodeBLEU is used for comparison between the generated code and the reference output. CodeBLEU takes into account not only the n-gram match as BLEU, but also considers the logic, syntax, and structure of the code. Therefore, CodeBLEU is often used in the evaluation of code generation models and is also used in many benchmarks for code, such as CodeXGLUE [65].

D. Implementation Details

We use Hugging Face[3] as the provider for the selected LLMs and implementations of the LoRA and IA3 PEFT techniques.[4]

To use datasets for inference and fine-tuning, the inputs need to be pre-processed for each model. The corresponding pre-processing for each model is shown in Listing 1, inspired by

[3] https://huggingface.co/

[4] https://huggingface.co/docs/peft/index

Jiang et al. [8] with the addition of FIM pre-processing. In our experiments, we consider two fine-tuning scenarios: with buggy line, without buggy line. The scenario "with buggy line" uses the prompts as show in Listing 1, while the scenario "without buggy line" removes the lines "// bug start", "// bug end", and any line in between. Bloom and CodeGEN use formatting similar to FIM, helping models to predict only replacement for the buggy line, instead of the code after the buggy line. CodeT5 models are fine-tuned in an encoder-decoder fashion, where the commented target is used only for the output. The remaining models are trained using the FIM approach [25].

IV. Results and Discussion

In the first research question, we start by testing a set of models on three APR benchmarks without fine-tuning the models. The benchmarks consist of Java code and tests and vary in size and complexity of the bug. We proceed with model selection and fine-tuning on a dedicated APR dataset to study the performance of fine-tuned models on the same benchmarks (RQ2). To evaluate performance during fine-tuning, we use exact match and CodeBLEU. Finally, we used original models to compare the effect of full fine-tuning and parameter-efficient fine-tuning on APR results (RQ3). Given the closeness to the work by Li et al. [58], we compare our results with theirs were possible. These comparisons are done for the identical models and datasets, as well as 10 patches for evaluation.

Listing 1 Illustration of input code pre-processing for fine-tuning of selected models with the last line being the target, correct fix for the buggy line.

```
// Bloom:
public static int
       bitcount(int n) {
    int count = 0;
    while (n != 0) {
        // bug start:
        n = (n ^ (n - 1));
        // bug end
        count++;
    }
    return count;
}
// fix:
n = (n & (n - 1));

// CodeLlama2:
public static int
       bitcount(int n) {
    int count = 0;
    while (n != 0) {
        // bug start:
        n = (n ^ (n - 1));
        // bug end
        <FILL_ME>
        count++;
    }
    return count;
}
n = (n & (n - 1));

// DeepSeekCoder:
<|fim_begin|>
public static int
       bitcount(int n) {
    int count = 0;
    while (n != 0) {
        // bug start:
        n = (n ^ (n - 1));
        // bug end
        <|fim_hole|>
        count++;
    }
    return count;
}
<|fim_end|>
n = (n & (n - 1));
```

```
// CodeGEN:
public static int
       bitcount(int n) {
    int count = 0;
    while (n != 0) {
        // bug start:
        n = (n ^ (n - 1));
        // bug end
        count++;
    }
    return count;
}
// fix:
n = (n & (n - 1));

// CodeT5:
public static int
       bitcount(int n) {
    int count = 0;
    while (n != 0) {
        // bug start:
        n = (n ^ (n - 1));
        // bug end
        count++;
    }
    return count;
}
// n = (n & (n - 1));

// StarCoder:
<fim_prefix>
public static int
       bitcount(int n) {
    int count = 0;
    while (n != 0) {
        // bug start:
        n = (n ^ (n - 1));
        // bug end
        <fim_suffix>
        count++;
    }
    return count;
}
<fim_middle>
n = (n & (n - 1));
```

TABLE I
RQ1 (NO FINE-TUNING): NUMBER OF PLAUSIBLE PATCHES FOR QUIXBUGS (QB), HUMANEVAL-JAVA (HE), DEFECTS4J (D4J) COMPARED WITH JIANG ET AL. [8] IN PARENTHESES, WITH THE BEST RESULTS FOR EACH MODEL ON EACH BENCHMARK HIGHLIGHTED IN BOLD.

	Without buggy line			With buggy line		
Model	QB	HE	D4J	QB	HE	D4J
Bloom-560m	1	**10**	**8**	**4**	5	6
Bloom-1b7	**1**	**15**	**10**	**1**	12	8
Bloom-7b1	**6**	**24**	10	4	22	**14**
CodeGen-350M	**6**(7)	**25**(30)	**12**(3)	4	16	7
CodeGen-2B	**13**(15)	**44**(49)	**20**(4)	12	34	**20**
CodeGen-6B	**16**(16)	**42**(46)	**18**(8)	**16**	**42**	**18**
CodeLlama2-7b	**33**	95	83	28	**96**	**93**
CodeT5-small	**0**(3)	**4**(3)	**10**(2)	**0**	1	4
CodeT5-base	**0**(0)	**4**(5)	**8**(4)	**0**	0	2
CodeT5-large	**2**(3)	2(6)	**4**(1)	1	**3**	**4**
DeepSeekCoder-1.3b	**33**	94	72	31	**97**	**86**
DeepSeekCoder-6.7b	**33**	**107**	89	30	105	**103**
StarCoder-1b	**22**	69	62	22	**70**	**74**
StarCoder-3b	**32**	**94**	63	31	85	**87**
StarCoder-7b	**33**	**91**	82	31	**91**	**96**

A. *RQ1: Performance without Fine-tuning*

In the first research question, we investigate the performance of the 15 LLMs on the three APR benchmarking datasets (QuixBugs, HumanEval-Java, Defects4J) without any fine-tuning. In accordance with Jiang et al. [8], we query the LLMs with and without highlighting the buggy lines.

Table I summarizes the number of plausible patches achieved by each model for the three datasets. We can observe that the number of model parameters effects model performance, where more parameters usually correspond to better performing models. One example for this is CodeGen, for which CodeGen-2B and CodeGen-6B always generate more plausible patches than CodeGen-350M. The same holds for StarCoder, where the smallest model, StarCoder-1b, generates the least plausible patches in all six cases. Once the models reach a size of a billion parameters, the differences are less pronounced, with 2B and 3B models able to tie or even generate more plausible patches than their 7B counterparts. For CodeT5 however, we observe the opposite. CodeT5-small has the best performance in 3 out of 6 cases.

Moreover, the results compared to Jiang et al. [8] differ slightly for QuixBugs and HumanEval-Java and substantially on Defects4J benchmarks. There are multiple reasons: for instance, Jiang et al. did not specify the Java version used, furthermore, they filter out more programs based on their length, resulting in fewer fixes.

Importantly, pre-training of models uses large datasets, usually mined from GitHub. This may cause data leakage, where the benchmarks may have been seen by models at some point during pre-training because they are available as public repositories on GitHub. This could explain the behavior of best-performing (newer) models. If we consider the total number of buggy programs with the number of fixed ones, without buggy line, by the best performing model DeepSeekCoder-6.7b we obtain 82.5% of fixed codes for QuixBugs, with 65.6% for HumanEval-Java and 40.6% for Defects4J.

Intuitively, by adding a buggy line, we could steer the model in the direction of a potential fix. Thus, the model does not have to come up with the solution fully, but can utilize the already existing, wrong solution. However, as we can see in Table I, adding buggy lines did not improve performance in 29 out of 45 cases. Adding buggy lines for QuixBugs is rarely useful, it only increases the number of plausible patches for Bloom-560 from 1 to 4. While adding a buggy line does not lead to performance improvements in the majority of cases, it does improve the best performing models (CodeLlama, DeepSeekCoder, StarCoder). In particular, the performance on Defects4J is improved for all six models sizes, and in three out of six sizes for HumanEval. One reason for such an inconsistent effect of including buggy lines in the repair task is the pre-training phase of the models, which determines if and how they are able to make use of such additional information.

Compared to the results of Li et al. [58] (Table 4, no fine-tuning), the CodeLlama2-7B model in our work solves more problems on all datasets. We appoint this difference to different prompting strategies and general stochasticity of LLM inference. The trend of larger models performing better holds in the study of Li et al. [58] as well. Comparing DeepSeekCoder results without fine-tuning from Table I is not possible because Li et al. [58] only reported results of the fine-tuned model with adapters.

To summarize the answer to RQ1: Benchmarking performance varies between each model, where DeepSeekCoder-6.7b generates the largest number of plausible patches for QuixBugs (33 out of 40 problems), HumanEval-Java (107 out of 163 problems) and Defects4J (89 out of 219 problems). Larger models perform better in 34 out of 48 cases, while the usefulness of adding buggy lines varies across models.

B. RQ2: Performance with Fine-tuning

RQ1 illustrated the performance of LLMs without fine-tuning, and we observed that some models, such as CodeT5, have rarely produced plausible patches. In this RQ, we investigate whether the performance of the pre-trained LLMs can be improved by fine-tuning them for the APR task. Therefore, we fully fine-tune selected models on the CLM dataset (Section III-A). Given the inconsistent results from RQ1, we decided to focus the following investigation on scenarios without including buggy lines for fine-tuning. In the work by Jiang et al. [8], models were fine-tuned for only one epoch. The reason being computation time and the non-significant decrease in loss after the first epoch. Here, we fine-tune models for 3 epochs to see if there is additional improvement of the models after three epochs, and more importantly provide numerical evidence.

The dataset was pre-processed to match the specific input format for each of the selected models (Section III-D). Furthermore, the CLM dataset is split into training and validation datasets. We log the fine-tuning process of models after every epoch to see their performance change on the CLM validation dataset with loss, CodeBLEU and exact match.

We selected models for fine-tuning based on their resource usage. The deciding factor is the computational resource, which we chose to be 1 node with A100, and V100 GPUs, however, note that larger models can be trained on several GPUs. This led to fine-tuning of models with less than 3b parameters, resulting in the selection of Bloom-560m/1b7, CodeGen-350M/2B, CodeT5-small/base/large, DeepSeekCoder-1.3b, and StarCoder-1b/3b.

Table II lists the model behavior during the three training epochs. As can be seen in Table, the models improve all three metrics (CodeBLEU, loss and exact match) on the validation dataset. CodeT5 models improved the most, since they had the worst loss and CodeBLEU values at the start. The best performing model in Table II is DeepSeekCoder-1.3b, with a CodeBLEU of 0.65 and an exact match of 0.26 on the validation dataset after three epochs of fine-tuning. Notice the difference between CodeBLEU and exact match, where the model understands the code structure but makes small mistakes, leading to loss of exact match. Smaller CodeT5 models have a similar loss on training and validation datasets, from which we hypothesize that they do not overfit, while the remaining models overfit to a higher extent.

All models seem unable to capture the complexity of program repair tasks as they are trained further. It is visible by considering the progress of losses as we train models further, where in some cases validation loss stagnates at high values or even increases at epoch 3. There may be multiple reasons, one of them being the high variability of the CLM dataset. Another potential reason is the use of relatively small models for complex tasks. However, compared to Jiang et al. [8], we observe improvements in exact match and CodeBLEU even though the loss stagnates. Therefore, there is potential for training models beyond one epoch.

Table III shows the results of various models and their performance in terms of plausible repairs of programs without a buggy line, compared to the results from Table I. From this table, we observe that the models, compared to the base (non fine-tuned), perform differently for each model. CodeT5, Bloom, and CodeGen show improvements, whereas DeepSeekCoder and StarCoder show deterioration after fine-tuning compared to using the base models.

TABLE II
RQ2 (FULL-MODEL FINE-TUNING): MODEL PERFORMANCE ON VALIDATION DATASET (FOR EPOCHS 0, 1, 2, AND 3). RESULTS ON THE TRAINING DATASET ARE DISPLAYED FOR EPOCH 3. THE BEST RESULTS ON THE VALIDATION DATASET ARE HIGHLIGHTED IN BOLD. THE ARROWS AFTER METRICS INDICATE WHETHER LARGER (↑) OR SMALLER (↓) VALUES ARE CONSIDERED BETTER.

	CodeBLEU (↑)					Loss (↓)					Exact Match (↑)				
	Validation Epoch					Validation Epoch					Validation Epoch				
	0	1	2	3	Train	0	1	2	3	Train	0	1	2	3	Train
Bloom-560m	0.45	0.49	0.54	0.57	0.73	1.21	0.88	0.74	0.76	0.25	0.00	0.06	0.13	0.19	0.53
Bloom-1b7	0.46	0.53	0.58	0.60	0.84	1.18	0.68	0.59	0.66	0.14	0.00	0.08	0.16	0.21	0.69
CodeGen-350M	0.41	0.57	0.61	0.62	0.86	1.06	0.49	0.46	0.55	0.13	0.00	0.11	0.18	0.22	0.70
CodeGen-2B	0.38	0.53	0.59	0.61	0.88	0.90	0.58	0.52	0.64	0.13	0.00	0.10	0.19	0.23	0.79
CodeT5-small	0.05	0.38	0.39	0.38	0.39	4.37	0.69	0.66	0.65	0.59	0.00	0.03	0.05	0.05	0.06
CodeT5-base	0.07	0.36	0.38	0.39	0.45	4.78	0.57	0.53	0.52	0.33	0.00	0.08	0.12	0.14	0.25
CodeT5-large	0.01	0.36	0.42	0.43	0.52	4.38	0.56	0.50	0.50	0.23	0.00	0.12	0.19	0.23	0.41
DeepSeekCoder-1.3b	0.45	0.60	0.64	**0.65**	0.89	0.75	**0.44**	**0.44**	0.56	0.10	0.00	0.15	0.23	**0.26**	0.82
StarCoder-1b	0.44	0.58	0.61	0.63	0.87	1.05	0.57	0.53	0.59	0.11	0.00	0.12	0.20	0.24	0.72
StarCoder-3b	0.46	0.52	0.60	0.62	0.86	0.89	0.64	0.58	0.66	0.13	0.00	0.11	.019	0.24	0.75

We argue that the reason for this deterioration is due to different data distributions. We used the CLM dataset, which may not be representative of various bugs encountered in the benchmarking datasets, including Defects4J, HumanEval-Java, and Quixbugs. One solution to improve the quality of models is to use larger fine-tuning datasets.

To summarize the answer to RQ2: Full fine-tuning leads to improvement of models that performed poorly without fine-tuning, such as CodeT5 and Bloom, but leads to worsening of the best performing models, including DeepSeekCoder.

TABLE III
RQ2 (FULL-MODEL FINE-TUNING): NUMBER OF PLAUSIBLE PATCHES FOR QUIXBUGS (QB), HUMANEVAL-JAVA (HE), AND DEFECTS4J (D4J). THE BEST RESULTS FOR EACH MODEL ON EACH BENCHMARK ARE HIGHLIGHTED IN BOLD.

	Base			Epoch 1			Epoch 3		
Model	QB	HE	D4J	QB	HE	D4J	QB	HE	D4J
Bloom-560m	1	10	8	5	17	**41**	**6**	**19**	**41**
Bloom-1b7	1	15	10	8	10	43	**11**	**23**	**53**
CodeGen-350M	6	25	12	13	**43**	**67**	**16**	42	61
CodeGen-2B	**13**	**44**	20	8	25	**66**	11	36	64
CodeT5-small	0	4	10	**14**	**44**	53	13	39	**60**
CodeT5-base	0	4	8	14	**45**	68	**17**	39	**75**
CodeT5-large	2	2	4	9	36	62	**16**	**47**	**72**
DeepSeekCoder-1.3b	**33**	**94**	72	19	54	**84**	15	64	80
StarCoder-1b	**22**	**69**	62	12	41	**72**	13	49	71
StarCoder-3b	**32**	**94**	**63**	6	26	51	11	37	**63**

C. RQ3: Performance with Parameter-efficient Fine-tuning

In addition to standard full-model fine-tuning performed in RQ2, RQ3 investigates PEFT methods. In particular, we use LoRA and IA3. Furthermore, we investigate the impact of hyperparameters for fine-tuning with LoRA (RQ4). Due to computational resources and time constraints, CodeGen, CodeT5, and DeepSeekCoder models were selected based on their performance.

Similar to RQ2, we fine-tune the models for 3 epochs. We follow the default parameters recommended by Hugging Face PEFT,[5] where the default recommended rank and scaling factor of LoRA were $r = 8, \alpha = 16$ at the time of writing. We follow Hugging Face default settings for IA3, too.

Table IV shows the validation and training performance of the fine-tuned models using LoRA and IA3. Both methods improve the losses mainly up to the first epoch, where CodeBLEU and validation set loss stagnate for almost all the models. A possible reason for this is that adapters significantly reduce the number of trainable parameters and the PEFT-trained models reach their plateau performance at this point. Interestingly, Li et al. [58] noticed a drop in metrics with the increasing number of epochs (Table 6 [58]), which goes in line with the finding that PEFT training does not necessarily benefit from more epochs.

We notice a large drop in the exact match compared to the full-model fine-tuning (Table II). However, the drop in CodeBLEU is not as large: we observe rather high values with DeepSeekCoder-6.7b having a CodeBLEU of 0.63 (LoRA) and 0.62 (IA3). This shows that models can capture code structure similarly to full-finetuning, with the main difference being small mistakes like variable names, leading to low exact match. When using PEFT, models perform similarly for training and validation datasets, thus reducing overfitting. Finally, we point out the stagnation in metrics and loss in most models after the first epoch. For this reason, we chose

[5] https://huggingface.co/docs/peft/index

TABLE IV
RQ3 (PEFT): MODEL PERFORMANCE ON VALIDATION DATASET (FOR EPOCHS 0, 1, 2, AND 3). RESULTS ON THE TRAINING DATASET ARE DISPLAYED FOR EPOCH 3. THE BEST RESULTS ON THE VALIDATION DATASET ARE HIGHLIGHTED IN BOLD. THE ARROWS AFTER METRICS INDICATE WHETHER LARGER (↑) OR SMALLER (↓) VALUES ARE CONSIDERED BETTER.

	CodeBLEU (↑)					Loss (↓)					Exact Match (↑)				
	Validation Epoch					Validation Epoch					Validation Epoch				
	0	1	2	3	Train	0	1	2	3	Train	0	1	2	3	Train
LoRA															
CodeGen-350M	0.39	0.58	0.58	0.58	0.60	1.12	0.47	0.47	0.46	0.44	0.00	0.06	0.07	0.08	0.09
CodeGen-2B	0.35	0.61	0.62	0.62	0.64	0.94	0.40	0.39	0.39	0.34	0.00	0.10	0.13	0.13	0.17
CodeGen-6B	0.42	0.61	0.62	0.62	0.65	0.93	0.38	0.38	**0.37**	0.33	0.00	0.11	**0.14**	**0.14**	0.20
CodeT5-small	0.04	0.33	0.35	0.34	0.34	4.39	0.84	0.82	0.81	0.79	0.00	0.02	0.03	0.03	0.03
CodeT5-base	0.08	0.34	0.35	0.34	0.35	4.80	0.69	0.67	0.67	0.65	0.00	0.04	0.05	0.05	0.05
CodeT5-large	0.01	0.33	0.35	0.35	0.36	4.37	0.57	0.56	0.55	0.53	0.00	0.06	0.07	0.07	0.08
DeepSeekCoder-1.3b	0.44	0.61	0.61	0.61	0.63	0.76	0.43	0.42	0.42	0.39	0.00	0.09	0.11	0.12	0.14
DeepSeekC.-6.7b	0.48	**0.63**	**0.63**	**0.63**	0.65	0.69	0.39	0.38	0.38	0.35	0.00	0.13	**0.14**	**0.14**	0.16
IA3															
CodeGen-350M	0.41	0.56	0.56	0.56	0.57	1.08	0.53	0.52	0.52	0.51	0.00	0.02	0.02	0.02	0.03
CodeGen-2B	0.38	0.59	0.59	0.59	0.60	0.93	0.46	0.45	0.45	0.44	0.00	0.04	0.04	0.04	0.05
CodeGen-6B	0.43	0.59	0.59	0.59	0.60	1.08	0.48	0.47	0.46	0.44	0.00	0.04	0.05	0.05	0.05
CodeT5-small	0.04	0.22	0.24	0.25	0.25	4.39	1.20	1.11	1.09	1.08	0.00	0.01	0.01	0.01	0.01
CodeT5-base	0.08	0.26	0.27	0.28	0.28	4.76	0.94	0.88	0.87	0.86	0.00	0.01	0.01	0.02	0.02
CodeT5-large	0.01	0.29	0.30	0.30	0.31	4.39	0.79	0.67	0.66	0.65	0.00	0.03	0.04	0.04	0.04
DeepSeekCoder-1.3b	0.46	0.60	0.60	0.60	0.61	0.83	0.49	0.48	0.47	0.47	0.00	0.06	0.06	0.06	0.06
DeepSeekC.-6.7b	0.50	**0.62**	**0.62**	**0.62**	0.62	0.75	0.43	**0.42**	**0.42**	0.42	0.00	0.08	0.08	**0.09**	0.09

the first epoch of each model fine-tuned using LoRA and IA3 and summarize their performance in Table V.

In Table V, we can observe performance improvements with PEFT techniques over base models, in particular with LoRA on CodeGen and DeepSeekCoder. CodeGen with LoRA improves over full fine-tuning, especially on the Defects4J benchmark with 34 additional plausible patches on CodeGen-2b. In general, by using adapters we were able to achieve improvements on all benchmarks. Contrary to expectation and trends observed by Li et al. [58], LoRA performs better than IA3 in 21 out of 24 cases. Li et al. reported that IA3 outperformed LoRA for DeepSeekCoder-6.7b, two CodeLlama model sizes but not Llama2-7b (see Table 5 in their study [58]). However, in the majority of our experiments, LoRA showed better performance than IA3, except for DeepSeekCoder-6.7b on D4J. Furthermore, CodeT5 models improved even more compared to full-model fine-tuning on HumanEval-Java and Defects4J. We conclude that adapters are a viable approach to fine-tuning LLMs that offer, in most cases, additional improvements in terms of the number of plausible patches generated in addition to using less resources.

Lastly, we present the number of trainable parameters utilized during fine-tuning with LoRA and IA3 in Table VI. When LoRA and IA3 are used, models have $< 1\%$ of the original number of trainable parameters. Importantly, we were able to fine-tune and fit larger original models (ca. 6B parameters) into GPU memory with the reduced number of parameters with respect to the PEFT method.

TABLE V
NUMBER OF PLAUSIBLE PATCHES FOR QUIXBUGS (QB), HUMANEVAL-JAVA (HE), AND DEFECTS4J (D4J) BASED ON FULL-MODEL FINE-TUNING (FMFT) FOR THREE EPOCHS, PEFT USING LORA, IA3 TRAINED FOR ONE EPOCH AND COMPARED WITH THE BASE MODELS WITHOUT FINE-TUNING. THE BEST RESULTS ARE HIGHLIGHTED IN BOLD FOR EACH BENCHMARK, MODEL, AND METHOD. "X" INDICATES MODEL CONFIGURATIONS WE WERE UNABLE TO TRAIN DUE TO HARDWARE LIMITATIONS.

	Base			FMFT			LoRA			IA3		
Model	QB	HE	D4J	QB	HE	D4J	QB	HE	D4J	QB	HE	D4J
CodeGen-350M	6	25	12	**16**	42	61	**16**	**64**	**82**	13	52	68
CodeGen-2B	13	44	20	11	36	64	**19**	**81**	**98**	15	67	74
CodeGen-6B	16	42	18	X	X	X	**24**	**81**	**104**	14	70	83
CodeT5-small	0	4	10	**13**	**39**	**60**	10	36	50	5	18	42
CodeT5-base	0	4	8	**17**	39	**75**	12	**50**	67	8	32	52
CodeT5-large	2	2	4	16	47	**72**	**17**	**60**	**72**	10	47	61
DeepSeekCoder-1.3b	**33**	94	72	15	64	80	27	**106**	100	28	94	**101**
DeepSeekCoder-6.7b	**33**	107	89	X	X	X	31	**108**	108	30	100	**114**

To summarize the answer to RQ3: Fine-tuning in a parameter-efficient way with adapters leads to an improvement of several models compared to full-model fine-tuning, e.g., for CodeGen. For instance, by using LoRA for CodeGen-2B, we used only 0.09% of trainable parameters of the full model, while achieving performance gains of 172%, 225%, 153% on the QuixBugs, HumanEval-Java and Defects4J benchmarks. Furthermore, we observed that most of the time, LoRA achieves better results than IA3, specifically, in 21 out of 24 cases.

TABLE VI
AMOUNT OF TRAINABLE PARAMETERS FOR EACH OF THE USED MODELS WITH LORA AND IA3 ADAPTER.

Model	Total	LoRA Trainable (%)	IA3 Trainable (%)
CodeT5-small	60,787,200	294,912 (0.49)	43,008 (0.07)
CodeT5-base	223,766,784	884,736 (0.39)	129,024 (0.06)
CodeT5-large	739,998,720	2,359,296 (0.32)	344,064 (0.05)
CodeGen-350M	357,367,808	655,360 (0.18)	61,440 (0.02)
CodeGen-2B	2,779,683,840	2,621,440 (0.09)	245,760 (9e-3)
CodeGen-6B	7,068,538,880	4,325,376 (0.06)	405,504 (6e-3)
DeepSeekCoder-1.3b	1,348,044,800	1,572,864 (0.12)	230,400 (0.02)
DeepSeekCoder-6.7b	6,744,707,072	4,194,304 (0.06)	614,400 (9e-3)

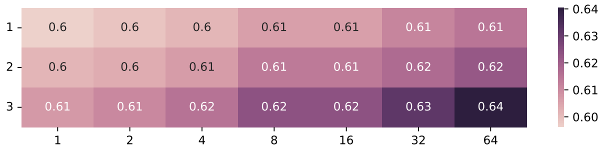

Fig. 3. CodeBLEU change with scaling factor for three epochs.

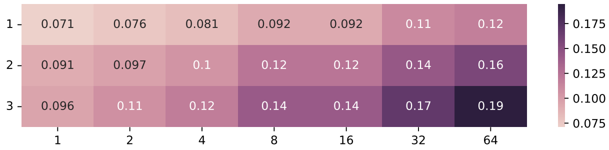

Fig. 4. Exact match change with scaling factor for three epochs.

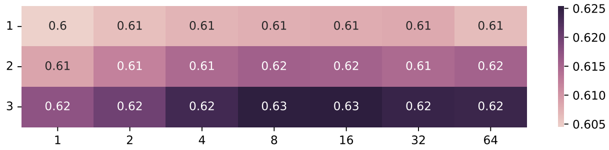

Fig. 5. CodeBLEU change with rank for three epochs.

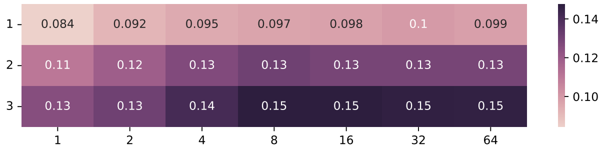

Fig. 6. Exact match change with rank for three epochs.

D. RQ4: Effects of LoRA Parameters

LoRA keeps the parameters of pre-trained models frozen and adds a layer with a few trainable parameters. The main hyperparameters in LoRA that impact the resulting model size and also performance are rank and scaling factor (Section II-B1). Rank affects the amount of trainable parameters, whereas the scaling factor controls the magnitude of parameter updates. To test the effect of LoRA parameters on fine-tuning results, we performed experiments with several values of rank and scaling factor. In particular, we investigate the following 8 values for both rank and scaling factor: 1, 2, 4, 8, 16, 32, 64. The default values for rank and scaling factor of LoRA are $r = 8, \alpha = 16$. We fix a default value for one and change the values of another hyperparameter in our experiments.

Figures 3 and 4 show the effect of the scaling factor while Figures 5 and 6 show the effect of the rank for LoRA when applied to CodeGen-2B. CodeBLEU is almost not affected by the change in rank or scaling factor and changes in the range between 0.6 and 0.64. However, the exact match shows larger deviations, interestingly more with the scaling factor than with rank. The conclusion is, however, the same: either rank or scaling factor does not affect the performance of models significantly.

We observe that the exact match metric is affected to a slightly larger degree by the scaling factor than by rank size and stays rather low in all cases. However, because exact match evaluates the output against only one version of bug fix, this metric is not the best to optimize LLMs for. These results obtained agree with [21], where the change in LoRA hyperparameters does not play a significant role.

Li et al. [58] varied the rank hyperparameter for one model, CodeLlama-7b, and did not change the scaling factor. The study shows that the best rank in terms of fixing bugs on the APR datasets is 16 while the scaling factor was fixed to 16. However, the study changes the rank while performing PEFT on a bug-fixing APR instructions dataset and testing the tuned models on the APR dataset. In our study, we notice the increase of the loss metrics with the increase of rank because we test LoRA on the validation part of the dataset used for PEFT, and observe a predictable increase in metrics values with the increase of the rank (and increase in trainable parameters). Intuitively, the more parameters are updated the better the metrics are on the data with the same distribution.

To summarize the answer to RQ4: Both LoRA hyperparameters, scaling factor and rank, lead to slight, negligible differences in the resulting metrics during training. Thus, following the recommended hyperparameter values as compared to their search does not drastically affect the performance of models.

E. Threats to Validity

There are several factors that can influence the results of our experiments and their validity. First, benchmarks like HumanEval-Java and QuixBugs were created from simple projects and consist of bugs that are not representative of complex real-world bugs. Another issue arises in the validation of the Defects4J benchmark, due to the specific packages and Java versions used (i.e., for running the running and evaluating the patches), causing differences in the number of problems and results compared to those of Jiang et al. [8].

Additionally, versioning of various libraries caused by internal dependencies leads to internal problems of benchmarks, where some tests are not possible to execute. Furthermore, the training APR dataset used that contains bug-fix pairs is not itself representative of bugs found in the benchmarks and has inherently different data distributions than the benchmarks themselves. To represent real-world complex bugs through benchmarks and training datasets, we would have to use much larger and more complex datasets.

The biggest reason for concern goes to data leakage, which can occur when models are pre-trained on a dataset on which we then test, skewing model performance [66]. LLMs nowadays use very large datasets that sometimes are not publicly available or too large to even check whether they include tested benchmarks. None of the models achieved close to the 100% benchmark performance, making it clear that the effect of data leakage is not the only factor that influences the performance of the model.

Lastly, we used plausibility to evaluate patches. Plausibility shows whether a patch passes all available tests but is not a guarantee of its correctness. One way to assess the correctness of patches is by manually checking. However, manual correctness verification is prone to subjectivity [67], which is why we remained with plausibility for patch evaluation.

V. Conclusions and Future Work

In this work, we conduct an extensive empirical study on the impact of fine-tuning LLMs on their APR performance. We provide a detailed comparison of different fine-tuning strategies for APR and offer practical insights into leveraging LLMs for automated software maintenance and evolution.

We studied a total of six state-of-the-art LLMs that were pre-trained on code, and evaluated their bug fixing performance in three settings: without fine-tuning, with full-model fine-tuning, with parameter-efficient fine-tuning. Our study has been inspired by the work of Jiang et al. [8] and draws parallels to the work of Li et al. [58]. We compared the models based on three popular APR datasets (QuixBugs, HumanEval-Java, and Defects4J). Furthermore, buggy lines were included or omitted from the input to analyze whether additional information improves the LLM's performance before fine-tuning.

Our findings show that some of the models decrease performance after full-model fine-tuning in comparison to their non-tuned counterparts. Reasons for this can be overfitting, dataset variability, and size. Full fine-tuning led to the improvement of smaller models and the deterioration of APR performance on selected benchmarks for larger models compared to previously made inferences. To alleviate issues of full fine-tuning, such as overfitting and computational efficiency, we investigate two parameter-efficient fine-tuning techniques (LoRA and IA3). Among these two, we observed that LoRA achieved better results. By using adapters as additional layers on top of frozen pre-trained models, we were able to solve issues of full-model fine-tuning and improve the performance of models compared to zero-shot inference for larger models.

Future work: Further research directions include the exploration of different adapters along with their interpretation. In addition, instruction-based models could provide additional help in steering the models toward a correct solution via adapting prompts and seem to be an interesting direction.

The exploration of adapters for larger models would provide additional performance and computational insights. One of the questions worth answering is whether we can automate the selection of adapters, based on dataset and model, to achieve the best performance. Finally, the use of different optimizers would allow us to decrease the memory usage of models and train larger models, providing additional direction for the optimization and training of large models.

VI. Data Availability

To support open science and allow for replication and verification of our work, a replication package with code and results is made available via Zenodo [23].

Acknowledgments

This work is supported by the Research Council of Norway through the secureIT project (IKTPLUSS #288787), and by the European Union through the Horizon Europe Marie Skłodowska-Curie Actions (#101151798). The empirical evaluation made use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), financially supported by the Research Council of Norway under contract #270053.

References

[1] A. Arcuri. "On the Automation of Fixing Software Bugs." In: *Companion of the 30th International Conference on Software Engineering*. Association for Computing Machinery, 2008, pp. 1003–1006. DOI: 10.1145/1370175.1370223.

[2] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang. *A Survey on Automated Program Repair Techniques*. 2023. DOI: 10.48550/arXiv.2303.18184. arXiv: 2303.18184 [cs].

[3] J. Bader, A. Scott, M. Pradel, and S. Chandra. "Getafix: Learning to Fix Bugs Automatically." In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360585.

[4] D. Drain, C. B. Clement, G. Serrato, and N. Sundaresan. *DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons*. 2021. DOI: 10.48550/arXiv.2105.09352. arXiv: 2105.09352 [cs].

[5] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen. "PyDex: Repairing Bugs in Introductory Python Assignments Using LLMs." In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (2024), 133:1100–133:1124. DOI: 10.1145/3649850.

[6] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. L. Traon. "Mining Fix Patterns for FindBugs Violations." In: *IEEE Transactions on Software Engineering* 47.1 (2021), pp. 165–188. DOI: 10.1109/TSE.2018.2884955.

[7] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande. "AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations." In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 2019, pp. 1–12. DOI: 10.1109/SANER.2019.8667970.

[8] N. Jiang, K. Liu, T. Lutellier, and L. Tan. "Impact of Code Language Models on Automated Program Repair." In: *Proceedings of the 45th International Conference on Software Engineering*. IEEE Press, 2023, pp. 1430–1442. DOI: 10.1109/ICSE48619.2023.00125.

[9] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen. "A Survey of Learning-Based Automated Program Repair." In: *ACM Trans. Softw. Eng. Methodol.* 33.2 (2023). DOI: 10.1145/3631974.

[10] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. Tan. "Automated Repair of Programs from Large Language Models." In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 1469–1481. DOI: 10.1109/ICSE48619.2023.00128.

[11] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy. *Challenges and Applications of Large Language Models*. 2023. DOI: 10.48550/arXiv.2307.10169. arXiv: 2307.10169 [cs].

[12] 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. DOI: 10.48550/arXiv.2312.12148. arXiv: 2312.12148 [cs].

[13] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis." In: *The Eleventh International Conference on Learning Representations*. 2023.

[14] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. "CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih. Association for Computational Linguistics, 2021, pp. 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685.

[15] T. L. Scao et al. "BLOOM: A 176B-parameter Open-Access Multilingual Language Model." In: *Corr* abs/2211.5100 (2022). DOI: 10.48550/ARXIV.2211.05100. arXiv: 2211.05100.

[16] B. Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. DOI: 10.48550/arXiv.2308.12950. arXiv: 2308.12950 [cs].

[17] R. Li et al. "StarCoder: May the Source Be with You!" In: *Transactions on Machine Learning Research* (2023).

[18] D. Guo et al. *DeepSeek-coder: When the Large Language Model Meets Programming – the Rise of Code Intelligence*. 2024. DOI: 10.48550/arXiv.2401.14196. arXiv: 2401.14196 [cs].

[19] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. "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*. Association for Computing Machinery, 2021, pp. 341–353. DOI: 10.1145/3468264.3468544.

[20] S. Ren et al. *CodeBLEU: A Method for Automatic Evaluation of Code Synthesis*. 2020. DOI: 10.48550/arXiv.2009.10297. arXiv: 2009.10297 [cs].

[21] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. "LoRA: Low-Rank Adaptation of Large Language Models." In: *International Conference on Learning Representations*. 2022.

[22] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel. "Few-Shot Parameter-Efficient Fine-Tuning Is Better and Cheaper than in-Context Learning." In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 1950–1965.

[23] R. Macháček, A. Grishina, M. Hort, and L. Moonen. *Replication Package for "The Impact of Fine-tuning Large Language Models on Automated Program Repair"*. Zenodo. 2025. DOI: 10.5281/zenodo.16359186.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention Is All You Need." In: *International Conference on Neural Information Processing Systems (NeurIPS)*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 5998–6008.

[25] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen. *Efficient Training of Language Models to Fill in the Middle*. 2022. DOI: 10.48550/arXiv.2207.14255. arXiv: 2207.14255 [cs].

[26] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen. "APPT: Boosting Automated Patch Correctness Prediction via Fine-Tuning Pre-Trained Models." In: *IEEE Transactions on Software Engineering* 50.3 (2024), pp. 474–494. DOI: 10.1109/TSE.2024.3354969.

[27] Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang. "Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review." In: *ACM Trans. Softw. Eng. Methodol.* 34.1 (2024), 14:1–14:26. DOI: 10.1145/3695993.

[28] Y. Shang, Q. Zhang, C. Fang, S. Gu, J. Zhou, and Z. Chen. "A Large-Scale Empirical Study on Fine-Tuning Large Language Models for Unit Testing." In: *Proc. ACM Softw. Eng.* 2.ISSTA (2025), ISSTA074:1678–ISSTA074:1700. DOI: 10.1145/3728951.

[29] M. Nashaat and J. Miller. "Towards Efficient Fine-Tuning of Language Models with Organizational Data for Automated Software Review." In: *IEEE Transactions on Software Engineering* 50.9 (2024), pp. 2240–2253. DOI: 10.1109/TSE.2024.3428324.

[30] Z. Chen, Y. Deng, Y. Wu, Q. Gu, and Y. Li. "Towards Understanding the Mixture-of-Experts Layer in Deep Learning." In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 23049–23062.

[31] M. Z. Haque, S. Afrin, and A. Mastropaolo. *A Systematic Literature Review of Parameter-Efficient Fine-Tuning for Large Code Models*. 2025. DOI: 10.48550/arXiv.2504.21569. arXiv: 2504.21569 [cs].

[32] A. Aghajanyan, S. Gupta, and L. Zettlemoyer. "Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning." In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Ed. by C. Zong, F. Xia, W. Li, and R. Navigli. Association for Computational Linguistics, 2021, pp. 7319–7328. DOI: 10.18653/v1/2021.acl-long.568.

[33] T. Y. Zhuo, A. Zebaze, N. Suppattarachai, L. von Werra, H. de Vries, Q. Liu, and N. Muennighoff. *Astraios: Parameter-Efficient Instruction Tuning Code Large Language Models*. 2024. arXiv: 2401.00788 [cs].

[34] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui. "Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models." In: *ACM Trans. Softw. Eng. Methodol.* (2025). DOI: 10.1145/3714461.

[35] S. Afrin, J. Call, K.-N. Nguyen, O. Chaparro, and A. Mastropaolo. "Resource-Efficient & Effective Code Summarization." In: *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. 2025, pp. 224–235. DOI: 10.1109/Forge66646.2025.00032.

[36] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu. "TransRepair: Context-Aware Program Repair for Compilation Errors." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2023. DOI: 10.1145/3551349.3560422.

[37] M. Yasunaga and P. Liang. "Break-It-Fix-It: Unsupervised Learning for Program Repair." In: *Proceedings of the 38th International Conference on Machine Learning*. 2021.

[38] C. S. Xia, Y. Wei, and L. Zhang. "Automated Program Repair in the Era of Large Pre-Trained Language Models." In: *Proceedings of the 45th International Conference on Software Engineering*. IEEE Press, 2023, pp. 1482–1494. DOI: 10.1109/ICSE48619.2023.00129.

[39] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung. "VulRepair: A T5-based Automated Software Vulnerability Repair." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2022, pp. 935–947. DOI: 10.1145/3540250.3549098.

[40] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. "Examining Zero-Shot Vulnerability Repair with Large Language Models." In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 2339–2356. DOI: 10.1109/SP46215.2023.10179420.

[41] N. T. Islam, J. Khoury, A. Seong, M. B. Karkevandi, G. D. L. T. Parra, E. Bou-Harb, and P. Najafirad. *LLM-powered Code Vulnerability Repair with Reinforcement Learning and Semantic Reward*. 2024. DOI: 10.48550/arXiv.2401.03374. arXiv: 2401.03374 [cs].

[42] Y. Li, S. Wang, and T. N. Nguyen. "DLFix: Context-Based Code Transformation Learning for Automated Program Repair." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, 2020, pp. 602–614. DOI: 10.1145/3377811.3380345.

[43] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities." In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 479–490. DOI: 10.1109/SANER.2019.8668043.

[44] 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: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020). DOI: 10.1145/3395363.3397369.

[45] F. Huq, M. Hasan, M. M. A. Haque, S. Mahbub, A. Iqbal, and T. Ahmed. “Review4Repair: Code Review Aided Automatic Program Repairing.” In: 143.C (2022). DOI: 10.1016/j.infsof.2021.106765.

[46] Y. Tang, L. Zhou, A. Blanco, S. Liu, F. Wei, M. Zhou, and M. Yang. “Grammar-Based Patches Generation for Automated Program Repair.” In: 2021, pp. 1300–1305. DOI: 10.18653/v1/2021.findings-acl.111.

[47] N. Jiang, T. Lutellier, and L. Tan. “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair.” In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 1161–1173. DOI: 10.1109/icse43902.2021.00107. arXiv: 2103.00073 [cs].

[48] Y. Li, S. Wang, and T. N. Nguyen. “DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair.” In: *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, 2022, pp. 511–523. DOI: 10.1145/3510003.3510177.

[49] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin. “CIRCLE: Continual Repair across Programming Languages.” In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2022, pp. 678–690. DOI: 10.1145/3533767.3534219.

[50] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. “Sk_p: A Neural Program Corrector for MOOCs.” In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. Association for Computing Machinery, 2016, pp. 39–40. DOI: 10.1145/2984043.2989222.

[51] M. Yasunaga and P. Liang. “Graph-Based, Self-Supervised Program Repair from Diagnostic Feedback.” In: *Proceedings of the 37th International Conference on Machine Learning*. JMLR.org, 2020.

[52] K. Abhinav, V. Sharvani, A. Dubey, M. D’Souza, N. Bhardwaj, S. Jain, and V. Arora. “RepairNet: Contextual Sequence-to-Sequence Network for Automated Program Repair.” In: *Artificial Intelligence in Education: 22nd International Conference, AIED 2021, Utrecht, the Netherlands, June 14–18, 2021, Proceedings, Part I*. Springer-Verlag, 2021, pp. 3–15. DOI: 10.1007/978-3-030-78292-4_1.

[53] A. Silva, S. Fang, and M. Monperrus. *RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair*. 2024. DOI: 10.48550/arXiv.2312.15698. arXiv: 2312.15698 [cs].

[54] R. Paul, M. M. Hossain, M. L. Siddiq, M. Hasan, A. Iqbal, and J. C. S. Santos. *Enhancing Automated Program Repair through Fine-Tuning and Prompt Engineering*. 2023. DOI: 10.48550/arXiv.2304.07840. arXiv: 2304.07840 [cs].

[55] D. Sobania, M. Briesch, C. Hanna, and J. Petke. “An Analysis of the Automatic Bug Fixing Performance of ChatGPT.” In: *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE Computer Society, 2023, pp. 23–30. DOI: 10.1109/APR59189.2023.00012.

[56] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. F. Bissyandé, C. L. Goues, and S. Jin. “Multi-Objective Fine-Tuning for Enhanced Program Repair with Llms.” In: *Arxiv Preprint Arxiv:2404.12636* (2024). arXiv: 2404.12636.

[57] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng. “A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Automated Program Repair.” In: *Arxiv Preprint Arxiv:2406.05639* (2024). arXiv: 2406.05639.

[58] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng. “Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair.” In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2024, pp. 719–731. DOI: 10.1145/3691620.3695066.

[59] K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu. “Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair.” In: *IEEE Transactions on Software Engineering* 51.4 (2025), pp. 904–928. DOI: 10.1109/TSE.2025.3532759.

[60] R. Just, D. Jalali, and M. D. Ernst. “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*. Association for Computing Machinery, 2014, pp. 437–440. DOI: 10.1145/2610384.2628055.

[61] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. “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*. Association for Computing Machinery, 2017, pp. 55–56. DOI: 10.1145/3135932.3135941.

[62] M. Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374 [cs].

[63] H. Touvron et al. “Llama 2: Open Foundation and Fine-Tuned Chat Models.” In: *Arxiv Preprint Arxiv:2307.09288* (2023). arXiv: 2307.09288.

[64] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. “Bleu: A Method for Automatic Evaluation of Machine Translation.” In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Ed. by P. Isabelle, E. Charniak, and D. Lin. Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135.

[65] S. Lu et al. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation.” In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. Ed. by J. Vanschoren and S. Yeung. Vol. 1. 2021.

[66] C. Xu, S. Guan, D. Greene, M. Kechadi, et al. “Benchmark Data Contamination of Large Language Models: A Survey.” In: *Arxiv Preprint Arxiv:2406.04244* (2024). arXiv: 2406.04244.

[67] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin. “Automated Patch Correctness Assessment: How Far Are We?” In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020, pp. 968–980. DOI: 10.1145/3324884.3416590.