

# The Art of Repair: Optimizing Iterative Program Repair with Instruction-Tuned Models

Fernando Vallecillos Ruiz  
Simula Research Laboratory  
Oslo, Norway  
fernando@simula.no

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

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

## Abstract

Automatic program repair (APR) aims at reducing the manual efforts required to identify and fix errors in source code. Before the rise of Large Language Model (LLM)-based agents, a common strategy was simply to increase the number of generated patches, sometimes to the thousands, which usually yielded better repair results on benchmarks. More recently, self-iterative capabilities enabled LLMs to refine patches over multiple rounds guided by feedback. However, literature often focuses on many iterations and disregards different numbers of outputs.

We investigate an APR pipeline that balances these two approaches, the generation of multiple outputs and multiple rounds of iteration, while imposing a limit of 10 total patches per bug. We apply three SOTA instruction-tuned LLMs – DeepSeekCoder-Instruct, Codellama-Instruct, Llama3.1-Instruct – to the APR task. We further fine-tune each model on an APR dataset with three sizes (1K, 30K, 65K) and two techniques (Full Fine-Tuning and LoRA), allowing us to assess their repair capabilities on two APR benchmarks: HumanEval-Java and Defects4J.

Our results show that by using only a fraction (<1%) of the fine-tuning dataset, we can achieve improvements of up to 78% in the number of plausible patches generated, challenging prior studies that reported limited gains using Full Fine-Tuning. However, we find that exceeding certain thresholds leads to diminishing outcomes, likely due to overfitting. Moreover, we show that base models greatly benefit from creating patches in an iterative fashion rather than generating them all at once. In addition, the benefit of iterative strategies becomes more pronounced in complex benchmarks. Even fine-tuned models, while benefiting less from iterations, still gain advantages, particularly on complex benchmarks. The research underscores the need for balanced APR strategies that combine multi-output generation and iterative refinement.

## CCS Concepts

• Software and its engineering; • Computing methodologies  
→ Natural language processing;

## Keywords

Automated Program Repair, Software Testing, Software Maintenance, Large Language Models



This work is licensed under a Creative Commons Attribution 4.0 International License.  
*EASE '25, Istanbul, Turkiye*  
© 2025 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-1385-9/25/06  
<https://doi.org/10.1145/3756681.3756966>

## ACM Reference Format:

Fernando Vallecillos Ruiz, Max Hort, and Leon Moonen. 2025. The Art of Repair: Optimizing Iterative Program Repair with Instruction-Tuned Models. In *Evaluation and Assessment in Software Engineering (EASE '25), June 17–20, 2025, Istanbul, Turkiye*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3756681.3756966>

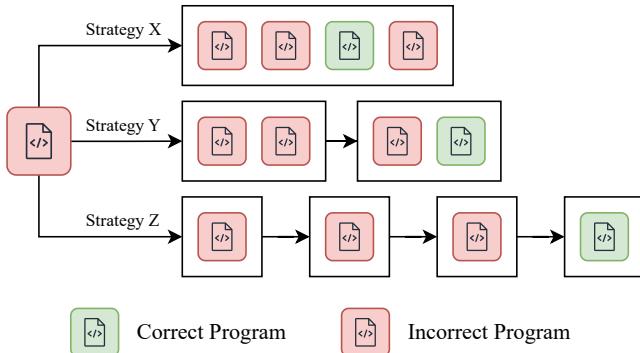
## 1 Introduction

Software bugs are inevitable in the software development cycle, often leading to system failures and increased maintenance costs [11, 33]. Automatic Program Repair (APR) aims to reduce the manual effort required to localize and fix errors in source code. Traditional APR methods can be broadly categorized into pattern-based [18, 21, 22], heuristic-based [30, 36, 56], and constraint-based approaches [6, 28]. However, they often faced challenges in scalability and adaptability since they tended to not generalize beyond a pre-set group of strategies. Consequently, researchers have started to investigate different method to address these limitations.

In recent years, learning-based approaches have tried to address some of these limitations. Neural Machine Translation (NMT)-based tools [26, 48, 49] treat program repair as a translation task from buggy code into correct code. This approach trains on historical bug fixes but it is dependent on the quantity and quality of the data.

Large Language Models (LLMs) have demonstrated promising results in code-related tasks thanks to their extensive pre-training on code repositories. Models such as CodeLlama [35] or DeepSeekCoder [9] have shown high competency in code generation, translation, and completion. The use of LLMs for APR has become an attractive and popular option [39, 58] often outperforming traditional APR methods [18, 21, 22]. Despite their competency, many LLM-based APR approaches generate hundreds or even thousands of patches for each bug [47, 52]. While this approach may improve results, it increases the computational overhead and can overwhelm developers who must sift through these outputs [32].

Recently, instruction-tuning [59] has emerged enabling LLMs to follow commands, improving their ability to perform tasks asked by the user. Instruction-tuned models have been key in the development of LLM-based agents. These agents simulate the cycle of debugging by generating patches, executing them, receiving feedback, and refining their previous answers. They have the potential to improve repair quality by focusing on refining previous answers instead of producing a large quantity of independent patches. However, many of these works introduce complicated agent-based pipelines that involve intricate control flows and arbitrary components. Although these systems achieve even higher performance than their non-agentic counterparts, they also introduce overhead and complexity.



**Figure 1: High-level illustration of different APR strategies: (Strategy X) generating a batch of patches all at once; (Strategy Y) iterating over patches in two rounds; (Strategy Z) iterating over one patch over four rounds.**

In the following paper, our goal is to bridge the gap between generating too many independent patches and relying on overly complex iterative pipelines. We propose a balanced, *developer-centric* approach limiting the number of generated patches to a practical maximum (e.g., ten patches per bug) [32], thereby mirroring real-world constraints while still leveraging iterative refinement. We assess how the size of fine-tuning data and technique used can impact their ability to leverage iterative feedback. By varying these factors, we explore whether a larger dataset always leads to better outcomes or if there are caveats. Additionally, we perform a thorough analysis on different generation strategies (e.g., generating all patches at once, or generating a single patch and iterating over it multiple times) as illustrated in Figure 1. By varying the number of iterations and number of outputs per iteration, we aim to maximize repair success while keeping the number of generated patches low, aligning with constraints faced by developers.

To evaluate our approach, we conduct experiments on two common APR benchmarks: HumanEval-Java [15] and Defects4J [17]. For these two datasets, we apply three state-of-the-art instruction-tuned LLMs: Llama 3.1 [5], CodeLlama [35] and DeepSeek-Coder [9].

The approach proposed focuses on quality over quantity; we are aiming to generate fewer, higher-quality patches so it can be feasible for developers to review them. The study of agentic pipelines provides insights into the optimum generation strategies for iterative tools with the goal of maximizing repair success while keeping the number of generations low. Our contributions in this work are as follows:

- We demonstrate the effectiveness of instruction-tuned LLMs for APR with minimal fine-tuning, challenging prior studies on the limited benefits of FFT and small-scale data.
- We investigate where plausible patches are found within the sequence of generated outputs, providing insights into the contribution of the different positions.
- We develop an iterative pipeline enhancing the capabilities of instruction-tuned LLMs to perform APR through consecutive iterations incorporating execution feedback.

- This work shows how different dataset sizes (1K, 30K, 65K) affect LLM performance on APR, identifying thresholds where further fine-tuning conduces to diminishing results.
- We evaluate seven strategies for patch generation, comparing batch generation vs. iterative refinement to determine the optimal trade-off between number of iterations and outputs per iteration.
- We release a replication package with the complete pipeline to ensure reproducibility and facilitate future use.<sup>5</sup>

The remainder of this paper is organized as follows. Section 2 presents background information on tuning language models as well as related APR approaches. In Section 3, we outline our experimental design. This includes the research question we pursue as well as studied models and datasets. In addition, we describe our implementation details on how we use instruction-tuned models for APR and how to use different iterative strategies to incorporate feedback in the repair process. The experimental results are shown in Section 4. Section 5 addresses threats to validity and Section 6 concludes our study.

## 2 Background and Related Work

### 2.1 Fine-tuning LLMs

Large language models are pre-trained for general purposes, and, although skilled in many tasks, they may not excel in them. Fine-tuning is a crucial step that allows LLMs to adapt to specialized tasks. Traditional fine-tuning adjusts all model parameters based on new task-specific data. As LLMs increase in size, fine-tuning can become computationally costly and susceptible to overfitting [23]. This challenge typically appears in program repair, where collecting diverse high-quality datasets can become difficult.

These issues can be addressed with approaches such as parameter-efficient fine-tuning (PEFT). These methods propose updating only a subset of the parameters in the fine-tuning process. One PEFT method for fine-tuning LLMs is Low-Rank Adaptation (LoRA), proposed by Hu et al. [14]. LoRA reduces the number of trainable parameters by introducing novel weights for training, rather than updating all weights of the model, as done by Full Fine-tuning (FFT).

In addition to fine-tuning LLMs to specialized tasks, they can be tuned to follow instructions. Instruction-tuned LLMs have been applied to various software engineering tasks, such as instructional code editing [2], program synthesis [19, 24] and secure code generation [10]. Moreover, models have been trained on different datasets and made publicly available. Shared instruction-tuned models include the following: WizardCoder [25], OctoCoder [31], InstructCodeT5+ [44], MagiCoder [46], PanguCoder [37], DeepSeek-Coder [9], WaveCoder [55]. The sharing of models enables an easy reuse and application to various tasks and studies.

While various works show the capabilities of instruction-tuning on Natural Language Processing (NLP) tasks, Yuan et al. [57] evaluated 10 open-source models on four code related tasks: defect detection, clone detection, assertion generation, code summarization. The 10 instruction models have been applied to the different tasks under three settings: zero-shot, few-shot and fine-tuned. Their findings showed that fine-tuning instruction LLMs can improve performance over zero- and one-shot setting. Another work, by Zhuo et al. [62], studied which PEFT method should be used for

instruction-tuning. In particular, they considered 7 PEFT methods and 4 model sizes of OctoCoder. Similarly, we set out to investigate the performance of fine-tuning instruction-based models for the program repair task. Additionally, we measure the impact of training choices on fine-tuning performance (using LoRA, varying training data sizes).

## 2.2 Iterative refinement for SE

Successive feedback loops in the generation process have been shown to improve outcomes [3, 8]. This approach is very effective in tasks where initial generations may fail but are close to the desired output. Each cycle contains additional context, helping to refine the output and adjusting it through successive iterations.

Self-feedback is a common approach in which an LLM generates feedback on its own outputs to refine them without external supervision [27]. Alternatively, a different LLM can be specialized on providing feedback for refinement. This multi-agentic approach allows for more complex corrections by specializing an LLM on generating concrete helpful feedback to resolve the remaining problems [16]. In addition to feedback from models, external tools like test suites and compilers can also be used in the refinement process. These tools provide execution feedback, through failed test cases or compilation errors, which the LLM is able to leverage to fix concrete errors in the code [1, 12, 54]. Furthermore, new approaches have been trying to further train LLMs by leveraging execution feedback in real time [7].

Recent work has curated datasets with iterative code-generation goals in mind, featuring multi-turn interaction with human feedback [60]. However, these datasets rely on artificially introduced errors, GPT-generated bugs, rather than real software faults, which can limit their applicability in true APR scenarios. In contrast, our research focuses on analyzing how different factors impact iterative repair, yielding new insights into the trade-offs between generation strategies, data size, and fine-tuning techniques.

## 2.3 LLMs for APR

Jiang et al. [15] investigated four LLMs (PLBART, CodeT5, CodeGen, InCoder) with different number of parameters, and investigated them with and without fine-tuning on a dataset with 143,666 buggy functions and their fix. Fine-tuning improved the number of repaired programs for every model type and size. Moreover, they investigated the impact of sharing information of the buggy line in the prompt, which LLMs can learned to make use of after fine-tuning. They shared their framework for testing LLMs on three APR datasets (Defects4J, HumanEval-Java, Quixbugs).

Silva et al. [38] proposed the fine-tuned model RepairLLaMA and investigated different combinations of code representations for the input and output for LLMs and their impact on APR performance. RepairLLaMA is based on CodeLlama-7b and fine-tuned on the Megadiff dataset [29]. To avoid overfitting, RepairLLaMA was fine-tuned with LoRA, which achieved better performance than full fine-tuning and the use of models without fine-tuning. RepairLLaMA was even able to outperform GPT-4 on the Defects4J v2 dataset.

In contrast to conventional APR approaches with LLMs, which generate a range of patches and validate them on tests, one can perform APR in an iterative fashion. An example of this are the

works by Xia and Zhang [50, 51] who advocated for conversational APR. In their work, patches are validated and failing tests are used as feedback to prompt LLMs again with additional information.

The two most closely related works to ours are from Li et al. [20] and Yang et al. [53]. Li et al. [20] addressed a lack of APR-specific datasets for instruction tuning. Unlike existing datasets to teach language models code instructions, they created the APR-INSTRUCTION dataset focused on the single task of APR and the use of enriched instructions. Training samples are enriched with problem descriptions and bug causes. Proceeding, they fine-tuned four pre-trained base LLMs (CodeLlama-7B, CodeLlama-13B, DeepSeek-Coder-6.7B, Llama-2-7B) on this dataset and have evaluated them on three APR datasets (Defects4J, Quixbugs and HumanEval). Moreover, Li et al. investigated the impact of four PEFT techniques on the performance of fine-tuned models in contrast with full model fine-tuning. In contrast to this work, which used base LLM models and trained them on an APR instruction-dataset, we further fine-tuned an instruction-model for the APR task. Thereby, the language models are able to learn one task at a time (first, the base-model is fine-tuned for instructions and then further tuned for APR), rather than multiple tasks at once (instructions and APR).

Yang et al. [53] proposed a novel multi-objective fine-tuning approach for instruction models on the APR task (MOREPAIR). MOREPAIR is used to fine-tune instruction LLMs to learn 1) to repair code; 2) provide explanations for the repair. In addition to the training procedure MOREPAIR, Yang et al. created TUTORLLMCODE, a dataset of 1600 samples with buggy and fixed codes, as well as guidance written by GPT-4. To fine-tune with fewer parameters, QLoRA is used. MOREPAIR is applied to four LLM (CodeLlama-13B-instruct, CodeLlama7B-instruct, StarChat-alpha, and Mistral-Instruct7B-v0.1) and tested on new benchmarks (EvalRepair-C++ and EvalRepair-Java), which correspond to HumanEval with additional tests. Results show that fine-tuning instruct models with MOREPAIR is able to improve performance over base models, standard fine-tuning, as well as existing works which used larger datasets.

Although Li et al. [20] and Yang et al. [53] rely on LLMs to generate their datasets, our approach leverages an existing dataset, rephrasing the samples (buggy and fixed code) to conform with an instruction format. More importantly, we investigate several repair strategies incorporating execution feedback over multiple rounds, under a practical budget of patches. This real-world constraint ensures that the pipeline is developer-friendly, in contrast to methods that generate a large number of candidate patches. In doing so, this work contributes to a deeper exploration on how fine-tuning, iterations, and constraint patch generation interplay to improve APR outcomes.

## 3 Experimental Design

### 3.1 Research Questions

We address the following research questions in our study:

- (RQ1) What impact does fine-tuning instruction models on automatic program repair have on their performance?
  - (a) How does the size of the dataset impact the model's abilities to repair bugs?
  - (b) Where do models find plausible patches within the sequence of generated outputs?

To address this question, we fine-tune three instruction-tuned models using subsets of three sizes composed of 1K, 30K, and 65K samples from an APR dataset. We evaluate the base and fine-tuned models on two APR benchmarks (Section 3.3). The evaluation focuses on the LLM ability to generate plausible patches - patches that pass the test suite associated with the problem. During the evaluation, we do not only analyze the success rate but also the position within the output sequence where plausible patches were found. We aim to understand the impact of further fine-tuning on the models' repairs by comparing the performance across the different variants of the models and analyze the placement of plausible patches within the generated outputs.

(RQ2) How does the relationship between the number of outputs per iteration and the total number of iterations influence the effectiveness of APR using LLMs?

- How do base and fine-tuned models respond to variations in outputs per iteration and total iterations?
- What is the optimal combination of outputs per iteration and total iterations with a fixed total output limit?

To explore this question, we implement multiple generation strategies consisting of different combinations of number of iterations and number of outputs generated per iteration (Section 3.5). These strategies are applied to base models and fine-tuned models within our iterative repair pipeline, incorporating feedback in each subsequent iteration. We again evaluate the success on the same two APR benchmarks as in RQ1. We assess the success of each strategy by measuring the number of plausible patches generated and further analyze the uniqueness of the problems solved between the strategies.

### 3.2 Models

**Llama 3** [5] is the most recent version of the Llama models [41, 42] created by Meta. In contrast to Llama 2, the models are trained on higher quality and quantity of data (i.e., 15T tokens as compared to 1.8T tokens for Llama 2). After pre-training, several rounds of post-training are performed to align Llama 3 with human instructions. This is achieved by supervised fine-tuning and Direct Preference Optimization. In addition to teaching Llama 3.1 models instructions in a post-processing stage, the models are enhanced with coding capabilities. We chose the 8B parameter variant of the Llama 3 model.<sup>1</sup>

**CodeLlama** [35] presents a specialized version of Llama 2 for coding tasks. In particular, CodeLlama is initialized with Llama 2 models and further trained on 500B tokens from code data present in the Llama 2 training dataset. Overall, three model types are shared, covering four sizes (7, 13, 34, 70B)<sup>2</sup>: foundation/base models, Python specialized models, instruction-based models. For instruction tuning, CodeLlama models are fine-tuned on two data sources: 1) proprietary dataset; 2) self-instruct. The proprietary dataset is based on the instruction-tuning dataset for Llama 2 to teach CodeLlama to follow instructions and abide safety properties. The self-instruct dataset consists of interview-style programming questions created

<sup>1</sup><https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct>

<sup>2</sup><https://huggingface.co/codellama>

by Llama 2 and solutions as well as tests created by CodeLlama. We chose the 7B parameter variant of the CodeLlama model.<sup>3</sup>

**DeepSeek-Coder** [9] presents a range of open-source code models with 1.3B, 6.7B and 33B parameters. The models are based on the DeepSeek LLM architecture [4] and are trained from scratch, on 2 trillion tokens from 87 programming languages. To allow DeepSeek-Coder to understand instructions, the base models have been fine-tuned on instructions following the Alpaca Instruction format [40] and 2B tokens. The performance of DeepSeek-Coder models is competitive, being able to perform similar to larger models or even outperforming GPT-3.5 Turbo in several benchmarks. We chose the 6.7B parameter variant of the DeepSeek-Coder model.<sup>4</sup>

### 3.3 Datasets

*Fine-tuning:* The training data for the fine-tuning process is collected from commits of open-source GitHub Java projects [61]. The dataset contains a total of 143,666 samples of single-hunk fixes. From this larger dataset, we create three subsets consisting of 1K, 30K, and 65K samples. These three different subsets allow for the study of the impact of variant scales of fine-tuning data.

*Benchmarks:* The evaluation of the models is done through two APR benchmarks in Java: Defects4J [17] and HumanEval-Java [15]. Defects4J v2.0 consists of 835 real-world bugs extracted from open-source Java projects. We follow the classification of previous work and select a subset of 217 single-hunk bugs [15]. On the other hand, HumanEval-Java is a bug benchmark containing 164 single-function bugs. Due to its recency, it reduces the risk of data leakage in the pre-training. Both benchmarks contain buggy code and one fix along with unit tests to assess the plausibility of the patches generated.

### 3.4 Implementation

The pipeline for these experiments starts with the problems from the benchmarks. Each input consists of functions where the buggy code is delimited using the tokens <bug\_start> and <bug\_end>. We adopt a beam-based search decoding strategy without stochastic sampling to maintain reproducible and deterministic outputs across all experiments. The pipeline uses the following template to generate the initial prompt for the LLM:

```
"""
The input is buggy code. The bug starts from
'<bug_start>' and ends at '<bug_end>'.
Please fix the following code. Return the fixed
complete method.
```
{buggy_function}
```
"""
```

The output generated is parsed by looking for the triple back-quote symbol (```) that may be followed by the keyword ('``java') to extract the generated code.

In the validation phase, the output generated is inserted into the original problem from the benchmark and the related tests are executed. This execution results in four possible outcomes:

<sup>3</sup><https://huggingface.co/meta-llama/CodeLlama-7b-Instruct-hf>

<sup>4</sup><https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct>

- **Plausible:** All tests pass.
- **Wrong:** At least one test fails.
- **Timeout:** At least one test times out.
- **Uncompilable:** The generated program cannot be compiled.

If the result is plausible, no further processing is done. The other results extract feedback for the next iteration. In case of a wrong or timeout result, the pipeline extracts the name of one of the tests that fails or times out. The pipeline then retrieves the source code of the corresponding test. If the patch is deemed un compilable, the pipeline extracts the compilation errors from the logs. This information is referred to as *feedback* from the validation step.

After the validation step is completed, the pipeline starts the iterative process for the non-plausible patches. If a patch is marked as **Wrong** or **Timeout**, the model is prompted with the previous chat context and extended with the following template:

```
"""
The code is still not correct.
It fails the following test.
"""
```

```
{failed_test_code}
"""
```

```
Fix the original code so it passes the test.
"""
```

If a path is deemed **Uncompilable**, the following template is used:

```
"""
The code is still not correct. It does not compile.
This is the compilation error.
"""
```

```
{compilation_error}
"""
```

```
Fix the original code.
"""
```

This iterative process continues until all problems are solved or the maximum number of patches is generated.

### 3.5 Generation Strategies

Recent work in automatic program repair generates a prohibiting amount of patches, sometimes in the thousands, for every problem [47, 52]. Generating a large number of patches is resource intensive and further increases the computational cost. Furthermore, developers were found to be unlikely to consider more than 10 patches [32]. When designing the experiments, we take into account the practical limitations of computational resources and evaluation time. To address these concerns, we are limiting the number of patches generated per bug to a maximum of 10. This approach not only reduces computational costs, but also aligns with common practices in APR research, allowing comparison with previous work [15, 20, 38, 53].

We explore different generation strategies by varying the number of outputs in the initial generation ( $n_o$ ), the number of outputs in subsequent generations ( $n_i$ ), and the total number of iterations ( $i$ ). We ensure that  $n_o + (n_i \times i) \leq 10$ .

To avoid exponential growth in the number of patches, we focus the iterative process on the first patch generated, as the first patch

represents the output LLMs deem most likely to be correct. We refine the first patch through iterations rather than iterating on all generated patches. This approach ensures that the total number of outputs remains manageable.

- **Strategy A (10×1):** Generate ten outputs in a single iteration.
- **Strategy B (8-2):** Generate eight outputs in the first iteration, and two outputs in the next iteration.
- **Strategy C (5×2):** Generate five outputs per iteration over two iterations.
- **Strategy D (6-2-2):** Generate six outputs in the first iteration, and two outputs in the next two iterations.
- **Strategy E (4-3-3):** Generate four outputs in the first iteration, and three outputs in the next two iterations.
- **Strategy F (2×5):** Generate two outputs per iteration over five iterations.
- **Strategy G (1×10):** Generate one output per iteration over ten iterations.

### 3.6 Evaluation Metric

In our APR pipeline, we evaluate the effectiveness of our approach by measuring the number of problems for which at least one plausible patch is generated. A patch is plausible if it (1) compiles successfully and (2) passes all tests associated with the targeted bug. Otherwise, the patch is considered implausible. We work with 12 models, each generating up to 10 outputs per problem for multiple strategies. Although some of the outputs may overlap between the strategies, the test suite of each problem allows us to efficiently assess tens of thousands of patches across the experiments.

**Manual Assessment and Transparency:** Across all experiments, we produced over 9,000 plausible patches. Given the extensive number of generated patches, manual checking of each one would be prohibitively labor-intensive. To further confirm correctness, we manually inspected 3,298 plausible patches. Of these, 3,167 were confirmed to be correct, while 131 were found to be overfitting to the test suite. To promote transparency and reproducibility, *all* generated patch files, the code including the seed used to randomly sample these patches, and manual assessments are released in our replication package, enabling other researchers and practitioners to examine or extend our work.

## 4 Experimental Results

### 4.1 Results of RQ1

**4.1.1 Influence of data size on fine-tuning.** In RQ1, we investigate the impact of fine-tuning instruction models for APR. We first focus on the influence of the fine-tuning dataset size. For that purpose, we present the results in Table 1 and Table 2 with the number of problems with plausible patches by each model variant using Strategy A (i.e., generating 10 outputs in a single iteration).

The results show that fine-tuning, even using a relatively small dataset, enhances the APR performance of LLMs. For HumanEval-Java, the performance of CodeLlama increased the number of fixed problems to 107 (78% improvement) after FFT with 1K examples. Similarly, an improvement of 70% and 59% is observed for DeepSeek-Coder and Llama3.1 respectively. Although small data sets led to substantial performance gains, increasing the dataset beyond 1K

	CodeLlama	DeepSeek-Coder	Llama3.1
Base	60	76	68
FFT (1K)	<b>107</b>	<b>129</b>	108
FFT (30K)	104	121	<b>113</b>
FFT (65K)	100	122	112
LoRA (1K)	75	79	68
LoRA (30K)	98	<b>128</b>	<b>118</b>
LoRA (65K)	<b>100</b>	126	109

**Table 1: Number of unique problems with at least one plausible patch in HumanEval-Java. The best performing training set size is highlighted per model and training regimen (i.e., FFT and LoRA).**

samples did not always result in improvements. In some models, performance plateaued or even decreased with larger datasets. For instance, CodeLlama FFT and DeepSeek-Coder decreased from 107 and 129 problems solved with 1K examples to 100 and 122 problems with 65K examples respectively. We can observe similar results for the Defects4J datasets. The best performing FFT variants are trained on 1K or 30K samples rather than 65K. These findings suggest that there is a threshold beyond which, results diminish. This problem has been documented for APR [15, 20].

**Finding 1:** Full fine-tuning can achieve large improvements with relatively small datasets, such as 1K samples.

The causes of this can be narrowed down to: 1) Data quality, 2) Overfitting, 3) Limited model capacity. Li et al. [20] curated a higher quality dataset that suffers from the same problem, therefore lowering the likelihood data quality being the cause. Moreover, previous work indicates that instruction-tuned models are good Zero-Shot Learners [45] and LLMs are usually undertrained [13] which would suggest that the issue is not due to limited model capacity. As a result, we believe this problem to be caused by overfitting with large datasets with small degree of variations.

**Finding 2:** Results suggests that there is a threshold beyond which adding more data yields limited improvements, likely due to overfitting.

Our results using LoRA confirm these findings. Both, FFT and LoRA methods, achieved substantial performance increases. However, the number of samples required for each method to achieve successful results differs. While one thousand samples appear to be sufficient for FFT, the LoRA counterparts do not achieve similar results. For example, the best results for Defects4J and LoRA are achieved by training on 65K samples for each of the three models. Previous work on APR focused on PEFT fine-tuning [20, 38] due to promising results compared to FFT. Li et al. [20] performed an analysis on the size of the fine-tuning dataset only for a PEFT approach after discarding FFT given its lower results after being finetuned on a 30K sample dataset. Although our work uses a different dataset, we also achieve similar results on a similar dataset size

	CodeLlama	DeepSeek-Coder	Llama3.1
Base	31	24	28
FFT (1K)	<b>98</b>	97	<b>96</b>
FFT (30K)	85	<b>109</b>	85
FFT (65K)	84	104	82
LoRA (1K)	32	33	36
LoRA (30K)	89	81	93
LoRA (65K)	<b>91</b>	<b>83</b>	<b>103</b>

**Table 2: Number of unique problems with at least one plausible patch in Defects4J. The best performing training set size is highlighted per model and training regimen (i.e., FFT and LoRA).**

for a PEFT approach but not on FFT. Our work shows that similar performance can be achieved with a fraction of the training data if FFT is performed.

**Finding 3:** While prior APR studies saw limited improvements with large datasets using FFT, our results show that it can achieve strong APR performance with less data.

**4.1.2 Position of plausible patches.** We further investigate which of the 10 generated patches, in Strategy A, finds plausible patches. Specifically, we count the positions where the first plausible patch for each problem has been found, to show the contribution of each patch position (i.e., 1-10) on the performance of the models. Figure 2 shows these results for the three models on HumanEval-Java and Defects4J. For each model, we compare the positions of plausible patches generated by the base models and the fine-tuned models. For this purpose, we average the results of the six fine-tuned variants (as shown in Table 1 and Table 2).

For HumanEval-Java, we observe that fine-tuned models find the majority of plausible patches in the first five positions. In total, we found 90%, 92% and 92% of plausible patches in the first five positions for CodeLlama, DeepSeek-Coder and Llama3.1 respectively. When comparing the base model to fine-tuned variants, we can see that the plausible patches generated are focused on the earlier positions, in particular, the first patch accounts for at least 80% of patches as compared to 60% for fine-tuned models.

Plausible patches for Defects4J are more spread out than for HumanEval-Java. For instance, the first five patches account for approximately 80% of the plausible patches found, 10% less than the first 5 patches for HumanEval-Java. Additionally, the performance of the first generated patch by fine-tuned models is almost halved, ranging from 36% to 43% for plausible patches found for the three model types. The base models behave similarly for Defects4J as our observations for HumanEval-Java. Patches are found earlier than with fine-tuned models, and it is rare to find plausible patches beyond the 5th patch.

**Finding 4:** Base models generate plausible patches in earlier positions with almost no contribution from later patches,

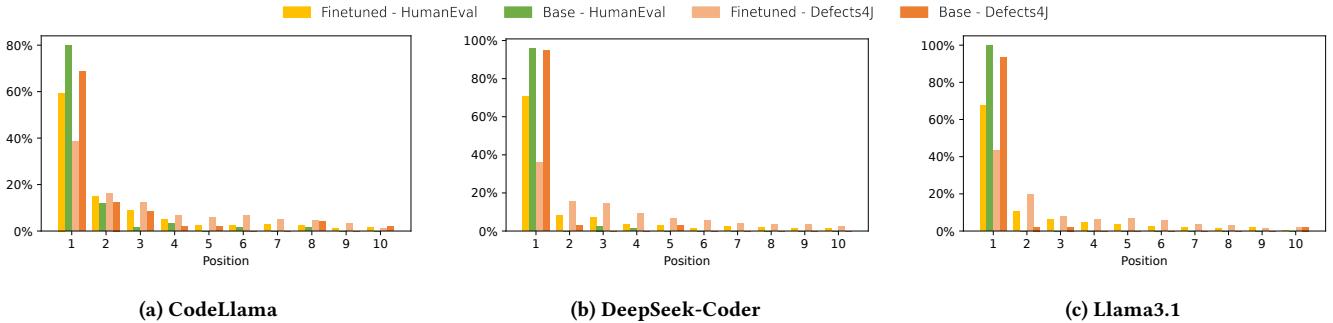


Figure 2: Position of the first plausible patches found for 10 outputs. Results are shown as proportion for all unique plausible patches found summarized over the 6 fine-tuning configurations for each model. The base model is shown separately.

while fine-tuned models obtain plausible patches across the first few outputs.

Overall, we observe that the later generated patches only contribute a fraction of the whole set of solved problems. In particular, the last 4-5 out of the 10 generated patches only account for 10% of the solved problems for fine-tuned models (i.e., 10% of patches are found in the last four patches for Defects4J and last five patches for HumanEval-Java). For base models, there are almost no plausible outputs found in the later patches, with the majority of patches found in the first position. This leads us to believe that, given a budget of 10 patches for evaluation, one can improve upon the standard practice of generating all 10 patches at once by following an iterative process. We investigate this in RQ2.

**Finding 5:** Efforts should be concentrated on early outputs, as later patches contribute less than 10% to the overall repair success.

## 4.2 Results of RQ2

In RQ2, we investigate the influence of fine-tuning on iterative automatic program repair. In addition, we study the influence of number of iterations and number of outputs per iteration when using iterative LLM-based agents for APR while limiting the number of evaluated patches to 10. In accordance with RQ1, we start analyzing the results on HumanEval-Java since it prevents data-leakage and consists of simpler cases, and then move onto more complex problems in the Defects4J benchmark.

To limit the models we investigate, we consider a total of three out of the seven studied configurations for each of the LLMs (CodeLlama, DeepSeek-Coder, Llama3.1). In particular, we chose the base model as well as the best performing variants trained with FFT and LoRA on HumanEval-Java according to RQ1.

**4.2.1 Influence of fine-tuning on iterations.** To assess the influence of fine-tuning on the models' ability to integrate iterative feedback, we use two strategies: Strategy A and Strategy G. We have selected Strategy A as a baseline. Strategy A does not iterate and directly generates 10 patches, while strategy G iterates 10 times over the

generated patch. The results are illustrated in Figure 4a and Figure 4b.

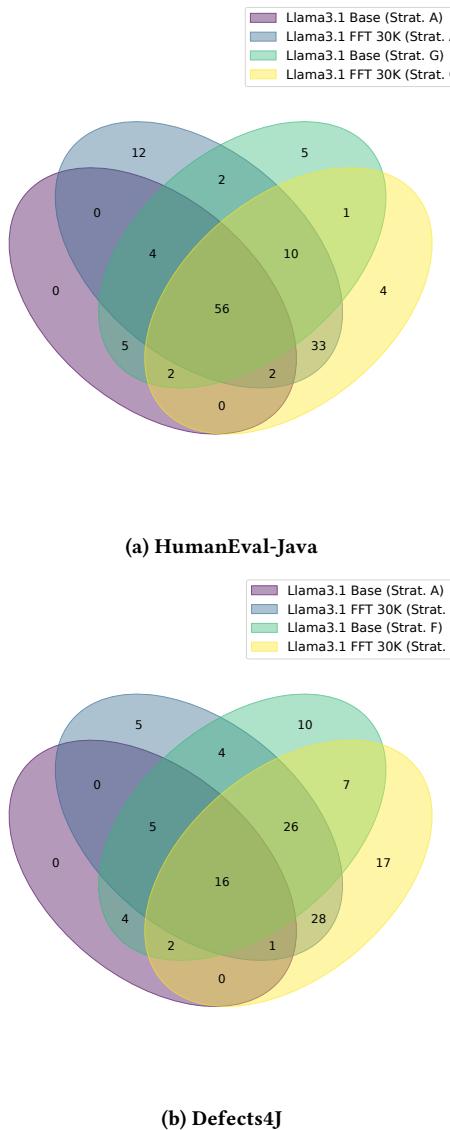
The base models show a consistently improved performance when iterative feedback is incorporated. This suggests that the feedback is successfully incorporated in successive attempts when failing to generate plausible patches.

In contrast, fine-tuned models achieve their highest performance through fewer iterations. The number of plausible solutions generated decreased substantially when iterations are added. This trend is observed across the different models and fine-tuning approaches. This decreased performance suggests that further fine-tuning of models not only enhances the model's initial solution quality, but it may also reduce their ability to leverage iterative feedback effectively. A possible explanation is that fine-tuned models quickly become overfitted to a single task. This would make the models reduce the probability of having diverse outputs in the initial or subsequent iterations. Alternatively, the fine-tuning process may over-specialize models to perform well under non-iterative conditions, therefore reducing the zero-shot capabilities of the model of incorporating feedback.

**Finding 6:** While iterative feedback consistently improve the performance of base models, fine-tuned models solve a higher number of problems but display reduced effectiveness with iterations, likely due to overfitting.

To further analyze the impact of fine-tuning on the iterative capabilities of the model, we study the problems solved by a base-model and its fine-tuned counterpart. For simplicity, we have chosen one model, Llama3.1, however, all results are disclosed in the replicability package. We illustrate the problems solved with the different strategies in Figure 3.

The diagram reveals that while the fine-tuned model solves a higher number of problems, there are still problems that only an iterative approach through a base model can solve. In the case of HumanEval-Java, there are 10 unique problems that Strategy G with a base model can solve that the fine-tuned version are not able to. In other words, almost 12% of the plausible solutions proposed, are unique to this combination. This percentage increases to 19% in the Defects4J benchmark when comparing the strategies with the most and least number of iterations.



**Figure 3: Venn diagram of problems with plausible patches generated by the variants of Llama3.1 comparing the least and most iterative strategies applied for each benchmark.**

**Finding 7:** While fine-tuned models solve a greater number of problems, there are certain unique problems that only iterative models can address.

This suggest that there are certain problems that inherently benefit from iterative feedback, and the fine-tuned model's ability to solve these is limited. While fine-tuning greatly enhanced the overall efficiency by improving the initial candidates, it can reduce the ability of the model to leverage iterative feedback. Therefore, combining fine-tuned models with iterative strategies and designing fine-tuning processes to preserve its zero-shot flexibility would lead

to more robust APR models. To address the limitation of base and fine-tuned models, we investigate how adjustments in the iterative framework can influence APR effectiveness.

**Finding 8:** Combining fine-tuned models with iterative strategies and/or defining fine-tuning processes to preserve zero-shot flexibility may lead to more versatile and robust APR models.

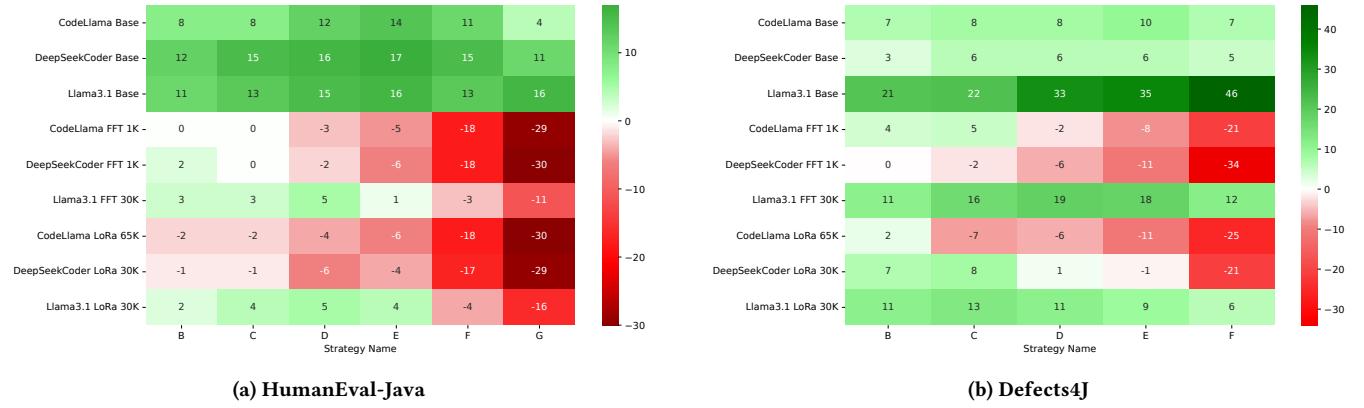
**4.2.2 Influence of different strategies.** In addition to the Strategy G, we investigate the remaining iterative strategies, ranging from two to five iterations (see Section 3.5). In terms of the problems solved for each of the strategies, we have identified key trends that are shared between the benchmarks from Figure 4.

**Base Models' Performance on HumanEval-Java:** For base models, increased iterations are beneficial. Strategies that emphasize fewer outputs per iteration but spread them over multiple ones consistently led to improved results. However, the highest number of iterations did not always equal better results even for base models. Strategies like D and E, which balanced the number of iterations and number of outputs per iteration led to the better average performance across the models. Strategies F and G, which increased iterations even further, led to diminishing returns in some of the models. This suggests that while iterative refinement often leads to improvement in base models, excessive iterations with minimal output per iteration becomes counterproductive.

**Fine-Tuned Models' Performance on HumanEval-Java:** Fine-tuned models, with FFT and LoRA, performed best when more outputs were generated in the initial iterations. Strategies A and B consistently yield the highest number of plausible patches. When the number of iterations are increased, the performance quickly decreased. The decrease indicates that fine-tuned models have a high probability of producing plausible patches early on. Additionally, their ability to incorporate feedback is substantially reduced compared to base models, in accordance with previous sections.

**Finding 9:** Base models benefit from iterative strategies, but reach a point of diminishing returns when iterations are increased too much.

**Resource Allocation Implications:** The selected strategy has strong implication on the computational efficiency of the generation process. Generating 10 outputs in the same inference is less computationally intensive than obtaining 10 outputs via 10 consecutive iterations. Furthermore, when incorporating feedback in each new iteration, the GPU memory needed for the inference increases. Given the length of the problems in the Defects4J benchmark, the inference for one single problem may take up to 200GB of GPU memory after repeated iterations. While GPU memory may be a constraint in certain applications, testing can become a bottleneck in some contexts. Therefore, the selection of the strategies is always depending on the context and the specific bottlenecks present. Due to these considerations, we decided that the resource requirement for increased iterations outweighed the marginal gains in performance. Consequently, we decide to drop Strategy G for the Defects4J benchmark. This decision allowed us to still provide a



**Figure 4: Impact of different generation strategies on the number of unique problems with at least one plausible patch. The heatmaps show the difference in the number of plausible patches with regard to the default strategy (Strategy A).**

balanced view of the different strategies without excessive GPU memory demands.

**Finding 10:** Increasing the number of outputs per iteration while minimizing the number of total iterations reduces the peak memory usage, making certain generation strategies more efficient if testing is not a constraint.

**Base Models’ Performance on Defects4J:** We see similar trends where more iterations lead to improved performance for base models. For instance, Llama3.1 Base increases its number of plausible patches from 28 under Strategy A to 74 under Strategy F. This trend suggests that the complexity of Defects4J may be amplifying the benefits of iterative refinement. Similar to the previous benchmark, the other base models do not achieve their highest number plausible patches with the highest number of iterations, reemphasizing that each model may have different optimum strategies.

**Fine-Tuned Models’ Performance on Defects4J:** A similar trend appears for fine-tuned models, where an increase in iterations does not always result in improved outcomes. In some cases, performance declines as iterations increase, with the negative impact becoming more and more pronounced. For example, DeepSeekCoder FFT 1K generates 97 plausible patches under Strategy A, but drops to 63 under Strategy F. CodeLlama and DeepSeekCoder still present improved performance with low number of iterations. However, Llama3.1 shows an improvement with a higher number of iterations, but its peaked performance is reached in Strategy C and D. These results show that even fine-tuned models may still benefit from iterations if the problems are complex enough to require them.

**Contrasting Trends Between Benchmarks:** We have showcased that different strategies are optimal for each benchmark. On the simpler and straightforward HumanEval-Java benchmark, base models showed improvement while fine-tuned models do not benefit from increased iterations. On the other hand, Defects4J consists of challenging and complex problems that benefit from iterative strategies. Base models significantly improved with more iterations, while fine-tuned models peaked earlier. These results emphasize

that, while most fine-tuned models can address simpler bugs with minimal iterations, base models can obtain more substantial gains from the iterative refinement.

**Finding 11:** Complex problems benefit greatly from iterative refinement, but each model has an optimal iteration threshold after which gains decline.

## 5 Threats to validity

One key internal threat is the potential data leakage from the benchmarks into the pretraining data of the models, especially for older widely-known benchmarks like Defects4J. We mitigate this threat by assessing the models on a complementary recent benchmark specifically designed to address data leakage (i.e., HumanEval-Java), and by including models like Llama3.1 which have shown less susceptibility to memorization [34]. Another internal validity threat is the introduction of bias when selecting hyperparameters in the fine-tuning and generation. We address this by using standardized hyperparameters and generation setting across the different models, thus reducing performance variability attributable to tuning decisions.

The main external threat is the use of two benchmarks on the same programming language, which may limit the applicability of our approach. To mitigate this threat, we select benchmarks that are widely known in the literature and cover a range of real-world bug types. The insights in this paper should generalize to arbitrary programming languages.

A key construct validity threat arises from our reliance on plausibility, a binary metric where a patch is considered successful if it compiles and passes all the tests. Although practical, this metric does not guarantee true correctness of the patch, since tests may not cover all edge cases and outputs may overfit to the tests. To partially address this limitation, we have manually analyzed over 3,000 of the 9,000 plausible patches generated across our experiments. Manual checks are labor-intensive and subjective, since reviewers may apply different standards [43]. We have published our manual assessments alongside the patches in our replication package.<sup>5</sup> This

step promotes transparency and has the potential to reduce future validation efforts. Another construct threat relates to the stopping criteria. The maximum number of iterations or outputs per iteration could impact the evaluation. To mitigate this threat we selected the fixed number of outputs based on empirical observations from related work [15, 20, 38, 53].

## 6 Conclusion

In this work, we investigated the effectiveness of instruction-tuned LLMs in APR tasks through an iterative repair pipeline. We focused on three state-of-the-art models, CodeLlama, DeepSeekCoder, and Llama3.1. With the help of these models, we studied the position of the plausible patches within the generated batch. We explored how fine-tuning said models with varying size of APR datasets impacts their effectiveness. This work study the different generation strategies to balance the number of iterations and the number of outputs per iteration.

Our experiments on two widely recognized APR benchmark, HumanEval-Java and Defects4J, revealed that FFT with relatively small datasets can lead to substantial improvements in repair performance. This suggest that limiting the amount of data in the fine-tuning can enhance the model's ability to generate plausible patches. In other words, increasing the fine-tuning dataset size did not consistently yield better results and, in some cases, led to declined performance. Moreover, the analysis of different generation strategies showed that base models benefit from an increased number of iterations. Yet, we find also an upper threshold for the number of iterations beyond which the process became counter-productive due to excessive resource demands with diminishing returns. On the other hand, fine-tuned models generated plausible patches in earlier iterations and benefited from less iterative feedback, especially on simpler tasks. The optimal generation strategy varies depending on the bottlenecks of the process and the complexity of the task: while fine-tuned models achieved their best results with fewer iterations, base models improved their outcomes substantially with higher number of iterations.

Resource efficiency was a crucial factor in our study. By focusing on generating fewer patches, our approach aligns with real-world developer constraints and enhances computational efficiency. While strategies that minimize iterations reduced memory usage, incorporating feedback in subsequent iterations was able to generate plausible patches for unique problems that fine-tuned models could not. Our emphasis on resource-efficient generation ensures that the benefits of APR using LLMs can be put into practice without imposing excessive computational costs.

This study contributes with insights into the optimization of instruction-tuned LLMs for APR tasks. The work presented highlights the importance of selecting an appropriate generation strategy based on the fine-tuning specification of the models and the complexity of the problem. By displaying that significant performance gains can be achieved with small datasets and suitable generation strategies, we present a path towards accessible and efficient APR. This work does not only advance the field of APR, but it also provides practical guidance for the deployment of LLMs in complex pipelines.

## 6.1 Future Work

This study opens multiple avenues for future research. Methods and regularization to mitigate overfitting when fine-tuning with larger datasets could further improve model performance. Additionally, exploring hybrid generation strategies that can dynamically adapt based on the problem complexity and feedback incorporated can lead to enhanced repair effectiveness.

Incorporating different types of feedback, such as from developers, and integrating these models into development tools can narrow the gap between research and industry. By continuing to refine models and methodologies in an efficient manner, we can move closer to realizing the potential of automatic program repair, contributing to the end-goal of reliable and efficient software development process.

## 7 Data Availability

The replicability package for this work is available online.<sup>5</sup> This package includes (1) the source code required to replicate the experiments presented in this work, (2) the generated patches for both benchmarks by all models used, and (3) the nine fine-tuned models.

## 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. In addition, we acknowledge Sigma2, Norway for awarding this project access to the LUMI supercomputer, owned by the EuroHPC Joint Undertaking, hosted by CSC (Finland) and the LUMI consortium through the Research Council of Norway.

## References

- [1] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs] doi:10.48550/arXiv.2403.17134
- [2] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, et al. 2024. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. arXiv:2312.12450 [cs] doi:10.48550/arXiv.2312.12450
- [3] Xinyun Chen, Maxwell Lin, Nathanael Schärlí, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs] doi:10.48550/arXiv.2304.05128
- [4] DeepSeek-AI, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, et al. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. arXiv:2401.02954
- [5] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs]
- [6] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST '16)*. Association for Computing Machinery, New York, NY, USA, 85–91. doi:10.1145/2896921.2896931
- [7] Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2024. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. URL: <https://arxiv.org/abs/2410.02089v1>.
- [8] Anastasia Grishina, Vadim Liventsev, Aki Härmä, and Leon Moonen. 2025. Fully Autonomous Programming Using Iterative Multi-Agent Debugging with Large Language Models. *ACM Trans. Evol. Learn. Optim.* 5, 1 (March 2025), 8:1–8:37. doi:10.1145/3719351

<sup>5</sup><https://doi.org/10.5281/zenodo.1529469>

[9] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs] doi:10.48550/arXiv.2401.14196

[10] Jingxuan He, Mark Vero, Gabriela Krasnopsolska, and Martin Vechev. 2024. Instruction Tuning for Secure Code Generation. arXiv:2402.09497 [cs]

[11] Krasner Herb. 2021. *Cost of Poor Software Quality in the U.S.: A 2020 Report*. Technical Report. Consortium for Information and Software Quality.

[12] Dávid Hidvégí, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. 2024. CigaR: Cost-efficient Program Repair with LLMs. arXiv:2402.06598 [cs] doi:10.48550/arXiv.2402.06598

[13] Jordan Hoffmann, Sébastien Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, et al. 2022. Training Compute-Optimal Large Language Models. arXiv:2203.15556 [cs] doi:10.48550/arXiv.2203.15556

[14] 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:2106.09685 [cs] doi:10.48550/arXiv.2106.09685

[15] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 1430–1442. doi:10.1109/ICSE48619.2023.00125

[16] Haolin Jin, Zechao Sun, and Huaming Chen. 2024. RGD: Multi-LLM Based Agent Debugger via Refinement and Generation Guidance. arXiv:2410.01242 [cs]

[17] René Just, Dariush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, San Jose, CA, USA, 437–440. doi:10.1145/2610384.2628055

[18] Anil Koyuncu, Kui Liu, Tegawende F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Software Engineering* 25, 3 (May 2020), 1980–2024. doi:10.1007/s10664-019-09780-z

[19] Kirby Kuznia, Swaroop Mishra, Mihir Parmar, and Chitta Baral. 2022. Less Is More: Summary of Long Instructions Is Better for Program Synthesis. arXiv:2203.08597 [cs] doi:10.48550/arXiv.2203.08597

[20] Guochang Li, Chen Zhi, Jialiang Chen, Junxiao Han, and Shuguang Deng. 2024. A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Automated Program Repair. arXiv:2406.05639 [cs]

[21] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandé. 2019. 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, Hangzhou, China, 1–12. doi:10.1109/SANER.2019.8667970

[22] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Beijing China, 31–42. doi:10.1145/3293882.3330577

[23] Junyi Lu, Lei Yu, Xiaojie Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. arXiv:2308.11148 [cs]

[24] Xianzhen Luo, Qingfu Zhu, Zhiming Zhang, Xu Wang, Qing Yang, Dongliang Xu, and Wanxiang Che. 2024. Semi-Instruct: Bridging Natural-Instruct and Self-Instruct for Code Large Language Models. arXiv:2403.00338 [cs]

[25] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568 [cs] doi:10.48550/arXiv.2306.08568

[26] 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 *SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 101–114. doi:10.1145/3395363.3397369

[27] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, et al. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 46534–46594.

[28] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. arXiv:1712.03854 doi:10.48550/arXiv.1712.03854

[29] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size. arXiv:2108.04631 [cs] doi:10.48550/arXiv.2108.04631

[30] Manish Motwani and Yuriy Brun. 2023. Better Automatic Program Repair by Using Bug Reports and Tests Together. arXiv:2011.08340

[31] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction Tuning Code Large Language Models. arXiv:2308.07124 [cs]

[32] Yannic Noller, Ridwan Sharifdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. arXiv:2108.13064 [cs]

[33] Devon H. O'Dell. 2017. The Debugging Mindset: Understanding the Psychology of Learning Strategies Leads to Effective Problem-Solving Skills. *Queue* 15, 1 (Feb. 2017), 71–90. doi:10.1145/3055301.3068754

[34] Daniel Ramos, Claudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2025. Are Large Language Models Memorizing Bug Benchmarks? arXiv:2411.13323 [cs] doi:10.48550/arXiv.2411.13323

[35] Baptiste Rozière, Jonas Gehring, Fabian Gloclekle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, et al. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs] doi:10.48550/arXiv.2308.12950

[36] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE, Urbana-Champaign, IL, USA, 648–659.

[37] Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, et al. 2023. PanGu-Coder2: Boosting Large Language Models for Code with Ranking Feedback. arXiv:2307.14936 [cs]

[38] Andre Silva, Sen Fang, and Martin Monperrus. 2025. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *IEEE Transactions on Software Engineering* Early Access (2025), 1–16. doi:10.1109/TSE.2025.3581062

[39] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. arXiv:2301.08653 [cs] doi:10.48550/arXiv.2301.08653

[40] Rohan Taori, Ishaaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-Following LLaMA Model. URL: [https://github.com/tatsu-lab/stanford\\_alpaca](https://github.com/tatsu-lab/stanford_alpaca).

[41] Hugo Touvron, Thibaut Lavit, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambo, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs] doi:10.48550/arXiv.2302.13971

[42] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs] doi:10.48550/arXiv.2307.09288

[43] 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 *35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 968–980. doi:10.1145/3324884.3416590

[44] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2305.07922 [cs] doi:10.48550/arXiv.2305.07922

[45] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned Language Models Are Zero-Shot Learners. arXiv:2109.01652 [cs] doi:10.48550/arXiv.2109.01652

[46] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering Code Generation with OSS-INSTRUCT. In *Proceedings of the 41st International Conference on Machine Learning (ICML '24)*. Vol. 235. JMLR.org, Vienna, Austria, 52632–52657.

[47] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. arXiv:2303.10494 [cs] doi:10.48550/arXiv.2303.10494

[48] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *45th International Conference on Software Engineering (ICSE '23)*. IEEE, Melbourne, Victoria, Australia, 1482–1494. doi:10.1109/icse48619.2023.00129

[49] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 959–971. doi:10.1145/3540250.3549101

[50] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. arXiv:2301.13246 [cs] doi:10.48550/arXiv.2301.13246

[51] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 Bugs for \$0.42 Each Using ChatGPT. arXiv:2304.00385 [cs] doi:10.48550/arXiv.2304.00385

[52] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Haotian Zhang, and Yiqun Zhang. 2024. How Far Can We Go with Practical Function-Level Program Repair? arXiv:2404.12833 [cs]

[53] Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawende F. Bissyandé, Claire Le Goues, and Shunfu Jin. 2024. Multi-Objective Fine-Tuning for Enhanced Program Repair with LLMs. arXiv:2404.12636 [cs]

[54] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3623337

[55] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiu Feng Yin. 2024. WaveCoder: Widespread And Versatile Enhancement For Code Large Language Models By Instruction Tuning. arXiv:2312.14187 [cs]

[56] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (Oct. 2020), 1040–1067. doi:10.1109/TSE.2018.2874648

[57] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating Instruction-Tuned Large Language Models on Code Comprehension and Generation. arXiv:2308.01240 [cs]

[58] 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 (Dec. 2023), 55:1–55:69. doi:10.1145/3631974

[59] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, et al. 2024. Instruction Tuning for Large Language Models: A Survey. arXiv:2308.10792 [cs]

[60] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhui Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 12834–12859. doi:10.18653/v1/2024.findings-acl.762

[61] 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 *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 341–353. doi:10.1145/3468264.3468544

[62] Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. 2024. Astraios: Parameter-Efficient Instruction Tuning Code Large Language Models. arXiv:2401.00788 [cs]