RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair

André Silva*
KTH Royal Institute of Technology
Stockholm, Sweden
andreans@kth.se

Sen Fang*
KTH Royal Institute of Technology
Stockholm, Sweden
senf@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
monperrus@kth.se

ABSTRACT

Automated Program Repair (APR) has evolved significantly with the advent of Large Language Models (LLMs). Fine-tuning LLMs for program repair is a recent avenue of research, with many dimensions which have not been explored. Existing work mostly fine-tunes LLMs with naive code representations and is fundamentally limited in its ability to fine-tune larger LLMs. To address this problem, we propose RepairLLaMA, a novel program repair approach that combines 1) code representations for APR and 2) the state-of-the-art parameter-efficient LLM fine-tuning technique called LoRA. This results in RepairLLaMA producing a highly effective 'program repair adapter' for fixing bugs with language models. Our experiments demonstrate the validity of both concepts. First, fine-tuning adapters with program repair specific code representations enables the model to use meaningful repair signals. Second, parameter-efficient fine-tuning helps fine-tuning to converge and contributes to the effectiveness of the repair adapter to fix data-points outside the fine-tuning data distribution. Overall, RepairLLaMA correctly fixes 125 Defects4J v2 and 82 HumanEval-Java bugs, outperforming all baselines.

KEYWORDS

Automated Program Repair, Large Language Models, Code Representations, Parameter-Efficient Fine-Tuning

1 INTRODUCTION

Automated program repair (APR) [26] aims at automatically fixing a software bug without human intervention. Learning-based repair [7, 17, 18, 39, 42, 44, 47, 52] has become the mainstream solution to this problem due to the powerful ability of deep neural networks to learn complex bug fix patterns. Clearly, large language models (LLMs), pre-trained on vast amounts of data, have pushed learning-based repair to the next frontier [17, 43]. In program repair, LLMs have been mostly used with prompt engineering [21, 45], and recently, a line of work around fine-tuning has emerged [17, 18, 39, 49].

Fine-tuning LLMs for program repair is complex. Early work simply refines the network weights based on additional fine-tuning data. However, this kind of fine-tuning is rather primitive and suffers from two significant drawbacks. First, fine-tuning is also known to be able to adapt the input/output representations of the data under study [5]. In the context of program repair, there is an opportunity to fine-tune with code representations that maximize downstream task performance, that is, repair performance. In particular, previous work overlooks the realistic representation of fault localization in the input. Second, previous work considered the most basic fine-tuning technique, which is full-parameter fine-tuning. As LLMs increase in size [24], full-parameter fine-tuning poses important overfitting problems when fine-tuning data is limited, which is typically the case in program repair. In this paper, we address the problem of devising efficient fine-tuning techniques [14] for program repair, with a focus on code representations and adapters.

We propose RepairLLaMA, a new program repair approach that combines realistic repair-specific code representations with parameter-efficient fine-tuning. First, RepairLLaMA's code representations incorporate fault localization signals and are designed to support multi-location bugs. Second, RepairLLaMA utilizes Low-Rank Adaption (LoRA), a state-of-the-art parameter-efficient fine-tuning technique, to train a much smaller *repair adapter* (when compared to the full LLM) that adapts the LLM for program repair while helping prevent overfitting. As we will demonstrate in this paper, the concept of *repair adapter* is novel and potent.

Our experimental results validate RepairLLaMA's core designs. First, RepairLLaMA achieves state-of-the-art performance in two benchmarks, correctly fixing 125 Defects4J v2 [22] bugs and 82 bugs on recently proposed HumanEval-Java [17], which boosts internal and external validity. The experiments show that the devised code representations with repair signals allow the LLM to synthesize patches more effectively than with a naive code-only representation. Also, RepairLLaMA clearly outperforms non-fine-tuned baselines, incl. GPT-4. Moreover, our results also show the effectiveness of parameter-efficient fine-tuning: RepairLLaMA's repair adapters, with only 4M parameters, are 1600x smaller than the initial pre-trained LLM (CodeLLama-7B). To sum up, the efficient representations and repair adapters of RepairLLaMA outperform recent results on fine-tuning for program repair [15, 17, 39] as well as world-class models such as GPT-3.5 and GPT-4.

To sum up, we make the following contributions:

- We design RepairLLaMA, an original fine-tuning pipeline for automated program repair, that maximizes knowledge from the program repair domain, while keeping strong alignment with pre-training.
- We systematically evaluate code representations for program repair fine-tuning. Our results clearly show that the best code representation leverages task-specific signals, including fault localization and original buggy code.
- We demonstrate that parameter-efficient fine-tuning performs better than full-parameter fine-tuning in the context of program repair. The "repair adapters" of RepairLLaMA are training-efficient, inference-efficient, and powerful to repair bugs, achieving state-of-the-art performance across two

*These authors contributed equally to this work.

benchmarks, Defects4J and HumanEval-Java, outperforming even GPT-4.
- For the sake of open science, we publish our source code, models, and artifacts at https://anonymous.4open.science/r/repairllama-BC13 and provide a demo website at https://repairllama.github.io.

2 REPAIRLLAMA: EFFICIENT FINE-TUNING FOR PROGRAM REPAIR

2.1 Overview

Figure 1 illustrates the pipeline of RepairLLaMA for APR, which is divided into three consecutive stages. The core novelties of this pipeline are: 1) the APR specific code representations and 2) the end-to-end use of a parameter-efficient fine-tuning technique.

The core of RepairLLaMA is a *repair adapter*. A *repair adapter* is a plug-and-play extension of the model parameters that modifies the behavior of the LLM in order to maximize performance on the repair task, for a given programming language. The adapter is responsible for transforming a rich, tailored input representation of the buggy code into the fit output representation of the patch.

In the first stage of RepairLLaMA, the core choices are made, namely: 1) the initial pre-trained model (subsection 2.3); 2) the input code representation and output code representation (subsection 2.4); and 3) the fine-tuning dataset (subsection 2.5). These choices are all important and are further discussed in the remainder of this section.

In the second stage, a repair adapter is trained. The repair adapter is a much smaller (i.e., approx. 4M parameters) plug-and-play adapter of the initial LLM while remaining competitive on the task of program repair.

Finally, in the third stage, the repair adapter is employed to fix real-world bugs.

2.2 Target Bugs

The first consideration when designing a fine-tuning pipeline for program repair is the bugs we aim to fix. This relates to 1) the programming language, 2) the type of bugs (syntax errors, runtime errors, functional errors, etc), and, 3) the difficulty of bugs, which can be proxied by the code span to modify in order to fix the bug.

In this work, we focus on 1) Java bugs, 2) that are functional, and come with at least a failing test case, and, 3) that are intra-procedural with any length of the code span, i.e. that can be fixed with changes to a single function (called hereafter *single-function bugs*).

Note that we do want to support bugs that require changes in multiple locations in the function [48], beyond single-line or single-chunk bugs.

2.3 Choice of the Initial LLM

Choosing the suitable initial LLM for fine-tuning is crucial. For example, when fine-tuning for code-related tasks, an LLM pre-trained on large-scale code corpora is more effective than one pre-trained on pure natural language data. To effectively fine-tune an LLM for APR, we curate three criteria to choose the initial model.

First, the LLM should be publicly available and open-source. Fine-tuning a closed-source LLM on the task-specific dataset is not a valid option. Although some companies like OpenAI do provide an API for fine-tuning their LLMs, it is expensive, and the ownership of the final model (incl. weights) does not meet open-science reproduction criteria. Open-source models, such as LLaMA [36] or StarCoder [23], publish model weights online, allowing anyone to modify and deploy them.

Second, the LLM should be pre-trained with large-scale code data. As observed by related work [23, 32], LLMs pre-trained on massive code data achieve better performance in code-related tasks. Thus, we consider only LLMs specialized on code.

Third, the initial LLM should have been trained with an infilling objective [3] during pre-training. As observed by related work [17], infilling is a natural and effective learning objective for the program repair task, since it allows the model to synthesize code according to both the context appearing before and after. It should also be supported by an off-the-shelf parameter-efficient fine-tuning library.

In subsection 3.2 we instantiate those criteria in the context of functional program repair for Java.

2.4 Choice of Code Representations

Source code representation is a critical aspect that significantly impacts the effectiveness of the model [29]. In this section, we discuss key characteristics of the source code representation design space. We introduce, motivate, and elaborate on input and output code representations specific to the program repair task.

2.4.1 Representation of Fault Localization. Virtually all the APR literature assumes line-based fault localization, with a single line given as input to the repair algorithm. This is not appropriate to fix multi-location bugs [33, 48]. Consider Figure 3 (OR4), which shows the canonical patch for the multi-location bug Chart-5 from Defects4J. In this case, fault localization must identify a location where an entirely new if block should be synthesized and inserted as well as another pre-existing if condition, appearing later in the code. To our knowledge, there is no fault localization technique able to predict tuples of blocks to be repaired together.

In this paper, we propose a novel way to represent fault localization information: our core idea is to represent fault localization not as a single line, but as a region. In RepairLLaMA, we encode fault localization as a span ranging from the beginning of the suspicious region to its end. This encoding is realistic because 1) identifying a buggy method is within reach of existing fault localization methods, and 2) exhaustively listing all suspicious code regions of a buggy method is worst-case $O(n^2)$ in the number of method lines.

2.4.2 Input Representation Space. In APR, the design space of the input representation relates to what is shown from the buggy code and to the presence of additional information. For example, fault localization signals can be useful in scoping down where the code should be modified. However, such information might not be seen at the pre-training stage. For the LLM to utilize it, one must represent it in a way that it can learn during fine-tuning. To study the input representation space, we design four input representations tailored to APR (Figure 2):

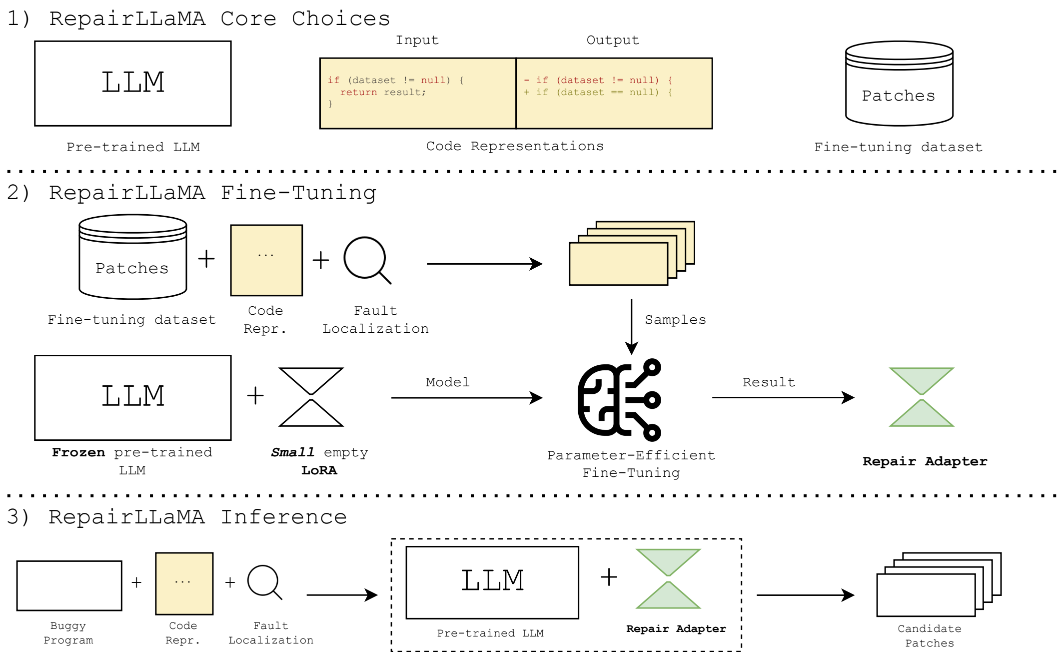

Figure 1: Overview of RepairLLaMA. The core novelties of RepairLLaMA are the APR specific code representations and the engineering of an effective *program repair adapter* that is plugged into the underlying LLM.

IR1: Buggy function This naive representation describes the code in the standard format as it is written, simply as text. Figure 2 (IR1) shows the buggy function of the multi-location bug Chart-5, a Defects4J bug. The advantage of IR1 is that it is the same representation LLMs observe during pre-training. When using this representation, the main limitation is that the model has no access to fault localization information and, thus, needs to determine where to change the code, which can be considered as implicit anomaly detection.

IR2: Buggy function w/ FL comments This representation adds two comments signaling the start and end of the buggy chunk of code. For example, in Figure 2 (IR2), the three lines between the start and end of the suspicious region are surrounded by comments signaling the beginning and end of the buggy chunk. By providing fault localization information, the model can scope its changes to the buggy section.

IR3: Buggy function w/ infilling mask This representation uses the infilling scheme some LLMs are trained for during pre-training [3]. The buggy chunk is replaced by the infilling token, which prompts the model to fill it. For example, in Figure 2 (IR3), the three lines between the start and end of the suspicious region are replaced by the *<FILL_ME>* token. This representation yields shorter inputs and requires less fine-tuning since the infilling objective has been used during pre-training. However, by masking the buggy portion of code, this representation incurs information loss that can be useful to generate a fix.

IR4: Buggy function w/ infilling mask and buggy code This representation combines the buggy code with the infilling scheme. The buggy code is shown in a comment at the end of the prefix portion. For example, in Figure 2 (IR4), the buggy lines are kept in comments, and the <FILL_ME> token is placed immediately afterward. This representation is different from the one learned during pre-training and requires fine-tuning. Code found in the wild would typically not include buggy code as comments, which is considered bad practice. Yet, with fine-tuning, this representation might add valuable information to the infilling scheme.

2.4.3 Output Representation Space. Output representations in APR correspond to the representation of the synthesized fixed code. A natural output representation is a diff over the buggy code, aka a patch. As discussed in subsubsection 2.4.2, fine-tuning is required to adapt an LLM to generate such task-specific outputs. To study the output representation space, we design four output representations tailored to APR (Figure 3):

OR1: Fixed function The naive output is the full fixed function. It is not a diff. Figure 3 (OR1) shows the fixed function of the multi-location bug Chart-5. The major drawback of OR1 is that such output may be much larger than the actual code changes for fixing, and LLMs are known to be more effective at generating short sequences over long sequences.

OR2: Fixed chunk In this representation, the output is composed of the fixed chunk of code to replace the buggy chunk of code. The advantage is that the fixed chunk is typically shorter than the full function, i.e. shorter than OR1. For example, in Figure 3 (OR2), only 6 fixed lines are outputted. OR2 requires an input representation that includes fault localization (i.e. IR2, IR3, IR4) since the output contains no information regarding what to replace.

OR3: Three-line context-diff The output is a typical contextual diff with a three-line context, aka a unified diff. For example, in Figure 3 (OR3), a unified diff of the statement change is outputted. The main challenge of this representation is that the model needs to learn to locate the bug locations during fine-tuning, which is

Input Representations

IR1

```
public XYDataItem addOrUpdate(Number x, Number y) {
 if (x == null) {
  throw new IllegalArgumentException("Null 'x' argument.");
 }
 XYDataItem overwritten = null;
 int index = indexOf(x);
 if (index >= 0 && !this.allowDuplicateXValues) {
  ...
 }
 fireSeriesChanged();
 return overwritten;
}
```

IR2

```
public XYDataItem addOrUpdate(Number x, Number y) {
 if (x == null) {
  throw new IllegalArgumentException("Null 'x' argument.");
 }
 // buggy code starts:
 XYDataItem overwritten = null;
 int index = indexOf(x);
 if (index >= 0 && !this.allowDuplicateXValues) {
 // buggy code ends
  ...
 }
 fireSeriesChanged();
 return overwritten;
}
```

IR3

```
public XYDataItem addOrUpdate(Number x, Number y) {
 if (x == null) {
  throw new IllegalArgumentException("Null 'x' argument.");
 }
 <FILL_ME>
  ...
 }
 fireSeriesChanged();
 return overwritten;
}
```

IR4

```
public XYDataItem addOrUpdate(Number x, Number y) {
 if (x == null) {
  throw new IllegalArgumentException("Null 'x' argument.");
 }
 // buggy code
 // XYDataItem overwritten = null;
 // int index = indexOf(x);
 // if (index >= 0 && !this.allowDuplicateXValues) {
 <FILL_ME>
  ...
 }
 fireSeriesChanged();
 return overwritten;
}
```

Figure 2: Buggy code of the multi-location bug Chart-5 represented in our four different input representations.

Output Representations

OR1

```
public XYDataItem addOrUpdate(Number x, Number y) {
 if (x == null) {
  throw new IllegalArgumentException("Null 'x' argument.");
 }
 if (this.allowDuplicateXValues) {
  add(x, y);
  return null;
 }
 XYDataItem overwritten = null;
 int index = indexOf(x);
 if (index >= 0) {
  ...
 }
 fireSeriesChanged();
 return overwritten;
}
```

OR2

```
 if (this.allowDuplicateXValues) {
  add(x, y);
  return null;
 }
 XYDataItem overwritten = null;
 int index = indexOf(x);
 if (index >= 0) {
```

OR3

```
 public XYDataItem addOrUpdate(Number x, Number y) {
  if (x == null) {
   throw new IllegalArgumentException("Null 'x' argument.");
+ }
+ if (this.allowDuplicateXValues) {
+  add(x, y);
+  return null;
  }
  XYDataItem overwritten = null;
  int index = indexOf(x);
- if (index >= 0 && !this.allowDuplicateXValues) {
+ if (index >= 0) {
   XYDataItem existing = (XYDataItem) this.data.get(index);
   try {
    overwritten = (XYDataItem) existing.clone();
```

OR4

```
   throw new IllegalArgumentException("Null 'x' argument.");
+ }
+ if (this.allowDuplicateXValues) {
+  add(x, y);
+  return null;
  }
@@
  int index = indexOf(x);
- if (index >= 0 && !this.allowDuplicateXValues) {
+ if (index >= 0) {
   XYDataItem existing = (XYDataItem) this.data.get(index);
```

Figure 3: Patch for multi-location bug Chart-5 represented in our four different output representations.

difficult. Additionally, this representation is also lengthier than generating a fixed chunk (OR2) only.

OR4: One-line context-diff The output is a contextual diff with a shorter, one-line context. OR4 uses a one-line diff context, making it shorter than OR3. For example, in Figure 3 (OR4), there are five source code lines less when compared with OR3. Despite this, it is still lengthier than OR2 and also requires the model to learn where to apply the patch.

Table 1: Possible code representation pairs for fine-tuning LLMs for automated program repair. They exploit the characteristics of the APR task, incl. the presence of fault localization signals and the notion of "buggy code".

Code Representations	FL	Aligned w/ PT	Buggy Code
IR1 x OR1	✗	✔/ ✗	✔
IR1 x OR3	✗	✔/ ✗	✔
IR1 x OR4	✗	✔/ ✗	✔
IR2 x OR2	✔	✗/ ✔	✔
IR3 x OR2	✔	✔/ ✔	✗
IR4 x OR2	✔	✗/ ✔	✔

2.4.4 Input/Output Representation Pairs. To utilize an LLM for APR, input and output representations must be carefully paired. This is because all input representations cannot be paired with all output representations. For instance, IR1 cannot pair with OR2 since one cannot apply a fixed chunk to the buggy function without the fault localization information. Table 1 provides the list of the code representation pairs that are studied in this paper. Each row corresponds to a code representation pair. Column *FL* indicates whether the pair includes or not fault localization information. Column *Aligned w/ PT* provides a relative assessment of the alignment of the representation w.r.t. the pre-training data/objective. A red cross means that the code representation is not aligned with the pre-training data and objective. The left side shows the input and the right the output representations. Column *Buggy Code* indicates whether the pair includes or not the original buggy code.

The first three rows (i.e., IR1xOR1, IR1xOR3, IR1xOR4) include code representation pairs that do not contain fault localization signals. The input is the same across all pairs (IR1), whereas the output can either be the full fixed function (OR1) or a diff (OR3, OR4). The key difference between the pairs is the output length and format.

The latter three rows (i.e., IR2xOR2, IR3xOR2, IR4xOR2) include code representation pairs that contain fault localization information, either as tokens or as infilling, which is specific to program repair. The most aligned representation with pre-training is IR3xOR2 since the pre-trained model has support for infilling. IR2 represents the infilling objective with never-before-seen comments, whereas IR4 keeps the buggy code as comments. The natural output representation to pair with these is OR2 since it only includes the new code to replace the already localized buggy chunk, minimizing output length. Note that we have empirically tested other combinations in a pilot experiment, and the ones not listed in Table 1 underperform.

2.5 Choice of Fine-Tuning Dataset

After choosing an initial model and appropriate code representations, the next step is to curate a fine-tuning dataset. First, the dataset must be relevant to the task at hand. In the APR task, a relevant dataset usually includes pairs of buggy and fixed code samples. Second, the type of samples included should be similar to the target bugs. Third, the size of the dataset should be considered. A larger dataset generally leads to better model performance as it provides more examples for the model to fine-tune from. However, it is important to balance size with quality - a smaller, high-quality dataset may be more beneficial than a larger, low-quality one. Fourth, the diversity of the dataset is important. A diverse dataset that covers a wide range of examples can help the model generalize better to unseen data. Lastly, the legality and ethics of the dataset should be considered, in particular regarding privacy and copyright.

2.6 Fine-Tuning Repair Adapters with LoRA

With the recent release of various LLMs, the scale of parameters has significantly increased. For instance, state-of-the-art models such as LLaMA [24] and CodeLLaMA [32] range from 7B to 70B parameters. Fine-tuning these LLMs often requires substantial GPU resources. As an example, Lv et al. [25] report that fine-tuning the full parameters of LLaMA-7B on an RTX 3090 consumes 126.08 GB at peak GPU memory usage, with the batch size and sequence length set to 1 and 1024 respectively. Fine-tuning current LLMs with limited resources is a challenge.

RepairLLaMA uses LoRA [14], a state-of-the-art parameter-efficient fine-tuning method. LoRA freezes the LLM and fine-tunes low-rank matrices in each layer of the language model that are trainable. The trained matrices compose an "adapter", which is many orders of magnitude smaller than the language model itself. In RepairLLaMA, the *repair adapter* is a LoRA adapter, dedicated to the program repair task.

2.7 Inference

The final step is to deploy the repair adapter. The target buggy program is fed to a fault localization algorithm and processed to generate an APR-specific code representation. Then, the code representation is fed to the initial model combined with the LoRA repair adapter, to generate a list of candidate patches for the buggy program at hand. Patches are then checked for plausibility and correctness per off-the-shelf techniques.

3 EXPERIMENTAL METHODOLOGY

3.1 Research Questions

In this work, we focus on the following research questions:

- **RQ1 (Code Representations for Fine-Tuning)**: What is the best code representation to fine-tune an LLM for program repair?
- **RQ2 (Parameter-Efficient Fine-Tuning vs. Full Fine-Tuning)**: How does parameter-efficient fine-tuning compare against full-parameter fine-tuning for program repair?
- **RQ3 (RepairLLaMA vs. ChatGPT-based APR)** How does RepairLLaMA compare against state-of-the-art ChatGPT-based program repair?

3.2 Implementation

Model to Fine-Tune Per the criteria of subsection 2.3, we choose CodeLlama-7b [32] as our initial LLM. CodeLLaMA is a publicly available LLM released in 2023 and is trained on 500B code tokens. Per the experiments reported in [32], CodeLLaMA outperforms GPT-3.5 on two code generation benchmarks.

Fine-tuning Dataset We choose Megadiff [27] as the fine-tuning dataset, and process all samples into the different code representations. First, the function pairs – each comprising a buggy version and its fixed counterpart – are extracted along with their corresponding diff identifiers. Subsequently, we eliminate pairs that do not change single functions, and remove duplicate pairs through textual comparison. After that, we compute our custom code representations. We keep only samples whose total length (input plus output) is shorter than 1024 tokens measured by the LLM tokenizer. Consequently, the fine-tuning datasets range from 30,000 to 50,000 fine-tuning pairs (see our appendix repository).

Evaluation Benchmark We select two Java benchmarks for our evaluation: Defects4J [22] and HumanEval-Java [17]. Following recent related work [19, 43, 45], we scope our evaluation to single-function bugs. Defects4J comprises 835 real-world bugs from 17 open-source Java projects, from which we identify 488 single-function bugs. HumanEval-Java is a bug benchmark containing artificial bugs inserted in HumanEval [4] Java programs. Contrary to Defects4J, HumanEval-Java suffers from less data leakage in the pre-training data since it is much more recent than Defects4J. HumanEval-Java contains 164 single-function bugs.

Fine-Tuning We fine-tune CodeLLaMA with LoRA for each of our curated code representations with the same hyper-parameter settings: we set the learning rate to 5e-4 with cosine decay, max input length to 1024, training epoch to 2, and batch size to 16 per GPU, and we use Adam_W as the optimizer. For LoRA, we use a rank of 8, alpha of 16, dropout of 0.05, and inject the adaptation matrices in the *q_proj* and *v_proj* layers. Using the same hyper-parameter settings for each code representation ensures fair comparison. Each fine-tuning run is executed on a server with 4xA100 40GB GPUs.

Inference In inference, we employ beam search as our decoding strategy with a beam size of 10 per previous research [17]. Hence, for each bug, we generate 10 candidate patches. We use the HuggingFace transformers library to implement all fine-tuning and inference experiments. Inference is run on a single A100 40GB GPU.

3.3 Patch Assessment

Following related work [17, 43, 48], we compute the following repair effectiveness metrics. A *plausible patch* is defined as one that successfully passes all test cases. An *exact match patch* is textually identical to a developer-provided reference patch. To refine our evaluation, we further assess the syntactical equivalence between the generated patches and the reference patches, through *AST match*. This enables us to compute performance regardless of formatting and indentation changes. This process involves converting plausible and reference patches into abstract syntax trees [31] and subsequently utilizing AST differencing [11] to compare their ASTs for discrepancies. A plausible patch with no AST differences compared to the reference patch is classified as an AST match patch. It is also more scalable and accurate than manually checking plausible patches for correctness without expertise in the programs under repair. For all three metrics, the higher the metric, the better the performance. We validate the candidate patches on a workstation with an 18-core Intel Core i9-10980XE CPU and 128 GB of RAM, operating under Ubuntu 22.04.3 LTS.

3.4 Methodology for RQ1

The objective of RQ1 is to investigate the most effective code representations for fine-tuning an LLM for program repair. While existing research has delved into the utility of LLMs for program repair, the impact of the code representations, such as their authenticity, has been overlooked. Additionally, it is crucial to note that variations in code representations may yield substantial differences in performance for fine-tuned LLMs [15]. Consequently, in RQ1, we curate 6 realistic code representations motivated in 2.4 and execute a series of experiments to measure their performance. We fine-tune an LLM as described in 3.2. We prompt the model to generate 10 patches for each bug using beam search decoding. We then evaluate the generated patches as outlined in subsection 3.3, to measure the effectiveness of each code representation.

Baseline. Since CodeLLaMA performs powerful zero-shot learning ability in other code-related tasks [32], we use the same method to prompt CodeLLaMA-7B with the infilling scheme (IR3xOR2) on our selected benchmark and take the result as the baseline, which can help us better evaluate the performance of every code representation and measure the effectiveness of fine-tuning.

3.5 Methodology for RQ2

The objective of RQ2 is to evaluate the respective effectiveness of parameter-efficient and full-parameter fine-tuning. Generally, parameter-efficient fine-tuning methods represent a trade-off between computational cost and model performance, which allows us to train LLMs with limited computational resources. While traditional full-parameter fine-tuning approaches often yield better results, they come at the expense of significantly higher memory requirements and a large-scale fine-tuning dataset. In other words, fully fine-tuning an LLM on a small fine-tuning dataset may drop into the overfitting problem. We explore and compare the effectiveness of parameter-efficient and full-parameter fine-tuning in the specific context of program repair, which has never been done to the best of our knowledge.

Baseline. We consider four baselines in RQ2. One is the baseline considered in RQ1. The second one comprises the full-parameter fine-tuned version of RepairLLaMA, a powerful but naive approach to fine-tuning a large language model for program repair. Here, we use the same hyper-parameters as in LoRA fine-tuning, apart from a lower learning rate of 2e-5. The third is Jiang et al.'s work [17], where several LLMs are fully fine-tuned for program repair. We compare against the best performing model reported, the fine-tuned version of Incoder-6B [12]. The last one is RAP-Gen [39], the current state-of-the-art fine-tuned LLM for program repair. Its key novelty is to use relevant fix patterns retrieved from a codebase of previous bug-fix pairs to augment the buggy input, which guides the model to generate accurate patches by learning from historical repair examples.

3.6 Methodology for RQ3

```
There's a bug in the Java program below. Try to fix it and return the
complete fix for the code in the form of the markdown code block.
Generate the code to replace the <FILL_ME> token.

[Buggy function represented with IR4xOR2]
```

Figure 4: The prompt used to prompt GPT-3.5 and GPT-4 as a strong baseline to generate patches.

The objective of RQ3 is to study how RepairLLaMA compares against state-of-the-art ChatGPT-based program repair. Recently, related work [45, 51] has shown that GPT-3.5-Turbo and GPT-4 achieve state-of-the-art results on program repair. To this end, we compare RepairLLaMA in two experiments.

First, we zero-shot prompt *gpt-3.5-turbo-0613* and *gpt-4-0613* to generate 10 patches for each bug. The prompt is shown in Figure 4, which is built by integrating an effective prompt from Zhang et al.'s work and our curated best code representation. It instructs LLMs to generate the fixed code chunk to replace the <FILL_ME> token.

Second, we compare RepairLLaMA with ChatRepair on Defects4J v1.2, since ChatRepair's authors mainly evaluate it on single-function bugs in Defects4 v1.2. To the best of our knowledge, ChatRepair achieves state-of-the-art performance on Defects4J, and such a comparison enables us to fully evaluate the effectiveness of RepairLLaMA. We utilize OpenAI's official APIs to call *gpt-3.5-turbo-0613* to conduct related experiments on Dec. 1, 2023.

4 EXPERIMENTAL RESULTS

4.1 Results of RQ1 (Code Representations for Fine-Tuning)

In RQ1, we investigate the most effective code representations for fine-tuning an LLM for program repair. The results of the evaluation are presented in Table 2, which shows the effectiveness of each code representation setting on both test benchmarks. The table is structured as follows: the first column displays the code representations used to fine-tune CodeLLaMA-7B, the second and

Table 2: Repair results of different code representations for fine-tuning an LLM for program repair. Our best model, RepairLLaMA using IR4xOR2, significantly improves over the baseline in both test benchmarks.

Code Representations	Defects4J v2 (488 bugs)			HumanEval-Java (164 bugs)		
	Plausible	AST Match	Exact Match	Plausible	AST Match	Exact Match
IR3 x OR2 (baseline, no fine-tuning)	133	71	52	107	81	71
IR1 x OR1	79	31	29	78	54	52
IR1 x OR3	41	17	15	39	21	21
IR1 x OR4	12	2	2	5	2	2
IR2 x OR2	**198**	122	121	**118**	77	69
IR3 x OR2	154	87	84	103	68	63
IR4 x OR2 (RepairLLaMA)	195	**125**	**124**	**118**	**82**	**75**

third meta-columns show the repair effectiveness results on, respectively, Defects4J v2 (single-function) and HumanEval-Java (single-function). Recall that the repair effectiveness evaluation is measured by three metrics described in subsection 3.3.

Our baseline uses the original non-fine-tuned CodeLLaMA-7B with the IR3xOR2 code representation. Recall that this is the most effective way to prompt the non-fine-tuned model as discussed in subsection 3.4. Our results show that it plausibly repairs 133 Defects4J v2 and 107 HumanEval-Java bugs. Moreover, it correctly repairs 52 Defects4J v2 and 71 HumanEval-Java bugs with patches that textually exactly match the developer-written ones. Furthermore, when considering AST match, it can repair 71 Defects4J v2 and 81 HumanEval-Java bugs with patches that are syntactically equal to the developer-written ones.

The fine-tuned model's effectiveness depends on the code representations. First, we observe that the three code representation pairs that do not have access to fault localization (IR1xORX) perform considerably worse than both the baseline and other code representations. These results show that fault localization signals are crucial for program repair. To that extent, all representations that simply use the full function as the input and ask to "fix the bug", can be considered too naive. Our work demonstrates that tailoring code representations with fault localization is a necessary step in the context of program repair. No pre-training objective has access to fault-localization signals, which further validates the need for fine-tuning.

Second, we observe that, within code representations that use fault localization signals, fine-tuned models significantly outperform the baseline on Defects4J v2 compared to the baseline (first row). For the same code representation as the baseline (first row), the fine-tuned model (sixth row, IR3xOR2) can plausibly repair 154 (+21) bugs, exactly repair 84 bugs (+32), and syntactically correctly repair 87 bugs (+16), respectively.

When considering the remaining two code representations (IR2xOR2, IR4xOR2), the fine-tuned models perform even better. The best model, RepairLLaMA, fine-tuned with the code representation IR4xOR2, plausibly repairs 195 (+62) Defects4J bugs than the baseline, exactly repairs 124 (+72), and syntactically correctly repair 125 bugs (+54), respectively. These results show that fine-tuning the initial LLM with repair-specific representations significantly improves the repair effectiveness over a non-fine-tuned model. Tailored code representations with bug localization signals allow the fine-tuned model to repair more bugs because these signals enable it to pay more attention to the buggy code.

To strengthen the external validity of our analysis beyond Defects4J, we perform the same experiment on HumanEval-Java. On HumanEval-Java, RepairLLaMA also achieves better repair effectiveness than the baseline (75 vs 71 exact match and 82 vs 81 ast match), confirming the results observed on Defects4J.

Third, we discuss the alignment between the repair-specific code representation and the pre-training objectives. IR2 and IR4 are input representations that include the same signals: the original function, fault localization, and the original buggy code. However, we see a performance gap between both representations. IR4 uses the same infilling scheme during pre-training while providing bug location signals by including the original buggy code in comments. IR2, on the other hand, mimics a similar scheme but provides bug location signals through fault localization comments, which is further away from the pre-training distribution. This difference explains the differences between IR4xOR2 vs. IR2xOR2: the former has a lower representation gap with pre-training, thus leading to better repair effectiveness. This finding holds for both benchmarks (Defects4J and HumanEval-Java) and for both metrics (Exact Match and AST Match).

In addition to comparing performance, we explore the effectiveness of our single-chunk prompting approach in addressing multi-chunk bugs. Our findings reveal that RepairLLaMA correctly repairs 30 instances of such bugs. For example, RepairLLaMA correctly fixes a complex multi-chunk bug, Math-86, from the Defects4J v2, as illustrated in Figure 5. Math-86 presents two error sections that require simultaneous attention and correction: 1) the removal of an if block that throws an exception, and 2) the introduction of a new if condition. Note that these two sections have more than 20 lines of distance between each other, showing that RepairLLaMA can fix bugs where the multiple edit locations are far away from each other. To the best of our knowledge, RepairLLaMA is the first program repair approach to correctly fix Math-86.

```
@@ -111,9 +111,6 @@
  final double[] lI = lTData[i];
- if (lTData[i][i] < absolutePositivityThreshold) {
-  throw new NotPositiveDefiniteMatrixException();
- }
  for (int j = i + 1; j < order; ++j) {
   final double[] lJ = lTData[j];
@@ -134,6 +131,9 @@
  final double[] ltI = lTData[i];
+ if (ltI[i] < absolutePositivityThreshold) {
+  throw new NotPositiveDefiniteMatrixException();
+ }
  ltI[i] = Math.sqrt(ltI[i]);
  final double inverse = 1.0 / ltI[i];
```

Figure 5: Exact match patch generated by RepairLLaMA for Math-86 from Defects4J v2. In this multi-location bug, RepairLLaMA is able to fix two distant buggy locations.

```
@@ -6,16 +6,18 @@
    int CAP = 0;
    int SM = 0;
    for (char c : s.toCharArray()) {
-    if (Character.toUpperCase(c) == c) SM += 1;
-    if (Character.toLowerCase(c) == c) CAP += 1;
+    if (Character.toUpperCase(c) == c) CAP += 1;
+    if (Character.toLowerCase(c) == c) SM += 1;
    }
-   val = CAP - SM;
-   strong = s;
+   if (CAP - SM > val) {
+    val = CAP - SM;
+    strong = s;
+   }
   }
  return class_name + "." + strong;
 }
```

Figure 6: Exact match patch generated by RepairLLaMA for *STRONGEST_EXTENSION* from HumanEval-Java. In this multi-location bug, RepairLLaMA modifies two if blocks and encapsulates two other statements in a new if block.

Figure 6 shows another example of an exact match multi-location patch generated by RepairLLaMA, which is for a HumanEval-Java bug *STRONGEST_EXTENSION*. RepairLLaMA first swaps the statements conditioned by the two existing if conditions, understanding that the wrong counters are being incremented in each case. Then, RepairLLaMA conditions the two statements updating the current values of *val* and *strong* only if the difference between the counters is greater than the already existing solution. Both this example and Math-86 show the effectiveness of RepairLLaMA in repairing multi-location bugs, thanks to tailored code representations that represent fault localization information in a realistic manner.

Answer to RQ1: Our results demonstrate the importance of designing code representations for fine-tuning LLMs for APR. Naive representations such as full functions are suboptimal, whether on the input or the output side of the model. Our experiments show that the best code representation pair is IR4xOR2, because it leverages two signals specific to the program repair task at hand (fault localization and the original buggy code) while maintaining alignment with the pre-training objective of the initial model. The model RepairLLaMA, fine-tuned with IR4xOR2, correctly repairs 125 Defects4J bugs and 82 HumanEval-Java bugs. This significant improvement in program repair capability demonstrates the need for curated code representations in automated program repair. While the community focuses a lot on prompt engineering, our original experimental results encourage research on domain-specific, expert code representations per downstream task in SE.

4.2 Results of RQ2 (Parameter-Efficient Fine-Tuning vs. Full Fine-Tuning)

Table 3: Repair effectiveness of RepairLLaMA compared with fully fine-tuned models. RepairLLaMA, trained with parameter-efficient fine-tuning, outperforms all comparative models on both Defects4J and HumanEval-Java.

Model	Defects4J v2 (488 bugs)			HumanEval-Java (164 bugs)		
	Plausible	AST Match	Exact Match	Plausible	AST Match	Exact Match
IR3 x OR2 (no fine-tuning)	133	71	52	107	81	71
IR4 x OR2 (full fine-tuning)	141	89	77	105	80	71
IR4 x OR2 (RepairLLaMA)	**195**	**125**	**124**	**118**	**82**	**75**

In RQ2, we study how parameter-efficient fine-tuning compares against basic full-parameter fine-tuning. Recall that in most of the closely related work in APR [17, 18, 35, 39, 42, 49], full-parameter fine-tuning is the standard paradigm. On the contrary, in RepairLLaMA, all models are fine-tuned using LoRA, a parameter-efficient fine-tuning technique that optimizes only a small adapter (approx. 4M parameters) instead of the whole LLM (approx. 7B parameters, a reduction of approx. 1600x). This allows the model to 1) be fine-tuned with less GPU memory, and 2) potentially reduce overfitting. In RQ2, we compare RepairLLaMA, built with the best code representation in RQ1, with its full-parameter fine-tuning version.

Table 3 presents the results of RQ2. The table reads as follows. The first column presents the model. The second and third meta-columns show the repair effectiveness results on, respectively, Defects4J v2 (single-function) and HumanEval-Java.

The results show that RepairLLaMA with parameter-efficient fine-tuning clearly outperforms both the baseline and its full-parameter fine-tuning version. In Defects4J, RepairLLaMA plausibly repairs 54 bugs more and exactly repairs 47 more than the fully fine-tuned model. When considering AST Match, RepairLLaMA also repairs 36 bugs more. In HumanEval-Java, although the improvement is smaller, RepairLLaMA still outperforms all baselines.

The gain in performance is clear, and it also requires fewer resources because it only needs to optimize a much smaller adapter (i.e., approx. 4M parameters). Possibly, the fully fine-tuned model may drop into overfitting due to the limited fine-tuning data, and parameter-efficient fine-tuning helps prevent overfitting since it only requires optimizing a small part of network weights. The model size constraints of LoRA appear to act as implicit regularizers.

When compared with the best fully fine-tuned model from Jiang et al. [17], RepairLLaMA correctly repairs 56 Defects4J bugs more and 12 HumanEval-Java bugs more. These results further validate RepairLLaMA's design choices, including the use of parameter-efficient fine-tuning. Even when fully fine-tuning a similar model, Incoder-6B in this case, RepairLLaMA still outperforms it.

To further evaluate RepairLLaMA, we also compare it with RAP-Gen [39], the state-of-the-art program repair approach that fully fine-tunes an LLM. RAP-Gen correctly repairs 125 Defects4J v2 bugs, according to the authors' manual verification, when generating 100 patches generation per bug. RepairLLaMA also correctly repairs the same number of Defects4J v2 bugs under the metric of AST match. However, it achieves this feat by generating 10x fewer patches for each bug, demonstrating the effectiveness of our fine-tuning and code representation approach. Lastly, for multi-location bugs, RAP-Gen assumes perfect multi-location fault localization, whereas RepairLLaMA assumes a more realistic setting where only the first and last buggy lines are identified, as discussed in subsubsection 2.4.1.

> **Answer to RQ2:** Parameter-efficient fine-tuning outperforms full fine-tuning in both Defects4J and HumanEval-Java. The fair experimental comparison yields a clear-cut result: performance is higher with an additional 36 correctly repaired bugs in Defects4J. Overall, our paper is the first to demonstrate the advantage of parameter-efficient fine-tuning in the context of automated program repair, achieving strong state-of-the-art results. Beyond program repair, parameter-efficient fine-tuning is feasible in academic labs for other software engineering tasks, while still working with powerful multi-million dollar trained models.

4.3 Results of RQ3 (RepairLLaMA vs. ChatGPT-based APR)

In RQ3, we compare RepairLLaMA with state-of-the-art ChatGPT-based program repair. ChatGPT-based program repair differs from RepairLLaMA since it does not involve fine-tuning LLMs on task-specific datasets. Instead, it involves designing effective prompt strategies to instruct a powerful general-purpose LLM like GPT-4.

Table 4 shows the repair effectiveness of RepairLLaMA compared with ChatGPT-based APR techniques. The first column indicates the model name. The second, and third show the results on two different benchmarks. Each benchmark is evaluated following the three patch assessment metrics described in subsection 3.3.

Our results show that RepairLLaMA is the most effective model across both benchmarks. GPT-3.5 and GPT-4 plausibly fix 63% and 38% less Defects4J v2 bugs than RepairLLaMA, respectively. RepairLLaMA fixes twice as many bugs as GPT4, with both the AST match and the exact match metrics. RepairLLaMA also correctly repairs the most HumanEval-Java bugs when compared with ChatGPT-based techniques, which increases both internal and external validity. Recall that both GPT-3.5 and GPT-4 are significantly larger than the RepairLLaMA model.

Finally, we compare against an even more sophisticated usage of ChatGPT, iterative prompting, per the results of Xia et al. [45]. Based on the patches shared by the authors via private communication, RepairLLaMA correctly fixes, according to the AST match metric, more Defects4J v2 bugs. Moreover, ChatRepair includes more information in its prompts than RepairLLaMA does in its input representation (IR4), including, for example, test execution feedback: despite fewer repair signals, RepairLLaMA correctly fixes more bugs, demonstrating the superiority of fine-tuning over iterative prompting.

To conclude, these results demonstrate the power of specializing in an LLM for APR. A smaller model, trained with a parameter-efficient fine-tuning technique, is more effective than a large general-purpose LLM, either instructed with task-specific prompts or even used in an advanced iterative manner. Overall, RepairLLaMA beats the strong baseline of GPT-4 on all the considered benchmarks.

> **Answer to RQ3:** RepairLLaMA beats state-of-the-art iterative prompting and even beats GPT-4, thanks to the combination of appropriate code representations and parameter-efficient fine-tuning. Our experiments demonstrate the power of task specialization, with task specific engineering and the training of a neural network adapter that distills capabilities on the task at hand. The RepairLLaMA program repair adapter is more powerful than ChatGPT for fixing bugs in Java.

5 DISCUSSION

5.1 Sampling Candidate Patches

One key aspect of any program repair approach is the number of generated candidate patches. Some recent works generate hundreds and even thousands of patches for a single bug [39, 40, 44, 45]. However, the cost of evaluating such a large number of candidate patches has been largely overlooked. Recall that to evaluate the plausibility of each candidate patch, one must run the test cases, which is expensive and even overcomes the one-time cost of fine-tuning. In contrast to this trend, RepairLLaMA achieves state-of-the-art results while generating only 10 candidate patches per bug. This shows that 1) RepairLLaMA natively priorities the best patches in the top-10 list, 2) RepairLLaMA minimizes the resources that are required in an end-to-end repair pipeline that includes plausibility checking.

5.2 Threats to Validity

The primary internal threat lies in the potential data leakage during the pre-training phase of LLMs. LLMs are pre-trained on vast corpora scrapped from the web, potentially including the same data used for testing, endangering the reliability of experimental results. To mitigate this threat, we assess all models on a recent benchmark specifically designed to address the data leakage issue, HumanEval-Java [17].

Table 4: RQ3: RepairLLaMA's effectiveness compared with state-of-the-art ChatGPT-based APR techniques. RepairLLaMA is more effective in finding correct and plausible patches in both test benchmarks, incl. against the strong baseline of GPT-4.

Model	Defects4J v2 (488 bugs)			HumanEval-Java (164 bugs)		
	Plausible	AST Match	Exact Match	Plausible	AST Match	Exact Match
GPT-3.5	73	34	24	50	62	107
GPT-4	121	61	48	**124**	74	64
RepairLLaMA (IR4 xOR2)	**195**	**125**	**124**	118	**82**	**75**

Another internal threat pertains to data leakage during the fine-tuning process of LLMs, since both our fine-tuning dataset, Megadiff, and Defects4J, contain samples from GitHub. To address this threat, we meticulously compare the samples in our fine-tuning dataset, Megadiff, with those in Defects4J. We found no identical samples shared by both datasets. However, it is worth noting that there are three samples (Math-28, Math-44, and JacksonDatabind-82) whose patch includes a function also found in Megadiff samples. To mitigate this threat, we exclude these three Defects4J samples from the evaluation of our fine-tuned models.

The main external threat is the focus on a single programming language, as our results may not generalize to other languages. To mitigate this threat, we evaluate on two benchmarks, including well-established Defects4J [22]. Our core novelties are programming language agnostic and should generalize to arbitrary languages.

6 RELATED WORK

6.1 Large Language Models for Program Repair

Fine-Tuning. Several works [17, 18, 20, 30, 35, 39, 42, 49, 50, 53] have proposed fine-tuning large language models for the program repair task. Notably, Jiang et al. [17] specifically study the impact of fine-tuning LLMs for program repair, reporting improvements lower than ours while using naive full-parameter fine-tuning. Huang et al. [15] also study different aspects of fine-tuning LLMs for program repair, including code representations and evaluation metrics. While they report state-of-the-art performance, it is achieved under the unrealistic assumption of perfect multi-line fault localization, which RepairLLaMA does not assume.

Overall, our work distinguishes itself from related work in three key aspects. First, we have designed and evaluated several code representations in RepairLLaMA, tailored to fine-tuning LLMs for program repair, which incorporate fault localization signals under realistic assumptions. This is different from previous work (e.g., [15, 39]) which assumes perfect multi-line fault localization. Second, RepairLLaMA is the first to employ LoRA to fine-tune LLMs for program repair, demonstrating that parameter-efficient fine-tuning can surpass full-parameter fine-tuning while reducing computational requirements. Third, unlike some previous work that generates hundreds, even thousands of patches for each bug, our best model, RepairLLaMA, improves state-of-the-art performance on Defects4J and HumanEval-Java with a budget of just 10 patches per bug, demonstrating the laser-style focus of the trained program repair adapter.

Prompting. Recent related work [2, 10, 16, 40, 43–45, 51] uses LLMs for program repair, without fine-tuning. The core of these works is prompting: they design and evaluate different prompting strategies to provide repair signals to the model, and guide the model to generate good patches.

Our work is fundamentally different from them since RepairLLaMA explicitly optimizes the model weights by parameter-efficient fine-tuning which is not the case of prompt-based research. The RepairLLaMA program adapter completely embodies the program repair knowledge. Also, unlike most prompt-based program repair approaches that generate hundreds of patches for a single bug, our model achieves state-of-the-art performance generating only 10 patches per bug.

6.2 Code Representations for Program Repair

Several code representations for program repair have been proposed by related work [1, 5, 6, 8, 13, 28, 29, 46, 47]. Notably, Namavar et al. [29] investigate the impact of different code representations for program repair for a restricted set of bug classes. Differently, our work targets a larger spectrum of bugs, including multi-location bugs.

Overall, our work distinguishes itself from preceding research in code representation in three dimensions. First, we design code representations that are aligned with the pre-training data and objectives, enabling the RepairLLaMA to well utilize the pre-learned knowledge. Second, our code representations are designed to support a large spectrum of bugs, including multi-location bugs, which is one frontier of program repair. Third, RepairLLaMA's pipeline and evaluation are not constructed under the unrealistic perfect fault localization for multi-location bugs.

6.3 Parameter-Efficient Fine-Tuning in SE

Parameter-efficient fine-tuning is a relatively under-explored area in Software Engineering. Wang et al. [38] explore parameter-efficient fine-tuning techniques for specializing LLMs for code search and code summarization, finding that parameter-efficient fine-tuning outperforms in-context learning. Weyssow et al. [41] confirm the dominance of parameter-efficient fine-tuning techniques over zero-shot learning in code generation, while Wang et al. [37] find that prompt-tuning outperforms traditional fine-tuning methods in code summarization. CodePrompt [9] proposes corpus-specific prompt templates similar to adaptations and boosts code generation performance. Lastly, Shi et al. [34] propose a parameter-efficient fine-tuning technique for code related tasks that selectively freezes layers of the model.

Overall, we are the first to employ and evaluate LoRA to fine-tune LLMs for program repair. RepairLLaMA's effectiveness calls for more work in parameter-efficient fine-tuning in program repair and related tasks such as overfitting detection.

7 CONCLUSION

In this paper, we have proposed RepairLLaMA, a novel program repair approach that combines parameter-efficient fine-tuning with program repair specific code representations. RepairLLaMA's code representations are unique in incorporating repair signals, such as fault localization, under realistic assumptions, and in aligning with pre-training data and objectives. To validate RepairLLaMA, we perform a series of extensive experiments on two benchmarks, incl. Defects4J and HumanEval-Java. Our results clearly validate our core design decisions, with RepairLLaMA correctly fixing 125 Defects4J and 82 HumanEval-Java bugs, outperforming strong baselines, incl. GPT-3.5 and GPT-4. RepairLLaMA opens an avenue for research on different kinds of efficient fine-tuning for program repair.

REFERENCES

[1] Kumar Abhinav, Vijaya Sharvani, Alpana Dubey, Meenakshi D'Souza, Nitish Bhardwaj, Sakshi Jain, and Veenu Arora. 2021. Repairnet: contextual sequence-to-sequence network for automated program repair. In *International Conference on Artificial Intelligence in Education*. Springer, 3–15.

[2] Toufique Ahmed and Premkumar Devanbu. 2023. Majority Rule: better patching via Self-Consistency. *arXiv preprint arXiv:2306.00108* (2023).

[3] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255* (2022).

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

[5] Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. PLUR: A unifying, graph-based view of program learning, understanding, and repair. *Advances in Neural Information Processing Systems* 34 (2021), 23089–23101.

[6] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

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

[8] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019).

[9] YunSeok Choi and Jee-Hyong Lee. 2023. CodePrompt: Task-Agnostic Prefix Tuning for Program and Language Generation. In *Findings of the Association for Computational Linguistics: ACL 2023*. 5282–5297.

[10] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust Compilation Errors using LLMs. *arXiv preprint arXiv:2308.05177* (2023).

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

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

[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[14] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

[15] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 1162–1174.

[16] Qing Huang, Jiahui Zhu, Zhenchang Xing, Huan Jin, Changjing Wang, and Xiwei Xu. 2023. A Chain of AI-based Solutions for Resolving FQNs and Fixing Syntax Errors in Partial Code. *arXiv preprint arXiv:2306.11981* (2023).

[17] N. Jiang, K. Liu, T. Lutellier, and L. Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1430–1442.

[18] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*.

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

[20] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263* (2023).

[21] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[22] Rene Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

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

[24] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning (Practical Experience Report). *arXiv preprint arXiv:2308.11148* (2023).

[25] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. 2023. Full Parameter Fine-tuning for Large Language Models with Limited Resources. *arXiv preprint arXiv:2306.09782* (2023).

[26] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[27] 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.SE]

[28] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124* (2023).

[29] Marjane Namavar, Noor Nashid, and Ali Mesbah. 2022. A controlled experiment of different code representations for learning-based program repair. *Empirical Software Engineering* 27, 7 (2022), 190.

[30] Rishov Paul, Md Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna CS Santos. [n. d.]. Enhancing Automated Program Repair through Fine-tuning and Prompt Engineering. ([n. d.]).

[31] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[32] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[33] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 13–24.

[34] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond. *arXiv preprint arXiv:2304.05216* (2023).

[35] Atsushi Shirafuji, Md Mostafizer Rahman, Md Faizul Ibne Amin, and Yutaka Watanobe. 2023. Program Repair with Minimal Edits Using CodeT5. *arXiv preprint arXiv:2309.14760* (2023).

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

[37] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 382–394.

[38] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter

Tuning for Code Search and Summarization. *arXiv preprint arXiv:2303.15822* (2023).

[39] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[40] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. *arXiv preprint arXiv:2309.00608* (2023).

[41] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *arXiv preprint arXiv:2308.10462* (2023).

[42] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 522–534.

[43] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179* (2022).

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

[45] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[46] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*. PMLR, 10799–10808.

[47] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[48] He Ye and Martin Monperrus. 2023. ITER: Iterative Neural Repair for Multi-Location Patches. *arXiv preprint arXiv:2304.12015* (2023).

[49] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 678–690.

[50] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[51] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair. *arXiv preprint arXiv:2310.08879* (2023).

[52] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. ACM, 341–353.

[53] Armin Zirak and Hadi Hemmati. 2022. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology* (2022).