Significance of Large Language Models for Automated Program Repair
Final Report

Sarthak Thakur
a1871690

April 19, 2024

Report submitted for **MATHS 7097B** at the School of Mathematical Sciences, University of Adelaide

THE UNIVERSITY *of* ADELAIDE

Project Area: **LARGE LANGUAGE MODELS, AUTOMATED PROGRAM REPAIR**
Project Supervisor: **BOYU ZHANG, LIMENG ZHANG, PROF. ALI BABAR**

Abstract

Automated Program Repair (APR) endeavors to bolster software reliability through the generation of patches for flawed/buggy programs. While various Code Language Models (CLMs) have exhibited efficacy in tasks like code completion, their evaluation for repair capabilities, referred to as the ”Fill-in-the-Middle” task in Large Language Model literature, remains underexplored. This study builds upon prior research by investigating newer CLMs while assessing advancements in the field, revealing substantial progress in CLMs' performance, nearly doubling in recent times. The findings advocate promising directions for APR, advocating for the utilization of CLMs with APR-specific designs.

The research further extends its objectives to include fine-tuning these CLMs for the APR task. This involves adapting the models to the specific domain of repairing faulty programs, thus enhancing their effectiveness in generating patches. The work involves comprehensive experimentation with various fine-tuning strategies, aiming to optimize the CLMs' performance specifically for the APR task. Results indicate notable improvements in patch generation accuracy and efficiency, demonstrating the efficacy of fine-tuning CLMs for APR. This expanded investigation highlights the significance of leveraging CLMs not only for zero-shot tasks but also for fine-tuning towards specialized objectives, such as program repair, thereby contributing to the advancement of software reliability and maintenance practices whilst also bridging the gap between academia and industry.

1 Introduction

Automated Program Repair (APR) [1-3] stands as a pivotal tool for developers, aiming to bolster software reliability by automatically generating patches to rectify software defects. Deep Learning (DL)-based APR techniques [4-6] have garnered significant attention, harnessing DL models to process faulty software programs and generate patched versions. These techniques typically construct neural network models using training sets consisting of pairs of faulty and corrected code. Leveraging the robust learning capabilities of DL models, these techniques excel in transforming faulty programs into patched ones, often surpassing traditional APR techniques like template-based, heuristic-based, and constraint-based. [7-9]

Despite the effectiveness of DL-based APR techniques, they encounter challenges in addressing a substantial portion of bugs. Additionally, the time-intensive process of generating and validating hundreds to thousands of candidate patches poses practical limitations [10], especially concerning developers' willingness to review numerous patches promptly [10]. This underscores a practical gap in the utilization of DL-based APR research.

In light of these challenges, code language models (CLMs) have emerged as promising alternatives for bug fixing, leveraging their effectiveness in natural language domains. Unlike DL-based APR models, CLMs are trained on large unlabeled code corpora for general code language modeling tasks, such as next token prediction.

Building upon existing literature [11], which suggests that CLM-based APR techniques surpass DL-based techniques, this study exclusively focuses on CLMs. It aims to extend previous research by exploring recently released models like ReFact [12] and CodeLlama [13]. The assessment of these models is conducted using the HumanEval-Java dataset [11], a translation of the original HumanEval dataset [14] by OpenAI from Python to Java. This translation aims to mitigate concerns related to data leakage, considering the likelihood that CLMs have encountered widely-used datasets during their pre-training phase.

Moreover, this study incorporates fine-tuning of CLMs alongside zero-shotting, aiming to investigate potential improvements over time. The primary research question explores **whether CLMs have evolved and improved in their ability to address software defects, and if so, in what aspects and also if fine-tuning CLMs for domain-specific tasks like APR is beneficial or not**.

2 Background

In this section, we present a succinct overview of large language models, delineating their developmental stages and emergence, alongside an exploration of Automated Program Repair (APR), encompassing both traditional methodologies predating the integration of learning-based models and contemporary advancements. Additionally, we illuminate related works acknowledged by the author during the project's execution, accompanied by descriptions of baseline models employed. It is imperative to acknowledge the inherent limitations in providing exhaustive coverage of large language models, encompassing aspects such as their evolving capabilities, scalability, and training methodologies, as well as the diverse array of approaches within APR. Consequently, our focus within this section is constrained to specific tasks pertinent to our research objectives, namely causal language modeling and masked language modeling, contextualized within the realm of our work. The deliberate curtailment of comprehensive coverage is motivated by the imperative to preserve conciseness and relevance within the confines of this report's scope.

2.1 Large Language Models

Language is a vital skill for humans, developing early in life and evolving over time [15]. However, machines lack this innate ability unless enhanced with advanced AI algorithms. Bridging this gap has long been a challenge, aiming to enable machines to comprehend and communicate like humans [16]. Language modeling (LM) is a key approach in advancing machine language intelligence. LM focuses on predicting the likelihood of word sequences, aiding in forecasting future (casual language modeling) or missing tokens (masked language modeling). LM research has undergone four major developmental stages, garnering significant attention in literature, as succinctly outlined below:

- *Statistical Language Models:* Statistical Language Models (SLMs) [17], originating in the 1990s, rely on statistical learning techniques. They operate using Markov rules, predicting the next token based on previous context. Models having fixed context lengths, known as n-gram language models (e.g., bigram, trigram), are common. Some applications of SLM include information retrieval (IR) and natural language processing (NLP). However, they face challenges like the curse of dimensionality, making it hard to estimate high-order language models accurately which can be mitigated to some extent with techniques like Backoff Estimation [18] and Good-Turing estimation [19].

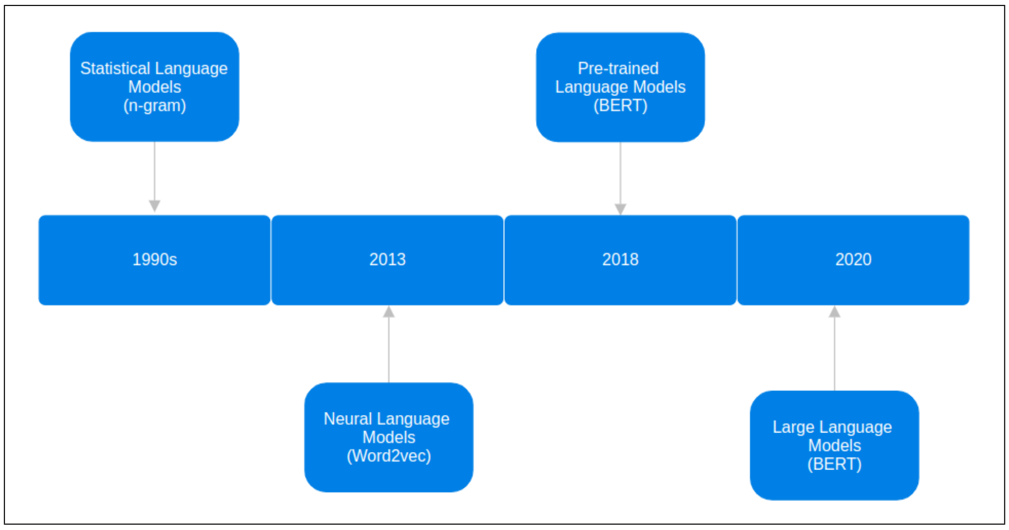
Statistical Language Models (n-gram)
Pre-trained Language Models (BERT)
1990s
2013
2018
2020
Neural Language Models (Word2vec)
Large Language Models (BERT)

Figure 1: Evolution of Language Modeling

- *Neural Language Models:* Neural Language Models (NLMs) [20] employ neural networks like multi-layer perceptrons (MLPs) and recurrent neural networks (RNNs) to characterize word sequence probabilities. They introduced distributed word representation [20], revolutionizing NLP tasks. Word2vec [21-22], a simplified neural network, was proposed for learning effective distributed word representations, proving highly effective across NLP tasks. These studies pioneered the use of language models for representation learning, extending beyond word sequence modeling and significantly impacting NLP.

- *Pre-trained Language Models:* Pre-trained Language Models (PLMs) like ELMo [23] and BERT [24] revolutionized NLP by capturing context-aware word representations. ELMo pre-trains bidirectional LSTM networks and fine-tunes them for specific tasks. BERT employs Transformer [13] architecture with self-attention mechanisms for pre-training bidirectional language models on large-scale corpora, setting a new standard for NLP tasks. This approach inspired numerous follow-up studies, establishing the "pre-training and fine-tuning" paradigm. Subsequent research introduced various architectures (e.g., GPT-2 [25], BART [26]) and improved pre-training strategies. Fine-tuning PLMs is common for adapting to different downstream tasks.

- *Large Language Models:* Large Language Models (LLMs) have demonstrated improved performance on downstream tasks through scaling, adhering to the scaling law [27]. Studies have divulged upon performance limits by training ascendingly larger PLMs, such as GPT-3 [28] and PaLM [29]. These larger models, despite similar architectures and pre-training tasks, exhibit distinct behaviors and emergent abilities , outperforming smaller PLMs like BERT and GPT-2. GPT-3, for instance, excels in few-shot tasks through in-context learning, distinguishing itself from GPT-2. Coined by the research community, LLMs garner growing attention for their remarkable capabilities. Notably, ChatGPT, derived from the GPT series, showcases impressive conversational abilities with humans.

2.2 Language Modeling with LLMs

In this section, we provide a brief overview of the basics of language modeling based on Transformers [13]. For the purpose of this report, our focus will be confined to specific tasks aligned with our interests. These tasks are delineated into causal language modeling and masked language modeling. Comprehensive coverage of all aspects has been intentionally delimited to maintain conciseness and relevance within the scope of this report.

- *Causal Language Modeling:* Unidirectional language models, also known as causal language models or next token prediction models, determine sentence probability through the chain rule, multiplying each token's conditional probability. For an input text x = *[*x_1, x_2, ..., x_n*]* with n tokens, it is modeled as

  $$P(x) = \prod_{i=1}^{n} p_\theta(x_i|x_{1:i-1}) \tag{1}$$

  where $x_{1:i-1}$ is a representation of tokens preceding x_i in the input, and θ represents the model parameters. Transformer decoders, exemplified by models like GPT [28] and LLaMA [30], enhance this probability calculation by introducing attention masks, ensuring tokens can only attend to preceding ones. During training, cross-entropy loss is computed for all tokens, and during inference, new tokens are generated in an autoregressive manner

- *Masked Language Modeling:* In contrast, bidirectional language models, exemplified by BERT [24], seek a nuanced contextual representation by training a Transformer encoder. This involves replacing a randomly selected set of tokens in the input with a special token [MASK] to create a noisy input $\hat{x}$ = *[[CLS],* x_1*, [MASK],*

x_3*, [MASK],* x_5*, [EOS]].* The model is then trained to recover the original tokens by maximizing this objective.

$$P(x) = \prod_{m} p_\theta(m|\hat{x}) \tag{2}$$

However, this approach faces challenges in training efficiency, as only a small subset of tokens (typically 15%) is masked and "trained on".

- *Fill-in-Middle* All categories of models face limitations in infilling tasks, where the objective is to generate text at a specific location within a prompt, considering both a prefix and a suffix. Left-to-right models, such as casual models, can only take the prefix into account. On the other hand, encoder-only and encoder-decoder models, like masked models, have the capability to consider suffixes; however, the lengths of infill regions observed during training are typically much shorter than what proves useful in practical applications. This limitation is particularly unfortunate because infilling is naturally relevant in scenarios where context is present both before and after the point of generation. Examples include applications like docstring generation, import statement generation, completing a partially written function, or, in the context mentioned, the removal of a buggy line followed by fixing the program through completion.

  Hence, [31] introduced a method for code refactoring, enabling the incorporation of the fill-in-the-middle (FIM) technique into causal decoderonly models. This is achieved through a simple modification to the training data, allowing models to learn FIM without compromising their existing left-to-right generation capability. Recent code language models (CLMs) like Codellama and ReFact have implemented this approach.

  The crucial step involves a slight transformation, concatenating the prefix and suffix parts and utilizing them for training purposes, as illustrated in the equations below.

$$\text{<PRE>} + \text{Enc(prefix)} + \text{<SUF>} + \text{Enc(suffix)} + \text{<MID>} + \text{Enc(middle)} \tag{3}$$

  Here, <PRE>, <SUF> and <MID> represent the Prefix, Suffix and Middle special tokens respectively and are used to format the prompt accordingly.

  During inference time, the middle part is removed from the above equation resulting in (4)

$$\text{<PRE>} + \text{Enc(prefix)} + \text{<SUF>} + \text{Enc(suffix)} + \text{<MID>} \tag{4}$$

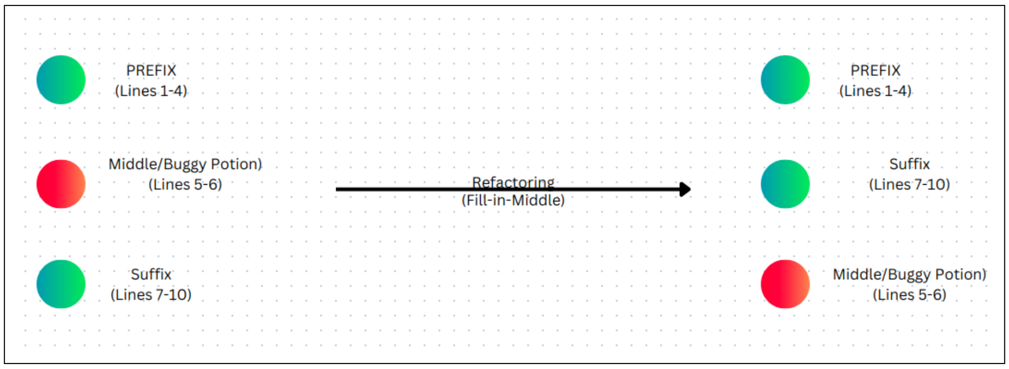

Figure 2: Fill-in-Middle Code Refactoring: Consider a code snippet of 10 lines with a bug at lines 5-6. This portion is referred to as middle. The code is refactored in such a way such that the prefix and suffix are concatenated together whilst pushing the middle at the end

2.3 Automated Program Repair

Automated Program Repair (APR) is a pivotal area of research in software engineering, aimed at automating the identification and rectification of software defects to enhance reliability. Traditional methods involve manual debugging, but APR leverages machine learning and natural language processing to automate this process. Deep learning models and code language models (CLMs) are key techniques in APR. They learn from datasets of faulty and corrected code to propose patches autonomously. Challenges include accuracy, scalability, and data scarcity. APR holds promise for improving software reliability and development efficiency. Continued research is crucial for advancing this field. APR can be subdivided into the following categories which will also serve as our baseline models.

- *Search-based:* Search-based techniques, also referred to as heuristic solutions, involve searching within a predefined patch space to identify the correct patch. These methods typically begin by searching for the most probable locations of buggy code in the program, often employing heuristics to generate potential fixes. Subsequently, they search for suitable fix patches through processes such as mutation-selection, test execution, traversal strategies, and similar approaches. Heuristic methods have been in use since around 2005. GenProg [32], a seminal work in this field, employed genetic programming for synthesizing program variants. More recently, SimFix [33] utilized advanced code search techniques for patch generation.

- *Constraint-based:* Constraint-based approaches, also known as semantic constraint approaches, operate by establishing a series of constraint specifications to guide the repair process. These methods convert the program repair problem into a constraint solver problem and utilize formal constraints to efficiently eliminate unnecessary parts of the search space. This approach aims to identify expression-level variations that fulfill constraints gathered from program analysis and tests, thereby facilitating patch generation. For instance, SemFix [34] utilize test cases as constraints to direct the patch synthesis process.

- *Template-based:* Throughout the history of Automated Program Repair (APR), heuristic search methods have evolved into mature solutions. However, their extensive search spaces significantly impede repair efficiency and incur high costs in fixing defects. Similarly, semantic constraint methods encounter limitations, often struggling to establish suitable constraint specifications for various types of bugs, thus relying on constraints with numerous constraints. To address these challenges, template-based APR techniques have emerged. These methods use predefined program fix templates [35], also known as ”fix patterns” or ”transformation schemas”, to formulate fixed patches. These fix templates can be manually extracted or automatically mined. Subsequently, template-based APR tools target defects to select corresponding fix templates and generate candidate patches. Recent studies [36] have highlighted the effectiveness of template-based repair tools compared to search-based and constraint-based approaches.

2.4 Results from Part A

In Part A of the study, the performance of various models in Automated Program Repair (APR) was evaluated using the pass@10 metric, which measures the number of programs fixed by applying the top 10 generated patches. The results are summarized as follows:

These results provide insights into the effectiveness of different models in addressing software defects through automated patch generation. It is evident that CLMs perform better than traditional LLMs and larger models, such as ReFact, CodeLLaMA (7B), and CodeLLaMA (13B), outperform smaller models like PLBART and CodeT5, indicating the importance of model size in APR tasks. Also, it is observed that models having FIM capibilities perform better than traditional CLMs for bug fixing tasks. These findings

Model	# of programs fixed (pass@10)
PLBART (140M)	39 (23.78%)
CodeT5 (60M)	3 (1.83%)
CodeGen (350M)	30 (18.29%)
InCoder (1B)	40 (24.39%)
ReFact (1.6B)	**70 (42.68%)**
CodeLLaMA (7B)	**91 (55.48%)**
CodeLLaMA (13B)	**102 (62.20%)**

Table 1: Pass@10 for CLM models on HumanEval-Java dataset

serve as a foundation for further investigation and refinement of APR techniques in Part B of the study.

3 Methods

In this section, we will elucidate the methodologies employed to fine-tune our models for the Automated Program Repair (APR) task. While several methods exist for adapting Large Language Models (LLMs) for fine-tuning, such as instruction tuning [37], Supervised Fine Tuning, and Reinforcement Learning from Human Feedback (RLHF) [38], we will selectively focus on those utilized in our work to maintain conciseness and relevance within the report's scope. Additionally, we integrated parameter-efficient and memory-efficient adaptation techniques, including PEFT, LoRA [39], k-bit quantization, among others, into our approach. Rather than delving into exhaustive detail on all methodologies, our discussion will center on those specifically employed in our study. Furthermore, we will provide insights into the dataset utilized for our experiments and any external libraries leveraged to facilitate them. Lastly, we will delineate the evaluation metrics employed on these benchmarks to facilitate comparative analysis of different models.

3.1 LLM Adaptation

Following pre-training, Large Language Models (LLMs) acquire broad capabilities for tackling diverse tasks. However, recent research suggests that these abilities can be further refined to suit specific objectives. Two primary approaches for adapting pre-trained LLMs have emerged: instruction tuning and alignment tuning. Instruction tuning primarily aims to bring forward the existing abilities of LLMs, while alignment tuning seeks to align LLM behaviors with human values or preferences. In this study, our focus is solely on instruction tuning, specifically Supervised Fine-Tuning, which was the method employed in our investigation.

- *Supervised Fine Tuning (SFT)*: A simple method involves optimizing LLMs using the traditional Seq2Seq objective, utilizing the fine-tuning data. As the finetuning data usually comprises an input instruction and an output response (usually paired as a prompt $\rightarrow$ response pair), the main training loss remains the conventional cross-entropy loss for language modeling.

3.2 Parameter-Efficient Model Adaptation

In the preceding section, we elaborated on the methodology of supervised fine-tuning to adapt Large Language Models (LLMs) according to specific objectives. Given the considerable magnitude of parameters within LLMs, conducting full parameter tuning can be prohibitively resource-intensive. Therefore, this section will delve into the strategies for conducting efficient tuning on LLMs. While numerous techniques exist for this purpose, including adapter tuning, prompt tuning, and LoRA, among others, our focus will be solely on LoRA and QLoRA [40], as these were the techniques employed in our study.

- *Low Rank Adaptation (LORA):* Low-Rank Adaptation (LoRA) [39] employs a low-rank constraint to approximate the update matrix at each dense layer, thereby reducing the number of trainable parameters required for adapting to downstream tasks. In LoRA, the original parameter matrix $\mathbf{W}$ is frozen, while the parameter update $\Delta\mathbf{W}$ is approximated using low-rank decomposition matrices, $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are trainable parameters for task adaptation where $\Delta\mathbf{W} = \mathbf{A} \cdot \mathbf{B}^{\intercal}$ and $\mathbf{k}$ where, $\mathbf{W} \in \mathbb{R}^{m \times n}$, $\mathbf{A} \in \mathbb{R}^{m \times k}$, $\mathbf{A} \in \mathbb{R}^{k \times n}$ and $\mathbf{k}$ represents the reduced rank. Here, $\mathbf{A}$ and $\mathbf{B}$ are trainable paramters for rank adaptation and $k \ll min(m, n)$. The update equation can then be written in the general form as:

$$W \leftarrow W + \Delta W \tag{5}$$

  The key advantage of LoRA lies in its ability to significantly reduce memory and storage usage, such as VRAM. Additionally, LoRA allows for the maintenance of a single large model copy while incorporating multiple task-specific low-rank decomposition matrices for adaptation to various downstream tasks. Furthermore, studies have proposed principled approaches for setting the rank, including importance score-based allocation and search-free optimal rank selection.

3.3 Memory-Efficient Model Adaptation

In this section, we discuss the methods to reduce the memory consumption of Large Language Models (LLMs) through model quantization, a popular compression technique. As with the previous sections, it would be impractical to discuss everything and so this section will only focus on the quantization methods actually utilized for this project. Quantization involves mapping floating-point numbers to integers, typically using 8-bit integer quantization (INT8). Two main types of data are quantized in neural network models: weights (model parameters) and activations (hidden activations). Model quantization utilizes scaling and zero-point factors to transform floating numbers into quantized values. To illustrate the underlying principle of model quantization, we introduce a simple yet popular quantization function:

$$x_q = R(x/S) - Z \tag{6}$$

Here, x is the floating point number which needs to be quantized and x_q is the quantized value. In this function, S and Z denote the scaling factor (which determine the clipping range) and the zero-point factor (determining symmetric or asymmetric quantization) respectively and R denotes the rounding operation to map the scaled floating values to an integer.

The reverse process is called *dequantization* which revovers the original value from the quantized value as per:

$$\tilde{x} = S \cdot (x_q + Z) \tag{7}$$

The quantization error is calculated as the numerical difference between the original and recovered values. For LLMs, two major quantization approaches are quantization-aware training (QAT) and post-training quantization (PTQ). Given the vast number of parameters in LLMs, PTQ methods are preferred due to lower computational costs. Additionally, LLMs exhibit unique activation patterns, making quantization more challenging, especially for hidden activations. We focus on QLoRa, a representative PTQ method used in our study and on mixed-precision decomposition, due to their relevance to our research objectives.

- *Mixed-Precision Quantization*: As documented in [41], when the model size surpasses 6.7 billion parameters, there is a notable occurrence of exceptionally large values in hidden activations, commonly referred to as outliers. Intriguingly, these outliers predominantly manifest within specific feature dimensions at Transformer layers. In response to this observation, a novel approach termed LLM.int8() is introduced in [41]. This method employs vector-wise quantization to distinguish feature dimensions containing outliers from those without during matrix multiplication. Subsequently, computations for these distinct subsets utilize 16-bit floating-point numbers and 8-bit integers, respectively, with the aim of accurately capturing and handling these outlier values with precision. In our project, we extensively make use of INT-8 quantization when working with larger models.

- *Quantized Low Rank Adaptation (QLoRA):* In the realm of post-training quantization, direct conversion to low-bit quantization like INT8 often leads to significant performance declines. Addressing this issue, QLoRA [40] introduces supplementary, small, adjustable adapters (operating at 16-bit precision) into quantized models. This integration aims to facilitate efficient and high-precision fine-tuning of models. QLoRA combines the strengths of LoRA (refer Section 3.1) and various quantization techniques. Experimental findings demonstrate that QLoRA enables 4-bit quantized models to achieve performance comparable to full 16-bit fine-tuning and we use it in our study while fine-tuning our larger models.

4 Experiment

The ensuing section delves into the equipment utilized in our experiments, encompassing details on the dataset and its characteristics, a rundown of any external libraries employed, as well as the evaluation metrics utilized for assessing the models. Additionally, it provides insights into the LoRA parameters configuration employed during the fine-tuning of our models.

4.1 Dataset

The experiments were conducted using the following datasets, each accompanied by its respective description.

- *Defects4J:* Defects4J v1.2 stands as the most extensively utilized version of the Defects4J benchmark, housing a total of 393 bugs, with 130 of them categorized as single-hunk bugs. On the other hand, Defects4J v2.0 represents the latest iteration of the Defects4J benchmark, encompassing an additional 444 bugs, of which 108 are single-hunk bugs. Both Defects4J v1.2 and Defects4J v2.0 derive from prominent Java projects like the "Google Closure compiler".

- *HumanEval-Java:* Defects4J v1.2 and Defects4J v2.0 serve as commonly utilized benchmarks for Automated Program Repair (APR) techniques. However, relying solely on these benchmarks for applying Code Language Models (CLMs) may pose challenges, as CLMs could have encountered these datasets during their pre-training which could mean these models are biased to perform extremely well on these datasets. Investigation into data sources such as CodeSearchNet [42] and BigQuery reveals that four repositories utilized by the Defects4J benchmark are present in CodeSearchNet, while the entire Defects4J repository is encompassed by BigQuery. This suggests a high likelihood that existing APR benchmarks have been encountered by CLMs during pre-training. To mitigate this concern, we use a new benchmark named HumanEval-Java. HumanEval-Java addresses the issue of CLMs potentially having seen publicly available test datasets by manually converting Python programs and their test cases from HumanEval into Java programs and JUnit test cases. Bugs are then injected into the correct Java programs to create an APR benchmark. HumanEval-Java comprises 164 (single-hunk) Java bugs, ranging from simple bugs like incorrect operator usage to complex logical bugs requiring modification of multiple lines of code to rectify. Given its derivation from HumanEval and manual bug injection, HumanEval-Java ensures that none of the CLMs have encountered it previously.

Among these datasets, we employed the Defects4J dataset to establish our baseline models based on traditional APR techniques, while utilizing the HumanEval-Java dataset for our CLM models.

Dataset	Models
Defects4J	Baseline Models (ARJA, JAID, Hercules)
HumanEval-Java	PLBART, CodeT5, CodeGen, InCoder, ReFact, CodeLLaMA

Table 2: Datasets used along with the respective models

4.2 External Libraries

- *PEFT:* PEFT (Parameter-Efficient Fine-Tuning) [43] stands as a toolkit designed to adeptly tailor extensive pre-trained models for diverse downstream applications without the necessity of fine-tuning the entirety of a model's parameters, which can be excessively costly. Instead, PEFT techniques focus solely on refining a limited subset of additional model parameters, substantially mitigating computational and storage expenses, all the while maintaining performance levels akin to those achieved by fully fine-tuned models. This approach enhances the feasibility of training and accommodating large language models (LLMs) on consumer-grade hardware.
- *TRL:* TRL [44] is a comprehensive toolkit offering a range of utilities for training transformer language models using Reinforcement Learning, spanning from the initial Supervised Fine-tuning (SFT) phase, through the Reward Modeling (RM) phase, to the subsequent Proximal Policy Optimization (PPO) step. This library seamlessly integrates with transformers, providing a cohesive solution for leveraging Reinforcement Learning in the training process.
- *Transformers:* Transformers [45], developed by HuggingFace, is a library that furnishes APIs and utilities for effortlessly accessing and training cutting-edge pretrained models. Leveraging pretrained models can effectively reduce computational expenses, environmental impact, and the time and effort needed to train a model from the ground up. These models cater to a variety of tasks across different domains, including Natural Language Processing (NLP), Computer Vision (CV), Audio processing, and Multimodal tasks.

The versatility of Transformers extends to its framework interoperability, facilitating seamless transitions between PyTorch, TensorFlow, and JAX. This allows users to train a model in one framework with just a few lines of code and seamlessly switch to another framework for inference purposes. Furthermore, models can be exported to formats like ONNX and TorchScript for deployment in production environments.

4.3 Evaluation Metrics

The evaluation metric used for evaluating all the models was **pass@k**. The pass@k metric [14] is a quantitative measure used to evaluate the effectiveness of Automated Program Repair (APR) techniques. It assesses the success rate of generated patches by determining if any of the top k patches applied to a buggy program pass the provided test cases. In this study, patches are generated and injected into faulty programs, and the pass@k metric evaluates whether any of the top k patches result in passing all test cases. This metric provides insight into the APR technique's ability to produce effective patches for rectifying code faults, thereby gauging the repair process's overall quality and reliability. In this study models are evaluated and compared with pass@10.

4.4 LoRA Configuration

The below LoRA configuration was used to finetune the larger CLMs. The same LoRA configuration can be used to replicate the experiments if needed.

Parameter	Value
Rank (r)	16
Alpha (lora_alpha)	32
Dropout (lora_dropout)	0.05
Bias (bias)	"none"
Task Type	"CAUSAL_LM"

Table 3: LoRA configuration used to fine-tune models

5 Results and Analysis

In this section, we will explore the outcomes obtained from our baseline models, followed by the results from both Part A and Part B of the report. Subsequently, we will conduct an in-depth analysis of these findings to ascertain the extent to which we have met our research objectives. Additionally, we will report any pertinent discoveries gleaned from our analysis.

5.1 Baseline

The findings for the baseline models have been condensed into the table provided below taken from in reference [46]. Upon examination, it becomes evident that the performance of these baseline models is lacking, even when applied to relatively straightforward datasets such as Defects4J. Additionally, traditional methods exhibit considerable time requirements for both patch generation for buggy programs and subsequent evaluation of these patches.

Model	# of programs fixed (pass@10)	Accuracy
Search-Based	ARJA	5.04%
Constraint-Based	JAID	9.80%
Template-Based	Hercules	12.88%

Table 4: Baseline models accuracy on Defects4J dataset

5.2 Results from Part A

The outcomes from Part A have been integrated into Section 2.4 for convenience. Nevertheless, to facilitate easier interpretation of the results and findings, a plot has also been incorporated over here as well (Figure 3)

Based on the aforementioned results, the author has summarized their findings as follows:

1. *Performance of CLM-based models:* CLM-based models show significant proficiency in bug fixing compared to alternative methods. However, CodeT5 displays an exception, possibly due to its training across various code tasks beyond bug fixing. There is optimism that fine-tuning CodeT5 with data from Automated Program Repair (APR) could enhance its bug-fixing capabilities.

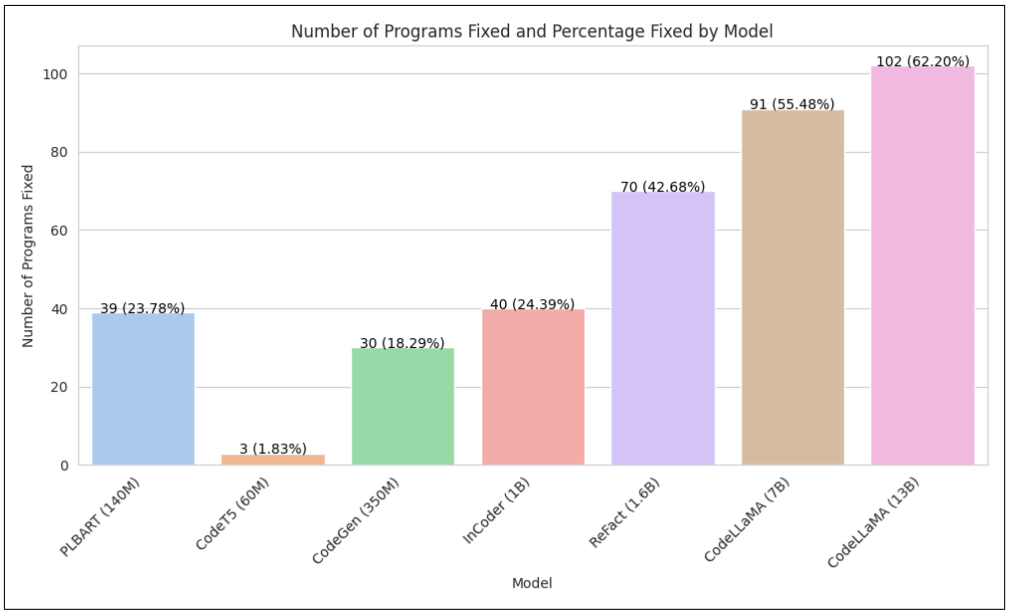

Figure 3: Pass@10 for CLM models on HumanEval-Java dataset

2. *Impact of Model Size on Bug Fixing:* A trend is observed indicating that as models increase in size, their bug-fixing capabilities also improve. This correlation suggests that larger models possess a more extensive grasp of natural language and benefit from emergent abilities with scaling which can be observed in Figure 4.

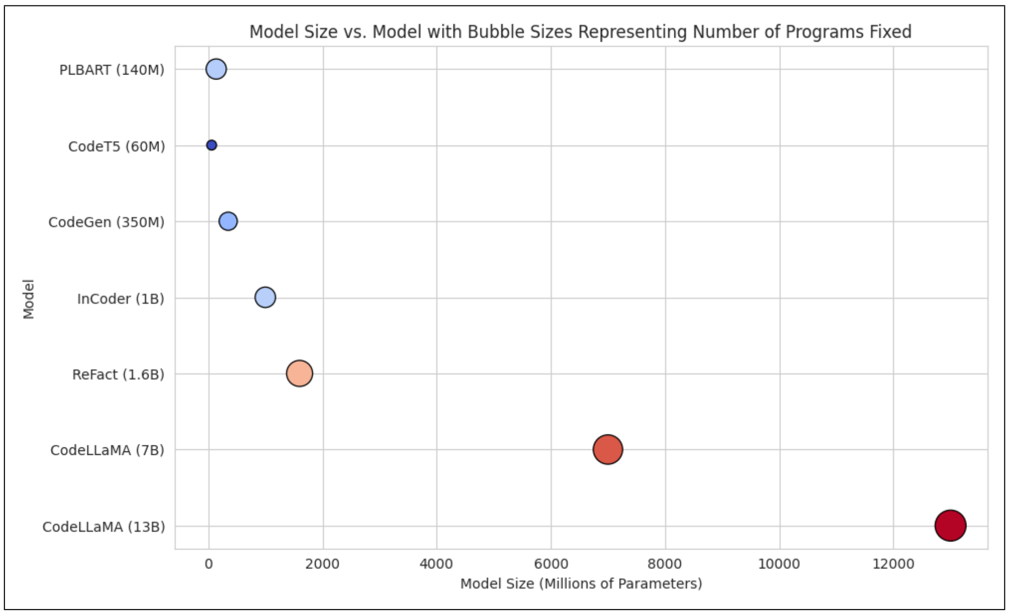

Figure 4: Correlation between model size and number of programs fixed

3. *Advantages of FIM-Enabled Models:* Models incorporating Fill-in-the-Middle (FIM) capabilities exhibit superior bug-fixing performance compared to conventional masked language models. FIM models consider both prefix and suffix components of the code, leveraging contextual information for result generation. Moreover, these models are trained to generate larger amounts of missing text, thereby enhancing their effectiveness.

5.3 Results from Part B

The models underwent fine-tuning on custom code data pertinent to Automated Program Repair (APR) tasks, as detailed in Section 3. The outcomes for pass@10 for the fine-tuned models are provided below:

Models (Finetuned)	# of programs fixed (pass@10)
PLBART (140M)	41 (25.00%)
CodeT5 (60M)	39 (23.78%)
CodeGen (350M)	50 (30.48%)
InCoder (1B)	61 (37.19%)
ReFact (1.6B)	**98 (59.75%)**

Table 5: Pass@10 for finetuned CLM models on HumanEval-Java dataset

A comparative analysis using side-by-side graphs of the zero-shot models and the fine-tuned models aids in comprehensively assessing any performance improvements resulting from fine-tuning the models on code data. This approach offers valuable insights into the efficacy of fine-tuning in enhancing model performance and can be seen in Figure 5.

Based on the aforementioned results, the author has summarized their findings as follows:

1. *Integration of New Capabilities:* Notably, all models exhibit performance gains following fine-tuning; however, CodeT5 appears to derive the most significant benefit. This phenomenon may stem from CodeT5's training on a comprehensive multi-task pretraining objective. Despite this, it is evident that these models possess latent programming knowledge acquired during their pre-training phase, which can be unlocked and further enhanced through fine-tuning. This observation serves as compelling evidence that LLMs refine or acquire new abilities when exposed to similar data during their pre-training stages.

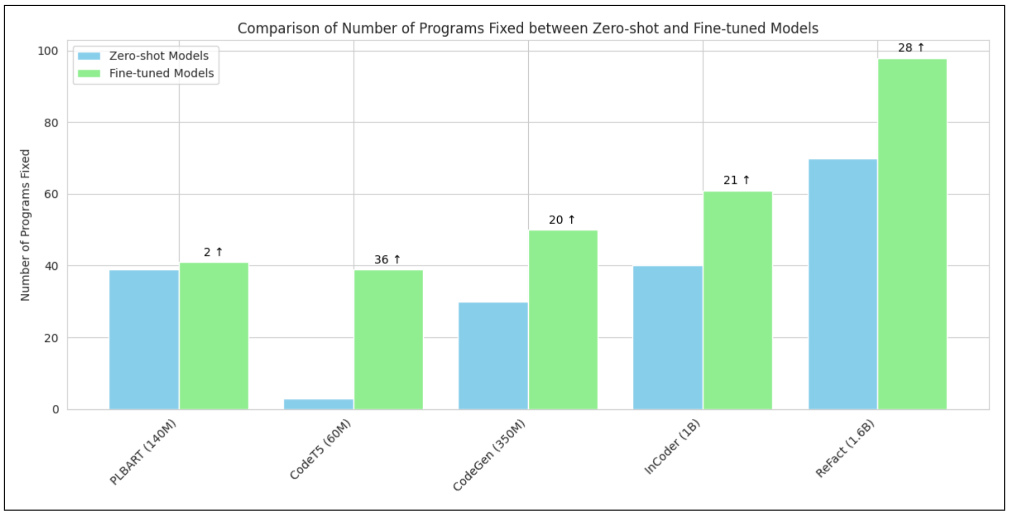

Figure 5: Comparison between zero-shot models and finetuned models on pass@10 on HumanEval-Java dataset

2. *Impact of Model Size on Bug Fixing:* As evidenced in the preceding section, larger models tend to derive the greatest benefits from pre-training or fine-tuning tasks, thereby showcasing their abilities to the fullest extent. Even following fine-tuning, all models exhibit improvement; however, the notable enhancement demonstrated by larger models such as Incoder and Refact underscores the significance of model size in dealing with LLMs.

6 Future Work

This study provides valuable insights into the evolution and effectiveness of Code Language Models (CLMs) in bug fixing tasks. However, there are several avenues for future research to explore further:

- *Fine-tuning with APR Data:* This study uses general code data as a base for it's experiments. However, the author believes that fine-tuning CLMs with Automatic Program Repair (APR) based domain specific code data could potentially enhance their bug fixing capabilities. Future studies could investigate the effectiveness of fine-tuning CLMs using APR-specific datasets and explore the impact on bug fixing performance.

- *Comprehensive Evaluation:* While this study focused on selected CLMs due to their popularity and availability, future research could conduct a more comprehensive evaluation of various CLMs. This could involve comparing a wider range of models and analyzing their performance across different bug types and programming languages.

- *Development of New Metrics:* The current study primarily assessed CLMs based on their bug fixing effectiveness. Future work could involve developing new metrics or refining existing ones to provide more comprehensive evaluations of CLMs' performance, considering factors such as computational efficiency, robustness, and scalability.

- *Exploration of Advanced Techniques:* With the rapid advancements in machine learning and natural language processing, future research could explore advanced techniques and methodologies for CLM development and optimization. This could include incorporating advanced architectural designs, exploring novel fine-tuning strategies, or leveraging model alignment learning approaches.

- *Mixture-of-Experts:* With the ongoing advancements in LLM research, new breakthroughs emerge frequently. Currently, Mixture-of-Experts (MoE) models have gained significant attention, as evidenced by the recent release of the Mistral models [47]. These models have demonstrated notable effectiveness in various tasks. Future research endeavors could explore the potential of utilizing MoE-based LLMs, which hold promise for further enhancing efficiency in bug-fixing tasks.

- *Agent based LLMs and Tool Manipulation:* In essence, LLMs are primarily trained as text generators using extensive plain text datasets, which may result in suboptimal performance on tasks that are not inherently textual, such as numerical computation or coding. To address these limitations, a recent approach involves leveraging external tools to supplement the capabilities of LLMs [48]. For instance, LLMs can utilize calculators for precise computations and utilize search engines to retrieve relevant information or find solutions to encountered bugs. This approach resembles an agent-based mechanism where the model continuously searches for and applies fixes until the issue is resolved.

- *Real-world Deployment and Validation:* Further research is needed to validate the effectiveness of CLMs in real-world software development scenarios. Future studies could focus on deploying CLMs in industry settings and evaluating their performance, usability, and impact on development workflows and software quality.

- *Ethical and Societal Implications:* As CLMs become more prevalent in software development, it is essential to consider the ethical and societal implications of their use. Future research could explore the ethical considerations surrounding CLMs, such as bias mitigation, fairness, and transparency, and investigate approaches to ensure responsible and ethical deployment of these models.

By addressing these future research directions, we can further advance our understanding of CLMs and their role in bug fixing and software development, ultimately leading to more efficient and reliable software systems.

7 Conclusion

This study embarked on an investigation to assess the evolution of Code Language Models (CLMs) over time and the impact of these advancements on their effectiveness. The results highlight a noticeable improvement in CLMs, primarily attributed to their Fill-in-the-Middle (FIM) capability. Integrating FIM into code-based models enhances their ability to comprehend the broader context of code, resulting in more effective bug fixing compared to traditional mask-based models.

While this study focused on two selected models chosen for their popularity and availability on platforms like Hugging Face, it's important to acknowledge the dynamic nature of CLM releases, with new models emerging weekly. Therefore, a selective approach was necessary, and although valuable insights were gained, a comprehensive examination of all models would be impractical. Future research may delve into a more exhaustive investigation.

Despite being trained on general code tasks, the observed performance of these CLMs suggests potential for further improvement. The author speculates that fine-tuning these models with Automatic Program Repair (APR) based code data could yield even more promising results, paving the way for future research.

In summary, CLMs show significant potential for bug fixing, serving as valuable tools that bridge the gap between Natural Language Processing (NLP) and Software Engineering. Furthermore, they serve as a connecting link between academia and industry, demonstrating their versatility and applicability in real-world software development scenarios. The enhancement of CLMs through fine-tuning in Part B of this study further emphasizes their efficacy and underscores the importance of ongoing research in this field.

Acknowledgements

The author expresses profound gratitude to their esteemed supervisors, Boyu Zhang, Limeng Zhang and Prof. Ali Babar, whose unwavering support and guidance were instrumental in the realization of this work. Without their invaluable input, this research would not have come to fruition. The author extends sincere thanks to the Centre for Research on Engineering Software Technologies (CREST) and the University of Adelaide for providing the necessary support and resources essential for the successful completion of this endeavor.

Furthermore, the author acknowledges and appreciates the authors of the models utilized in this study for their generosity in making their work open source and accessible to the wider research community. This openness has significantly contributed to the advancement of knowledge in the field. The author recognizes the pivotal role played by these individuals and institutions, without whom this work would not have been possible, and expresses heartfelt gratitude.

A Appendices

For the sake of transparency, openness, and reproducibility, the author has made the codebase and findings of this research project publicly available on CREST'S GitHub repo. Interested readers can access the materials and repository from here.

This ensures that the research work can be thoroughly examined, reproduced, and built upon by the scholarly community and interested individuals.

B References

1. M. Monperrus, "The living review on automated program repair," 2020.

2. C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," Commun. ACM, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

3. M. Monperrus, "Automatic software repair: A bibliography," ACM Comput. Surv., vol. 51, no. 1, pp. 17:1–17:24, 2018. [Online]. Available: https://doi.org/10.1145/3105906

4. Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair," TSE, 2019.

5. Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-Based Code Transformation Learning for Automated Program Repair," in ICSE. ACM, 2020, p. 602–614.

6. T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair," in ISSTA. ACM, 2020, p. 101–114.

7. Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 416–426.

8. J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," IEEE Trans. Software Eng., vol. 43, no. 1, pp. 34–55, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

9. M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings, ser. Lecture Notes in Computer Science, T. E. Colanzi and P. McMinn, Eds., vol. 11036. Springer, 2018, pp. 65–86. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9 3

10. Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, “Trust enhancement issues in program repair,” in Proceedings of the 44th International Conference on Software Engineering, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2228–2240. [Online].

11. Jiang, N. et al. (2023) ’Impact of code language models on automated program repair,’ arXiv (Cornell University) [Preprint]

12. Rozière, B. et al. (2023) ’Code llama: Open Foundation Models for code,’ arXiv (Cornell University) [Preprint].

13. Vaswani, A. et al. (2017) ’Attention is all you need,’ arXiv (Cornell University) [Preprint]. https://doi.org/10.48550/arxiv.1706.03762.

14. Chen, M. et al. (2021) ’Evaluating large language models trained on code,’ arXiv (Cornell University) [Preprint].

15. S. Pinker, The Language Instinct: How the Mind Creates Language. Brilliance Audio; Unabridged edition, 2014.

16. A. M. Turing, “Computing machinery and intelligence,” Mind, vol. LIX, no. 236, pp. 433–460, 1950.

17. R. Rosenfeld, “Two decades of statistical language modeling: Where do we go from here?” Proceedings of the IEEE, vol. 88, no. 8, pp. 1270–1278, 2000.

18. S. M. Katz, “Estimation of probabilities from sparse data for the language model component of a speech recognizer,” IEEE Trans. Acoust. Speech Signal Process., vol. 35, no. 3, pp. 400–401, 1987.

19. W. A. Gale and G. Sampson, “Good-turing frequency estimation without tears,” J. Quant. Linguistics, vol. 2, no. 3, pp. 217–237, 1995.

20. Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, “A neural probabilistic language model,” J. Mach. Learn. Res., vol. 3, pp. 1137–1155, 2003.

21. T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States, C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, Eds., 2013, pp. 3111–3119.

22. T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Eff icient estimation of word representations in vector space,” in 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, Y. Bengio and Y. LeCun, Eds., 2013.

23. M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” in Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers), M. A. Walker, H. Ji, and A. Stent, Eds. Association for Computational Linguistics, 2018, pp. 2227–2237.

24. J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186.

25. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever et al., “Language models are unsupervised multitask learners,” OpenAI blog, p. 9, 2019.

26. M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020, 2020, pp. 7871–7880.

27. J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” CoRR, vol. abs/2001.08361, 2020.

28. Brown, T.B. et al. (2020) Language models are few-shot learners, arXiv.org.

29. Chowdhery, A. et al. (2022) Palm: Scaling language modeling with pathways, arXiv.org.

30. Touvron, H. et al. (2023) Llama: Open and efficient foundation language models, [2302.13971v1] LLaMA: Open and Efficient Foundation Language Models.

31. Bavarian, M. et al. (2022) 'Efficient training of language mod- els to fill in the middle,' arXiv (Cornell University) [Preprint].

32. Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. IEEE Trans. Software Eng. 38, 1 (2012), 54–72.

33. Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA. 298–309.

34. Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In 35th International Conference on Software Engineering, ICSE. 772–781.

35. Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In 35th International Conference on Software Engineering, ICSE. 802–811.

36. Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang-Mao,andYvesLeTraon.2020. Ontheefficiencyoftestsuitebasedprogram-repair:ASystematicAssessment of 16 Automated Repair Systems for Java Programs. In 42nd International Conference on Software Engineering, ICSE. 615–627.

37. Zhang, Shengyu et al. (2024) Instruction tuning for large language models: A survey, [2308.10792] Instruction Tuning for Large Language Models: A Survey

38. Ouyang, L. et al. (2022) Training language models to follow instructions with human feedback, [2203.02155] Training language models to follow instructions with human feedback.

39. Hu, E.J. et al. (2021) Lora: Low-rank adaptation of large language models, arXiv.org.

40. Dettmers, T. et al. (2023) Qlora: Efficient finetuning of quantized llms, arXiv.org.

41.] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm.int8(): 8-bit matrix multiplication for transformers at scale," CoRR, vol. abs/2208.07339, 2022.

42. H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," CoRR, vol. abs/1909.09436, 2019. [Online]

43. Huggingface/PEFT: PEFT: State-of-the-art parameter-efficient fine-tuning., GitHub.

44. Huggingface/TRL: Train transformer language models with reinforcement learning., GitHub.

45. Wolf, T. et al. (no date) Transformers: State-of-the-art natural language processing, ACL Anthology.

46. Huang, K. et al. (2023) A survey on Automated Program Repair Techniques, arXiv.org.

47. Jiang, A.Q. et al. (2024) Mixtral of Experts, arXiv.org.

48. T. Schick, J. Dwivedi-Yu, R. Dess‘ ı, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," CoRR, vol. abs/2302.04761, 2023.