Improving the Ability of Pre-trained Language Model by Imparting Large Language Model's Experience

Xin Yin, Chao Ni*, Xinrui Li, Xiaohu Yang

[a] *The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China*

Abstract

Large Language Models (LLMs) and pre-trained Language Models (LMs) have achieved impressive success on many software engineering tasks (e.g., code completion and code generation). By leveraging huge existing code corpora (e.g., GitHub), these models can understand the patterns in source code and use these patterns to predict code properties. However, LLMs under few-shot learning perform poorly on non-generative tasks (e.g., fault localization and vulnerability localization), and fine-tuning LLMs is time-consuming and costly for end users and small organizations. Furthermore, the performance of fine-tuning LMs for non-generative tasks is impressive, yet it heavily depends on the amount and quality of data. As a result, the current lack of data and the high cost of collecting it in real-world scenarios further limit the applicability of LMs. In this paper, we leverage the powerful generation capabilities of LLMs to enhance pre-trained LMs. Specifically, we use LLMs to generate domain-specific data, thereby improving the performance of pre-trained LMs on the target tasks. We conduct experiments by combining different LLMs in our generation phase and introducing various LMs to learn from the LLM-generated data. Then, we compare the performance of these LMs before and after learning the data. We find that LLM-generated data significantly enhances the performance of LMs. The improvement can reach up to 58.36% for fault localization and up to 6.09% for clone detection.

Keywords: Language Model, Fault Localization, Clone Detection

*Chao Ni is the corresponding author.
He is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

1. Introduction

Large Language Models (LLMs) [56, 2, 85, 40, 66, 45, 44, 23] have been widely adopted due to advances in Natural Language Processing (NLP). These advances allow LLMs to be trained with billions of parameters and samples, resulting in significant performance improvements on various tasks. LLMs can easily be used for code-related tasks by being fine-tuned [79, 75] or prompted [70, 68, 75, 77] since they are trained to be general and can capture knowledge from various domains. LLMs perform better as their computational budget increases [27]. For instance, their performance on program synthesis benchmarks improves linearly with the number of model parameters [71]. However, many existing works in software engineering, such as fault localization and vulnerability detection, either train small models from scratch [32, 34] or fine-tune modest-sized models [43, 49]. They often overlook the potential benefits of fine-tuning state-of-the-art LLMs. This oversight occurs because fine-tuning LLMs is time-consuming and computationally expensive for end users and small organizations.

Recently, pre-trained Language Models (LMs) have shown remarkable success in various software engineering tasks [14, 17, 16, 1, 64, 63]. By leveraging huge existing code corpora (e.g., GitHub), these models can understand the patterns in source code and use these patterns to predict code properties. These models are small but powerful, usually having between 110M and 220M parameters. Therefore, they require less time and computational resources than LLMs to fine-tune for various software engineering tasks, and their effectiveness has been widely demonstrated [49, 69, 15, 18]. However, their performance significantly relies heavily on the amount and quality of data. The current scarcity of data, coupled with the high cost of data collection in real-world scenarios, presents a major challenge to the practical application of these models. Traditional generation tools can be used to generate data, but different tools are required to generate data specific to different tasks. There is no unified tool for generating data across all tasks, and many tools are based on specific patterns for data generation. For example, mutation testing tools can be used to generate faults, but they typically rely on predefined patterns to create mutations. These patterns are manually designed to simulate common programming errors or code changes, yet they may not cover all possible code variations.

Benefiting from pre-training on large-scale code corpora, LLMs possess the ability to generate large-scale and diverse code (e.g., buggy function

and code clone). To address the challenge of data scarcity, we leverage the powerful generation capabilities of LLMs to enhance pre-trained LMs. Specifically, we use LLMs to generate domain-specific data, thereby improving the performance of pre-trained LMs on the target tasks.

We evaluate the LLMs' capability to improve the LMs on two non-generative tasks (i.e., fault localization and clone detection) by using eight popular LLMs to generate faults and clones from Defects4J [25] and HumanEval-X [84] datasets, which can be divided into two categories: five code LLMs (i.e., Magicoder [66], DeepSeek-Coder [2], OpenCodeInterpreter [85], CodeLlama [56], and WizardCoder [40]) and three general LLMs (i.e., Llama 3 [44], Mistral [23], and Phi-2 [45]). We then design selection strategies to identify high-quality data and introduce various LMs to learn from the LLM-generated data. The LMs we utilize have fewer than 220M parameters and can be categorized into two types: encoder-only LMs (i.e., BERT [7], RoBERTa [36], CodeBERT [14], GraphCodeBERT [17], and UniXcoder [16]) and encoder-decoder LMs (i.e., PLBART [1], CodeT5 [64], and CodeT5+ [63]).

We compare the performance of these LMs before and after learning the LLM-generated data. We find that LLM-generated data significantly enhances the performance of LMs. The improvement can reach up to 58.36% for fault localization and up to 6.09% for clone detection. For example, in fault localization, by adding 30% of the generated data to the training set, GraphCodeBERT's F1-score increases from 0.384 to 0.550, Recall increases from 0.353 to 0.522, Precision increases from 0.421 to 0.582, and Accuracy increases from 0.889 to 0.901. Moreover, after learning from the LLM-generated data, LMs show an improvement of 0.74%∼6.09% on the CodeXGLEU-POJ104 [39] dataset. Similarly, on the CodeNet-Java250 [54] dataset, LMs exhibited an increase of 0.13%∼3.23%. Our study highlights that using LLMs to generate data for LMs can improve performance by a large margin.

To summarize, the main contributions of this paper are:

- We generate additional data for fault localization and code clone detection using LLMs and employ selection strategies to ensure data quality.
- We improve the capabilities of LMs on non-generative tasks using LLM-generated data. Our evaluation demonstrates that the LMs' effectiveness improves significantly after learning from the generated data.
- We conduct comprehensive experiments on eight LLMs and eight LMs using widely studied datasets [25, 42, 57, 84, 39, 54] to explore their effectiveness.

Our generated data, fine-tuned models, and code are publicly available [74].

The remainder of this paper is organized as follows. Section 2 provides information on the related work. Section 3 overviews our proposed approach. Experimental design and results are then described in Section 4 and Section 5, respectively. We provide discussion in Section 6 and conclude in Section 7.

2. Related Work

2.1. Language Model

Language Models can perform tasks such as clone detection and code generation. We discuss such models by summarizing them into pre-trained Language Models and Large Language Models.

Pre-trained Language Models. Pre-trained Language Models (LMs) are usually trained on a large volume of data and can be classified into two types of architectures: encoder-only and encoder-decoder. Encoder-only (e.g., BERT [7] and CodeBERT [14]) and encoder-decoder (e.g., PLBART [1] and CodeT5 [64]) models are trained using Masked Language Modeling (MLM) or Masked Span Prediction (MSP) objective, respectively, where a small portion (e.g., 15%) of the tokens are replaced with either masked tokens or masked span tokens, models are trained to recover the masked tokens. Apart from tasks in NL (e.g., cloze test and question answering), they are also widely used in code-related tasks (e.g., fault localization and clone detection). These models are small but powerful, usually having between 110M and 220M parameters.

Large Language Models. Large Language Models (LLMs) [56, 2, 85, 40, 66, 45, 44, 23] have been widely adopted since the advances in Natural Language Processing which enable LLMs to be well-trained with both billions of parameters and billions of training samples, which consequently brings a large performance improvement on tasks adopted by LLMs [75, 3, 51]. These models can be easily used for a downstream task by being fine-tuned [79, 75] or being prompted [70, 68, 76, 77] since they are trained to be general and they can capture different knowledge from various domains. Fine-tuning is used to update model parameters for a particular downstream task by iterating the model on a specific dataset while prompting can be directly used by providing natural language descriptions or a few examples of the downstream task. Compared to prompting, fine-tuning is expensive since it requires additional model training and has limited usage scenarios, especially in cases where sufficient training datasets are unavailable.

2.2. Machine Learning Fault Localization

Machine Learning Fault Localization (MLFL) techniques use program analysis to understand code behavior. They've used various data types like test coverage metrics [4, 82], co-changing method declarations [35], and structural information from the code, such as the abstract syntax tree (AST) [34]. Recent approaches, like GRACE [38] and FixLocator [35], encode both AST and test coverage into graph representations. These methods prioritize faulty methods by preserving all topological dependencies with graph neural networks. DeepRL4FL [34] uses a convolution neural network to analyze code coverage matrices. DeepFL [32] and TRANSFER-FL [43] integrate features based on semantics, spectrum, and mutations using a multi-layer perceptron model. LLMAO [72], unlike previous techniques, doesn't need test code or an AST parser. It includes a text tokenizer and embedding layer and leverages attention mechanisms and bidirectional adapter layers on pre-trained left-to-right LLMs directly on source code.

2.3. Code Clone Detection

Code clone detection aims to extract similar pairs of code snippets from large code bases. Several novel neural models [65, 80, 61] have been proposed by the community for this purpose, leveraging abstract syntax tree or data flow information obtained from compilers to comprehend code functionality. Pre-trained LMs have achieved impressive success on many software engineering tasks [16, 64, 63], including code clone detection, through fine-tuning. During the fine-tuning stage, code snippets undergo encoding into low-dimensional dense vectors. In this process, the similarity between vectors in the latent space is amplified for clone pairs, while it is diminished for non-clone pairs.

3. Approach

In this section, we begin by outlining our motivation for task selection. Then, we discuss the models selected for generation and evaluation. Next, we demonstrate practical methods for using LLMs to generate data, along with selection strategies to ensure data quality. Finally, we describe techniques for fine-tuning pre-trained LMs in fault localization and clone detection tasks. The overview of our approach is shown in Fig. 1.

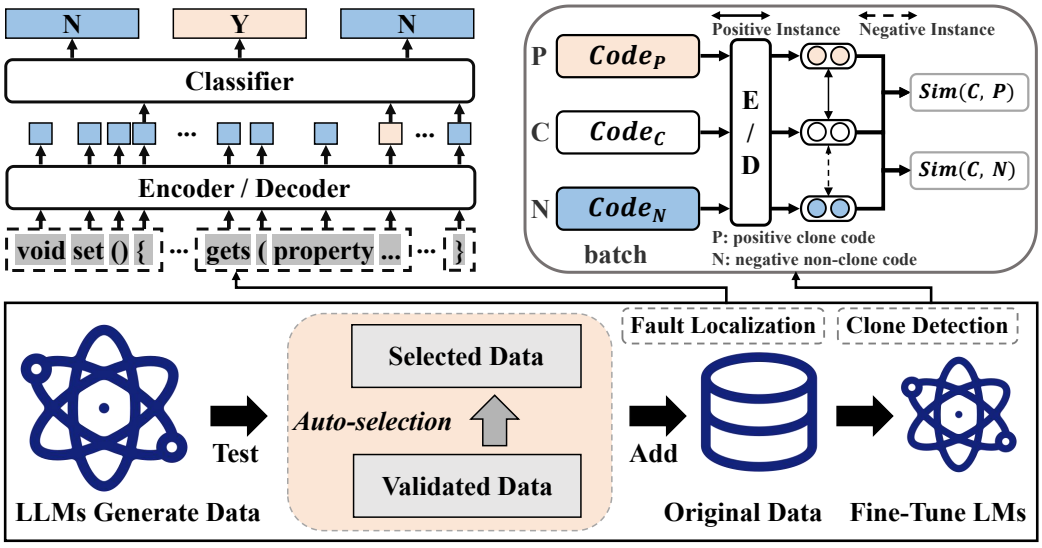

Figure 1: Fine-tune LMs to learn from LLM-generated data

3.1. Task Selection

While existing research on LLM distillation and synthetic data generation has made significant progress [9, 84, 50, 66, 40, 5], these works primarily focus on generative tasks (e.g., code generation, dialogue, and summarization). The key innovation of our work lies in being the first to systematically apply LLM-generated synthetic data to enhance the capabilities of pre-trained LMs on discriminative (non-generative) software engineering tasks, including fault localization and code clone detection. We not only propose a comprehensive implementation framework but also demonstrate its effectiveness through extensive experiments. We believe this work provides novel insights and empirical evidence for data augmentation and model enhancement in discriminative (non-generative) tasks, offering substantial theoretical and practical value to the field.

For non-generative software engineering tasks, we strategically select fault localization and code clone detection as our target domains for validation. These tasks are critical in software engineering, and we choose them for four key reasons:

- **Task Representativeness**. These tasks are central to the field of software engineering. fault localization directly impacts software quality and reliability, making it a crucial task in software testing and maintenance.

Code clone detection, on the other hand, is vital for code maintenance, refactoring, and intellectual property protection.

- **Research Value**. Both tasks hold significant importance within the software engineering community and are currently hot research topics, having been extensively studied and surveyed [4, 82, 34, 35, 38, 32, 43, 72, 55, 62, 75, 19, 65, 80, 61, 16, 64, 63, 29, 13, 10, 28]. However, despite their prominence, there's limited research comparing LLMs and pre-trained LMs on these tasks, and even less on the transferability of LLM-generated knowledge to LMs.

- **Method Validation**. While both are non-generative tasks, they present distinct characteristics and challenges, allowing us to thoroughly validate the effectiveness of our proposed LLM-assisted data generation method. fault localization focuses on identifying code errors for quality assurance, while clone detection deals with code similarity for code comprehension, covering different facets of software engineering.

- **Practical Applications**. These two tasks have broad applications in real-world software development. fault localization helps developers quickly find and fix code errors, thereby improving software reliability. Code clone detection identifies and manages duplicate code, which enhances maintainability and reduces the risk of propagating potential errors.

3.2. Studied Language Models

We describe the various pre-trained LMs we use for evaluation. As shown in Table 1, these models have fewer than 220 million parameters and can be categorized into two categories: encoder-only LMs and encoder-decoder LMs. Encoder-only LMs (i.e., BERT [7], RoBERTa [36], CodeBERT [14], GraphCodeBERT [17], and UniXcoder [16]) contain only the encoder component of a Transformer. They are designed for learning data representations and trained using the Masked Language Modeling (MLM) objective. For UniXcoder, we utilize the encoder-only mode. This is because UniXcoder is part of the CodeBERT series, and our objective is to compare UniXcoder (in encoder-only mode) with other models from the series, such as CodeBERT and GraphCodeBERT, under consistent conditions. Encoder-decoder LMs (i.e., PLBART [1], CodeT5 [64], and CodeT5+ [63]) have been proposed for sequence-to-sequence tasks. They are trained to recover the correct output sequence given the original input, often through span prediction tasks

where random spans are replaced with artificial tokens. All these models can potentially be used for classification in our tasks, so we evaluate these state-of-the-art LMs.

3.3. Studied Large Language Models

We describe the various open-source LLMs we employ for evaluation. General LLMs are trained on textual data, encompassing both natural language and code, and are versatile for numerous tasks. Conversely, code LLMs are tailored specifically for automating code-related tasks. Among the code LLMs, we select the five models released recently (in 2024), namely Magicoder [66], DeepSeek-Coder [2], OpenCodeInterpreter [85], CodeLlama [56], and WizardCoder [40]. For the general LLMs, we choose the top three models: Llama 3 [44], Mistral [23], and Phi-2 [45]. While many closed-source LLMs (e.g., Claude, GPT-4, and Gemini) have demonstrated rapid advancements and often superior performance, these models incur additional API costs, posing challenges for individual users and smaller organizations. Furthermore, organizations may have data security requirements, making it important to prioritize data privacy. The nature of these closed-source LLMs raises data security concerns, leading us to choose open-source, smaller-scale LLMs for better accessibility and privacy protection. Therefore, we use eight open-source LLMs to generate data. These models have demonstrated strong capabilities across various software engineering tasks, such as program repair [77, 73], unit test generation [78, 48], and code generation [53, 9]. With parameter sizes ranging from 2.7B to 8B, they are well-suited for individual users and smaller organizations, while also mitigating data security concerns. A summary of the characteristics of these selected LLMs is presented in Table 1.

Table 1: Overview of the studied models

Studied LMs	**# Para.**	**Model Type**	**Studied LLMs**	**# Para.**	**Model Type**
BERT	110M	Encoder-only LM	Magicoder	6.7B	Code LLM
RoBERTa	125M	Encoder-only LM	DeepSeek-Coder	6.7B	Code LLM
CodeBERT	125M	Encoder-only LM	OpenCodeInterpreter	6.7B	Code LLM
GraphCodeBERT	125M	Encoder-only LM	CodeLlama	7B	Code LLM
UniXcoder	125M	Encoder-only LM	WizardCoder	7B	Code LLM
PLBART	140M	Encoder-decoder LM	Phi-2	2.7B	General LLM
CodeT5	220M	Encoder-decoder LM	Mistral	7B	General LLM
CodeT5+	220M	Encoder-decoder LM	Llama 3	8B	General LLM

*For UniXcoder, we use encoder-only mode.

3.4. LLM-based Data Generation

In our study, we use two non-generative tasks for evaluation: fault localization and clone detection. We now describe how to generate data for these tasks.

3.4.1. Fault Generation

Since we evaluate function-level fault localization, we use LLMs to autoregressively inject faults into the original non-buggy functions. However, since LLMs are not specifically pre-trained for this task, simply providing them with the non-buggy function will not suffice. To facilitate the direct usage of LLMs for fault generation, we use a specific prompt to enable the LLMs to perform few-shot learning. This allows the LLMs to recognize the task and generate a buggy function by completing the input provided. We follow the prompt similar to those used in the artifacts, papers, or technical reports [58, 50, 31, 56, 75].

To help LLM perform downstream tasks effectively, a well-crafted prompt is crucial, a topic explored by various researchers [70, 67, 12]. Previous research [83] suggests that including a few diverse examples can enhance LLM's generalization ability. To achieve this, we employ an advanced selection strategy (i.e., Semantic-based Selection) to select semantically similar examples. This selection strategy adopts a pre-trained model (i.e., UniXcoder, which effectively comprehends code semantic information [16]) to embed all the functions and then uses the K-means algorithm [41] for clustering. We cluster the examples (i.e., pairs of non-buggy and buggy functions) from the training set of Defects4J based on their semantic similarity to ensure the selection of distinct examples [83, 33]. Considering the limitation of LLM's conversation windows, we adopt a two-shot experimental setup: we cluster all examples into two clusters and then select the most semantically similar example from each cluster based on cosine similarity.

Fig. 2 illustrates the prompt, consisting of two fault generation examples designed to showcase the task and the desired output format. Each example contains four elements: (1) task description, (2) non-buggy function, (3) indicator, and (4) buggy function, while input contains only the first three elements.

- **Task Description**. We provide LLM with the description constructed as *"// Inject a bug for the non-buggy function"*. The task descriptions used in

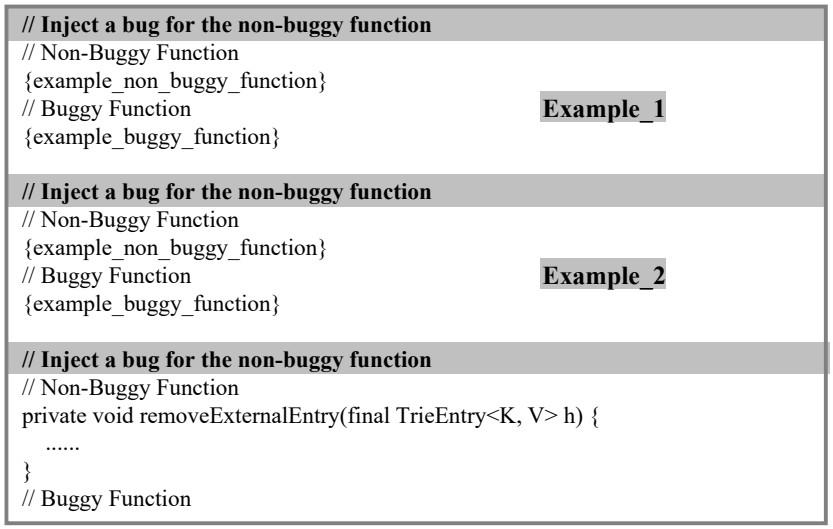

Figure 2: An example of prompt for fault generation

the fault generation task vary based on the source programming language we employ.

- **Non-Buggy Function**. We provide LLM with the non-buggy function. We also prefix with *"// Non-Buggy Function"* to directly indicate LLM about the context of the function.
- **Indicator**. We instruct LLM to think about the results. We follow the best practice in previous works [68, 75] and adopt the similar indicator named *"// Buggy Function"*.
- **Buggy Function**. We demonstrate the expected output in the example, showing the buggy function produced after injecting faults into the non-buggy function.

Given a corpus of non-buggy functions, we use the decorated prompt to collect buggy functions by injecting faults into the non-buggy functions.

3.4.2. Clone Generation

Previous studies [40, 2, 85, 63, 56, 66] have assessed the ability of LLMs to generate code from natural language specifications in a zero-shot setting. Similarly, in clone generation, we can guide LLMs to generate similar code clones for the same natural language specification. Leveraging the official

prompts from HumanEval-X [84], we generate clones in both C++ and Java languages.

3.5. Data Validation and Selection

To effectively evaluate the data generated by LLMs, we need to verify and filter out invalid data, and then select high-quality data for evaluation.

3.5.1. Data Validation.

We compile and run test suites (originally supported by the studied datasets, cf. Section 4.1) to verify all candidate code snippets generated by LLM. For fault generation, we retain only the buggy functions that fail to pass the test cases. For clone generation, we keep the generated functions that successfully pass the entire test suite.

3.5.2. Data Selection.

To select high-quality fault data, we analyze the Defects4J training set across three dimensions: (1) the average number of lines changed (LC_{ave}), determined by comparing file differences using the git diff command; (2) the average edit distance (ED_{ave}), calculated using Levenshtein edit distance between non-buggy and buggy functions; and (3) the average semantic similarity (SS_{ave}), obtained by inputting non-buggy and buggy functions into CodeBERT to retrieve vector representations and then computing cosine similarity. We then compute LC, ED, and SS for each generated data and calculate a score using the formula: $Score = |(LC - LC_{ave})/LC_{ave}| + |(ED - ED_{ave})/ED_{ave}| + |(SS - SS_{ave})/SS_{ave}|$. A lower score indicates closer alignment with the Detects4J training set's distribution. Since LLMs generate multiple buggy functions for each non-buggy function, to ensure diversity, we select the buggy function with the lowest score from the generated data, resulting in a unique pair of non-buggy and buggy functions. Finally, we select a subset or all of the data based on the $Score$, ranging from low to high (e.g., 10%×Generated in Section 5.3).

For generated clone data, our goal is to maximize differences. We calculate the average edit distance among all generated code clones for each task in HumanEval-X. Additionally, we compute the average relative edit distance for each code with other code clones, selecting those with an average relative edit distance greater than or equal to the overall average edit distance.

3.6. Model Fine-Tuning

As illustrated in Fig. 1, we demonstrate our process of selecting data from validated data and integrating it into the original dataset. Next, we fine-tune the LMs on two downstream tasks: fault localization and clone detection. We utilize eight LMs mentioned in Table 1 for our study, though other LMs like CuBERT [26] can be used interchangeably.

For fault localization, we follow previous works [72, 79] that use LMs to classify individual code lines as either buggy or non-buggy. For a token sequence $T = \{t_1, t_2, ..., t_n\}$ of the function, the model's encoder or decoder component, denoted as M, processes T to yield a sequence of output vectors: $O = M(T) = \{o_1, o_2, ..., o_L\}$, where O represents the output tensor with dimensions $L \times H$, L signifies the sequence length, and H denotes the hidden dimension size. During the process, the contextual information is captured by the self-attention or masked self-attention mechanisms in the encoder or decoder of LMs, where masked self-attention limits the sight to the preceding part of tokens. Each output vector o_i that represents the last token of one line is subsequently associated with a label (i.e., 0 or 1). The optimization process employs the binary cross-entropy as the loss function.

Regarding clone detection, we follow prior works [16, 64, 63] that directly use LMs to obtain embeddings of code snippets and then measure the similarity between two code snippets to predict whether they share common functionality.

4. Experimental Design

4.1. Datasets

4.1.1. Fault Datasets

To ensure the thoroughness and validity of our research findings regarding fault localization, we have leveraged three widely used Java fault datasets: **Bears** [42], **Bugs.jar** [57], and **Defects4J** [25]. Since we focus on function-level, we perform two filtering steps on the original datasets to obtain functions, and the filtering results of each dataset are displayed in Table 2. Specifically, the filtered dataset contains a total of 40,518 functions, comprising 3,214 buggy functions and 37,304 non-buggy functions. This yields a buggy to non-buggy function ratio of 1:11.6.

Step-1: Each commit is considered as a version of a project. We use the commit IDs to request commit histories, and for each commit, we extract the

Table 2: Statistic of the fault datasets

Datasets	# Buggy	# Non-Buggy	# Total	Buggy:Non-Buggy
Bears	132	1,637	1,769	1:12.4
Bugs.jar	1,953	18,995	20,948	1:9.7
Defects4J	1,129	16,674	17,803	1:14.8
Total	3,214	37,304	40,518	1:11.6

code changes before and after fixing a bug. Finally, we use the code change information to obtain the buggy and fixed version of a function. Thus, we collect the following information for a project: buggy functions with their fixes and non-buggy functions. In this step, we obtain the Bears dataset, consisting of 2,009 functions, the Bugs.jar dataset, consisting of 40,880 functions, and the Defects4J dataset, containing 31,423 functions.

Step-2: To clean and normalize the dataset, we start by removing duplicate functions. The three datasets are derived from various versions of projects (e.g., Defects4J extracted from 17 real-world Java projects), leading to a substantial number of duplicate functions extracted from different commits of the same project during step-1. In this step, we finally obtain the Bears dataset, which comprises 1,769 functions, the Bugs.jar dataset, which comprises 20,948 functions, and the Defects4J dataset, which comprises 17,803 functions.

Regarding fault generation, Defects4J includes test suites for evaluating generated functions to determine if they are buggy functions. In contrast, Bears and Bugs.jar do not include readily executable test suites. Therefore, we instruct LLMs to automatically inject faults in the non-buggy functions of Defects4J.

4.1.2. Clone Datasets

We consider two datasets for the evaluation of clone detection: CodeXGLUE-POJ104 [39, 46] and CodeNet-Java250 [54]. CodeXGLUE-POJ104 contains 104 programming challenges, and each has 500 C/C++ solutions from programmers. Code-XGLUE [39] reconstruct it as a public benchmark by splitting the dataset into Training (64 challenges), Validation (16 challenges), and Testing (24 challenges) sets, making sure that there are no overlapped challenges between any two sets. CodeNet-Java250 contains 250 Java programming

challenges from online judge websites, and each has 300 solutions from programmers. It splits the datasets into Training (125 challenges), Validation (62 challenges), and Testing (63 challenges) sets without overlapped challenges. The detailed statistics of these two datasets can be found in Table 3.

To generate additional clone data, we utilize the official prompts provided by HumanEval-X [84]. HumanEval-X is a benchmark for evaluating the multilingual ability of code generative models. It consists of 820 high-quality human-crafted data samples (each with test cases) in Python, C++, Java, JavaScript, and Go, and can be used for various tasks, such as code generation and translation.

Table 3: Statistic of the clone datasets

Datasets	**Language**	**# Training**	**# Validation**	**# Testing**
CodeXGLUE-POJ104	C/C++	32,000	8,000	12,000
CodeNet-Java250	Java	37,500	18,600	18,900

4.2. Evaluation Metrics

To evaluate the effectiveness of fault localization, we consider the following metrics: Accuracy, Precision, Recall, F1-score, and FPR.

Accuracy evaluates the performance of how many code lines can be correctly labeled. It is calculated as: $\frac{TP+TN}{TP+FP+TN+FN}$.

Precision is the fraction of true buggy lines among the located ones. It is defined as: $\frac{TP}{TP+FP}$.

Recall measures how many buggy lines can be correctly located. It is defined as: $\frac{TP}{TP+FN}$.

F1-score is a harmonic mean of *Precision* and *Recall* and can be calculated as: $\frac{2\times P\times R}{P+R}$.

FPR refers to the proportion of non-buggy ones that are predicted to be buggy. It is defined as: $\frac{FP}{FP+TN}$.

As for clone detection, we adopt the MAP@R metric [39, 54]. MAP@R is a common metric to evaluate the quality of information retrieval, and it measures the average precision scores of a set of the top-R clone candidates presented in response to a query program.

4.3. Implementation

We develop the fault generation and clone generation pipeline in Python, utilizing PyTorch [52] implementations of LLMs (i.e., Magicoder 6.7B, DeepSeek-

Coder 6.7B, OpenCodeInterpreter 6.7B, CodeLlama 7B, WizardCoder 7B, Phi-2 2.7B, Mistral 7B, and Llama 3 8B). We use the Hugging Face API [21] to load the model weights and generate outputs. We also adhere to the best-practice guide [58] for each prompt. Our default setting for generation uses top p = 0.95, temperature = 1. We use 10 samples for fault generation and 200 samples for clone generation. Regarding pre-trained LMs, we utilize their publicly available source code and perform fine-tuning with the default parameters provided in their original code. All these models are implemented using the PyTorch [52] framework. During fine-tuning, we employ the AdamW optimizer [37], which is widely adopted to fine-tune Transformer-based models to optimize the parameters of LMs and LLMs. In addition to standard fine-tuning, we also employ Efficient Fine-Tuning with LoRA [20] for LLMs. For this, we set the rank to 8, alpha to 16, and dropout to 0.1. In our experiment, we set the maximum epochs of fault localization and clone detection to 20 and adopt the early stop mechanism to obtain better parameters. The models (i.e., LMs and LLMs) with the best performance on the validation set are used for the evaluations. Our evaluation is conducted on a 32-core workstation equipped with an Intel(R) Xeon(R) Platinum 8358P CPU @ 2.60GHz, 2TB RAM, and 8×NVIDIA A800 80GB GPU, running Ubuntu 20.04.6 LTS.

5. Experimental Results

- **RQ-1 Effectiveness of LLMs in Fault Generation and Clone Generation.** *Can LLMs successfully inject faults into non-buggy functions and generate code clones?*
- **RQ-2 Comparable Study of LLMs and LMs.** *How does the performance of LLMs in fault localization and clone detection compare against LMs?*
- **RQ-3 Enhancing LMs with LLM-generated Data.** *How well do LMs perform at two tasks when learning from LLM-generated faults and clones?*

5.1. RQ-1: Can LLMs Successfully Inject Faults into Non-Buggy Functions and Generate Code Clones?

Objective. Recently, the exceptional performance of LLMs in code generation has garnered significant attention within the software engineering community. To overcome the challenge posed by the current scarcity and high cost of collecting real-world data, we leverage the powerful generation

Table 4: Statistics of the generated faults (RQ-1)

Models	# Test Fail	# Test Pass	# Time Out	# Other
Magicoder	22,809	56,850	1,565	23,747
DeepSeek-Coder	23,136	51,072	1,468	35,698
OpenCodeInterpreter	18,768	46,259	1,720	19,196
CodeLlama	22,442	50,491	1,617	23,852
WizardCoder	16,340	42,899	1,650	36,273
Phi-2	11,256	54,727	2,256	30,289
Mistral	21,511	40,225	3,111	37,904
Llama 3	24,718	39,983	2,443	29,304
All (Filtered)	**112,709**	**-**	**-**	**-**

capabilities of LLMs to enhance pre-trained LMs. Specifically, we utilize LLMs to generate domain-specific data, thereby enhancing the performance of pre-trained models on specific tasks. Therefore, in this RQ, we aim to investigate and analyze the characteristics of faults and clones generated by LLMs in software code.

Experimental Setup. To answer this research question, we investigate the characteristics of the faults using the metrics listed below. We compute these metrics for all LLMs, as well as the average of these metrics for each project. Crucially, before computing these metrics, we remove all comments from the functions.

- **Test fail**. This metric measures the number of confirmed faults (i.e., functions that fail to pass the test cases) generated by LLMs. Specifically, a test failure indicates that the compilation was successful and the syntax was correct, yet certain test cases still failed to pass. We follow the methodologies [22, 59, 60] by using test cases to measure faults. If a test case yields different outputs when executed on the original program and a program with a mutation fault, the mutation fault is categorized as "killed" by that test case, indicating the presence of a fault.

- **Test pass**. This metric measures the number of functions that pass the test cases. Specifically, for fault generation, this metric indicates instances where the LLMs may fail to inject faults or the issue is not necessarily covered by the existing tests. In contrast, for clone generation, it represents instances where the generated codes are correct.

- **Time out**. Verifying hundreds of thousands of generated functions is time-consuming. To address this, we follow prior works [77, 70] and set the timeout threshold to 180 seconds. This metric specifically measures the number of functions that exceed this threshold.

- **Other**. This metric measures the number of functions generated by LLMs that result in compilation failures or syntax errors.

- **Lines involved in fault generation**. These metrics measure the average number of lines added, removed, or modified to generate the faults. We calculate the number of changed lines using the diff tool.

- **Edit distance**. This metric measures the minimum number of single-character edits (i.e., insertions, deletions, or substitutions) required to change one string into another. Each code is represented as a string of characters to compute the Levenshtein edit distance [30].

For clone generation, we use the unbiased pass@k metric proposed in Codex [6, 84]. To evaluate pass@k, we generate $n \geq k$ samples per task and in this paper, we use $n = 200$ and $k \in \{1, 10, 100, 200\}$.

Results. Table 4 and Table 5 show the number of faults generated by LLMs and the characteristics of faults, while Table 6 shows the characteristics of clones generated by LLMs. Overall, **we find that LLMs possess the capability to generate faults and clones, enabling the creation of a vast array of non-duplicate data.** We discuss the results from the characteristics of faults and clones, respectively.

Characteristics of Faults. As shown in Table 4, we can draw the following observations for fault generation: (1) All the LLMs can successfully generate faults for the non-buggy function in the subject projects. Llama 3 generates 1,582∼13,462 more faults compared to other LLMs. **After filtering out duplicate functions (48,271 duplications were removed), we ended up with 112,709 buggy functions.** (2) During the fault generation process, LLMs often produce some invalid data. Specifically, the functions where LLMs fail to inject faults (i.e., # Test Pass) and the functions that result in compilation failures or syntax errors (i.e., # Other) account for a significant portion.

As shown in Table 5, we find that: **(1) The generated faults mainly come from JacksonDatabind, Lang, and Math projects.** These three projects account for 46.6% of all the generated faults. (2) The average

Table 5: Characteristics of the generated faults (RQ-1)

Projects	# Test Fail	Add	Remove	Modify	Edit Distance
Chart	6,815	0.82	1.85	0.90	66.55
Cli	3,900	1.05	1.78	1.08	61.87
Closure	5,061	1.57	1.32	1.19	62.10
Codec	2,824	0.83	0.87	1.06	47.24
Collections	1,314	0.88	1.51	0.95	56.30
Compress	7,426	0.89	1.35	1.01	60.96
Csv	2,019	0.80	0.78	0.98	50.10
Gson	577	1.62	0.82	1.04	52.47
JacksonCore	7,446	0.85	1.57	1.07	58.90
JacksonDatabind	12,927	0.68	1.28	1.02	60.56
JackonXml	778	0.83	2.07	0.81	75.05
Jsoup	9,845	0.77	0.71	1.03	43.48
JxPath	2,859	1.06	1.70	1.35	68.23
Lang	19,726	1.02	0.99	1.11	51.47
Math	19,912	0.66	0.84	1.11	42.67
Mockito	2,978	0.48	1.44	0.95	74.56
Time	6,302	0.58	0.91	0.98	49.84

number of lines added, removed, or modified to generate the faults ranges from 0.48 to 2.07. (3) The edit distance between different projects ranges from 42.67 to 75.05. Compared to other projects, JacksonXml's faults have higher edit distance values. This indicates that the changes made by LLMs in the JacksonXml project differ more significantly from the original code.

Characteristics of Clones. Table 6 presents the characteristics of the generated clones. In general, **we find that the code generation capabilities of LLMs are stronger for Java than for C++.** For example, LLMs achieve a higher average pass@200 score in HumanEval-X-Java compared to HumanEval-X-C++ (i.e., 55.79% v.s. 41.08%). Furthermore, after filtering out duplicate data, LLMs generated 5,000 clones in HumanEval-X-C++ and 10,546 clones in HumanEval-X-Java.

Computational Cost of Generation. Table 7 presents the details of the time cost and GPU memory cost for fault generation and clone generation. We use 10 samples for fault generation and 200 samples for clone generation. Consequently, an LLM is required to generate 166,740 data points for fault generation and 65,600 data points for clone generation. This process is highly time-consuming. However, since the LLMs can be executed in parallel, we

Table 6: Characteristics of the generated clones (RQ-1)

Models	HumanEval-X-C++					HumanEval-X-Java				
	pass@1	pass@10	pass@100	pass@200	# Test Pass	pass@1	pass@10	pass@100	pass@200	# Test Pass
Magicoder	11.62	44.77	67.57	70.12	3,812	18.50	50.50	66.43	68.90	6,067
DeepSeek-Coder	1.36	10.98	45.01	57.93	446	2.77	20.98	55.12	60.37	910
OpenCodeInterpreter	1.23	10.28	42.01	51.83	403	4.99	30.76	60.45	64.63	1,638
CodeLlama	1.68	11.98	36.13	45.73	552	9.73	33.14	53.64	59.15	3,191
WizardCoder	0.36	3.32	19.15	26.83	119	3.01	19.30	45.49	50.00	988
Phi-2	0.25	2.35	14.55	20.73	82	0.57	5.13	24.13	31.10	188
Mistral	0.83	6.18	27.49	34.76	272	5.13	22.60	41.02	46.34	1,684
Llama 3	0.22	2.07	13.71	21.34	72	3.78	25.05	59.64	65.85	1,239
All (Filtered)	-	-	-	-	**5,000**	-	-	-	-	**10,546**

calculate the average time and GPU memory cost for the models to generate a complete dataset in a single pass. As shown in the table, fault generation is considerably more time-consuming than clone generation, requiring approximately 280 hours compared to about 46 hours for clone generation. This significant difference in time is primarily attributed to the much larger number of data points generated for the fault task (166,740 vs. 65,600). In contrast, the GPU memory consumption is relatively similar for both tasks (17,394M for fault generation and 15,554M for clone generation). This is because memory usage is predominantly determined by the size of the loaded LLM itself, rather than the number of generation iterations. These results underscore that while generating large datasets with LLMs is computationally expensive, the time cost scales with the volume and complexity of the generation task, whereas the memory footprint remains relatively stable for a given model.

Table 7: Average time cost and GPU memory cost for fault generation and clone generation (RQ-1)

Models	Fault Generation		Clone Generation	
	Time	**GPU Memory**	**Time**	**GPU Memory**
LLMs	280h 16m 16s	17,394M	46h 09m 11s	15,554M

Finding 1*: LLMs are capable of generating a substantial amount of faults and clones, significantly enhancing the data available for fault localization and clone detection.*

5.2. RQ-2: How does Directly Applying LLMs for Two Tasks Compare Against LMs?

Objective. Benefiting from the powerful representation capability of deep neural networks, many pre-trained LMs have been proposed [14, 16, 17, 1,

64, 63] and they treat the source code in different ways. Meanwhile, recently, LLMs [2, 56, 40, 66, 85, 44, 23, 45] have attracted much attention since their powerful ability can be easily adapted to various types of downstream tasks [3, 51]. However, the efficiency and overhead of LLMs and pre-trained LMs in fault localization and clone detection have not been systematically compared. Considering these issues, we aim to conduct an extensive study to comprehensively compare LLMs and pre-trained LMs.

Experimental Setup. We consider eight LLMs (i.e., Magicoder [66], OpenCodeInterpreter [85], DeepSeek-Coder [2], CodeLlama [56], WizardCoder [40], Llama 3 [44], Mistral [23], and Phi-2 [45]) and eight LMs (i.e., BERT [7], RoBERTa [36], CodeBERT [14], GraphCodeBERT [17], UniXcoder [16], PLBART [1], CodeT5 [64], and CodeT5+ [63]).

Since we evaluate two distinct downstream tasks, the datasets and data processing methods used varied accordingly. For the fault localization task, we utilize all buggy functions from the Defects4J dataset. We divide these functions into training, validation, and testing sets in an 8:1:1 ratio. Additionally, we extend our evaluation by training and validating on the Detects4J dataset, but testing on the Bears and Bugs.jar datasets. This experiment aims to showcase the localization capabilities of different models in locating unknown real-world faults. Regarding the clone detection task, we use datasets divided in previous works [16, 8], including CodeXGLUE-POJ104 and CodeNet-Java250.

For LMs, we use the training set to fine-tune the models. In contrast, for LLMs, we utilize three settings: standard fine-tuning, fine-tuning with LoRA, and few-shot learning. The purpose of the few-shot setting is to evaluate the performance of LLMs under limited computing resources and time constraints. In terms of fine-tuning and fine-tuning with LoRA settings, we adopt the parameters and configurations described in Section 4.3 to fine-tune both LLMs and LMs, and then evaluate them on the testing set. For the few-shot setting, we use the testing set for the evaluation and instruct LLMs with the following task descriptions to tell it to act as a fault locator.

> ***Fault Localization:*** *I will provide you a buggy Java code snippet and please locate buggy lines.*

The clone detection metric MAP@R requires assessing the similarity between a code snippet and all other code snippets in the dataset. However, due to the token limitations of LLMs, we are unable to implement clone

detection in a few-shot setting.

In order to comprehensively compare the performance among LLMs and LMs, we consider six widely used performance measures (i.e., Precision, Recall, F1-score, Accuracy, FPR, and MAP@R).

Table 8: Results for LMs and LLMs in fault localization (RQ-2)

Models	Defects4J					Real-World Datasets				
	F1-score	**Recall**	**Precision**	**Accuracy**	**FPR**	**F1-score**	**Recall**	**Precision**	**Accuracy**	**FPR**
BERT	0.341	0.402	0.297	0.843	0.108	0.201	0.150	0.303	0.841	0.053
RoBERTa	0.394	0.412	0.378	0.876	0.073	0.286	**0.258**	0.322	0.836	0.079
CodeBERT	**0.428**	**0.434**	0.421	0.887	0.065	0.269	0.217	0.353	0.849	0.058
GraphCodeBERT	0.384	0.353	0.421	0.889	0.053	**0.290**	0.241	0.364	0.849	0.062
UniXcoder	0.409	0.396	0.423	0.889	0.058	0.220	0.162	0.343	0.856	0.044
PLBART	0.395	0.348	**0.456**	**0.894**	**0.046**	0.244	0.169	**0.443**	**0.865**	**0.032**
CodeT5	0.392	0.367	0.422	0.889	0.055	0.250	0.191	0.363	0.856	0.048
CodeT5+	0.392	0.360	0.431	0.891	0.052	0.241	0.177	0.395	0.863	0.038
Fine-Tuning										
Magicoder	0.359	0.331	0.391	0.885	0.056	0.236	0.182	0.337	0.850	0.052
CodeLlama	0.429	0.331	0.608	0.914	0.023	0.208	0.132	**0.491**	0.872	0.020
WizardCoder	0.404	0.331	0.517	0.904	0.034	0.217	0.148	0.409	0.864	0.031
DeepSeek-Coder	0.376	0.324	0.449	0.895	0.043	0.233	0.168	0.383	0.860	0.039
OpenCodeInterpreter	0.393	0.309	0.539	0.907	0.029	0.202	0.134	0.414	0.866	0.028
Phi-2	0.358	**0.456**	0.295	0.841	0.117	**0.264**	**0.299**	0.237	0.788	0.141
Mistral	0.310	0.309	0.311	0.866	0.074	0.162	0.126	0.225	0.834	0.063
Llama 3	**0.453**	0.355	**0.625**	**0.919**	**0.022**	0.164	0.098	0.489	**0.878**	**0.014**
Fine-Tuning (LoRA)										
Magicoder	0.432	0.353	0.558	0.910	0.030	0.232	0.193	0.290	0.856	0.060
CodeLlama	0.415	0.324	0.579	0.911	0.026	0.197	0.145	0.307	0.867	0.042
WizardCoder	0.402	0.316	0.551	0.908	0.028	0.217	0.156	**0.354**	**0.873**	0.036
DeepSeek-Coder	0.449	0.390	0.530	0.907	0.037	0.265	0.258	0.273	0.839	0.087
OpenCodeInterpreter	0.391	0.331	0.479	0.900	0.039	0.163	0.135	0.206	0.844	0.066
Phi-2	0.420	**0.397**	0.446	0.894	0.053	**0.273**	**0.280**	0.266	0.832	0.098
Mistral	0.323	0.294	0.357	0.880	0.057	0.161	0.130	0.210	0.847	0.062
Llama 3	**0.451**	0.340	**0.667**	**0.922**	**0.018**	0.157	0.108	0.291	0.870	**0.033**
Few-Shot										
Magicoder	0.151	0.425	0.092	0.617	0.366	0.197	0.425	0.128	0.603	0.374
CodeLlama	**0.177**	**0.603**	0.104	0.547	0.458	**0.220**	**0.558**	0.137	0.547	0.455
WizardCoder	0.170	0.582	0.100	0.541	0.463	0.218	0.528	0.137	0.567	0.429
DeepSeek-Coder	0.171	0.452	0.105	0.647	0.336	0.195	0.389	0.130	0.633	0.335
OpenCodeInterpreter	0.169	0.486	0.103	0.616	0.373	0.214	0.483	0.137	0.593	0.392
Phi-2	0.112	0.096	**0.133**	**0.877**	**0.055**	0.108	0.083	0.153	**0.843**	**0.059**
Mistral	0.098	0.096	0.100	0.857	0.076	0.140	0.111	**0.188**	**0.843**	0.062
Llama 3	0.158	0.280	0.110	0.762	0.196	0.173	0.251	0.132	0.728	0.211

Results. Table 8 and Table 9 show the results of fault localization and clone detection, while Table 10 shows the average time cost and GPU memory cost for fine-tuning different types of models for one epoch. Overall, **we find that fine-tuned LLMs exhibit slightly superior performance in fault localization compared to LMs. However, this comes at a significant cost in terms of time and computational resources.** We discuss the results from the aspects of effectiveness and efficiency, respectively.

Comparison of Effectiveness. As shown in Table 8, we can draw the following observations for fault localization: (1) There is no significant differ-

Table 9: Results for LMs and LLMs in clone detection (RQ-2)

Models	**CodeXGLUE-POJ104**	**CodeNet-Java250**
BERT	78.65%	73.62%
RoBERTa	81.65%	77.13%
CodeBERT	84.93%	80.98%
GraphCodeBERT	83.55%	83.09%
UniXcoder	90.50%	83.45%
PLBART	85.34%	82.79%
CodeT5	90.46%	**85.58%**
CodeT5+	**90.52%**	82.35%
Fine-Tuning/LoRA		
Magicoder	54.62%/65.35%	48.06%/59.51%
CodeLlama	56.52%/69.69%	48.42%/56.60%
WizardCoder	56.49%/61.47%	48.38%/53.30%
DeepSeek-Coder	54.68%/74.95%	48.06%/64.70%
OpenCodeInterpreter	56.49%/72.19%	48.04%/64.56%
Phi-2	**59.97%/78.59%**	**51.13%/65.67%**
Mistral	54.98%/65.14%	48.12%/56.82%
Llama 3	58.38%/67.26%	50.65%/61.11%
Without Fine-Tuning		
Magicoder	34.33%	21.99%
CodeLlama	33.97%	23.95%
WizardCoder	35.62%	22.75%
DeepSeek-Coder	34.34%	22.32%
OpenCodeInterpreter	**36.10%**	25.37%
Phi-2	32.88%	**27.36%**
Mistral	31.53%	20.42%
Llama 3	34.67%	21.65%

ence between encoder models and encoder-decoder models. After fine-tuning, all LMs show promising performance. Among them, PLBART exhibits the best performance in terms of Precision, Accuracy, and FPR on both the Defects4J and real-world datasets. **(2) Under the fine-tuning and fine-tuning with LoRA settings, although the best performance of LLMs surpasses that of LMs, the difference was not significant.** For example, on Defects4J, the F1-score increases from 0.428 to 0.453, the Recall increases from 0.434 to 0.456, the Precision increases from 0.456 to 0.625, the Accuracy increases from 0.894 to 0.919, and the FPR decreases from 0.046 to 0.022. (3) Under the few-shot setting, LLMs show very poor fault localization capability, with much lower precision and F1-score compared to LLMs and LMs under fine-tuning settings. This indicates the necessity of fine-tuning for specific datasets in specific task scenarios, as LLMs cannot be universally applied to any scenario. (4) Regardless of whether they are LMs or LLMs, their performance on the real-world datasets after being fine-tuned on the Defects4J dataset is very poor, even comparable to the few-shot setting. This highlights the importance of domain-specific data and indicates that the current limited datasets are insufficient for fine-tuning models effectively.

As shown in Table 9, **we find that LMs outperform LLMs significantly in clone detection.** For instance, CodeT5+ achieves a MAP@R of 90.52% on the POJ104 dataset and CodeT5 achieves a MAP@R of 85.58% on the Java250 dataset. In contrast, the highest-performing LLM, Phi-2 (in fine-tuning with LoRA setting), only achieves a MAP@R of 78.59% and 65.67% on POJ104 and Java250, respectively.

Comparison of Efficiency. Table 10 presents the details of the time cost and GPU memory cost for different models in our study. **We find that fine-tuning LLMs requires significantly more time and computational resources.** On the whole, fine-tuning encoder-only LMs and encoder-decoder LMs requires only about 35.2% to 81.3% of the time cost needed to fine-tune LLMs. Additionally, the GPU memory cost of LMs is much lower than that of LLMs. This is mainly because LMs typically have 110M to 220M parameters, while LLMs have 2.7B to 8B parameters. It's worth noting that even with fine-tuning using LoRA, where the trainable parameters for LLMs range from 12.69% to 44.36%, LLMs still require more time and computational resources than LMs. For instance, in the fault localization task, an epoch of training takes 2 minutes and 13 seconds and occupies 16,627 MB of GPU memory. In the clone detection task, an epoch of training takes 53 minutes and 35 seconds and uses 45,004 MB of GPU memory. These results show no

advantage compared to fine-tuning LMs.

Table 10: Average time cost and GPU memory cost for fine-tuning different models for one epoch (RQ-2)

Models	Fault Localization		Clone Detection	
	Time	**GPU Memory**	**Time**	**GPU Memory**
LMs	1m 14s	3,189M	46m 53s	11,551M
LLMs	3m 30s	64,031M	57m 40s	53,253M
LLMs (LoRA)	2m 13s	16,627M	53m 35s	45,004M

"Trainable Params": In LoRA fine-tuning, the trainable parameters of LLMs range from 12.69% to 44.36%.

***Finding 2**: Investing significant time and computational resources in fine-tuning LLMs, only to achieve minor performance improvements compared to LMs (and sometimes even inferior performance, as observed in clone detection), is not considered worthwhile.*

5.3. RQ-3: Does LLM-generated Data Improve the Performance of LMs?

Objective. In RQ2, we find that fine-tuning LLMs requires more time and computational resources but does not necessarily yield outstanding performance improvements. Furthermore, current LMs are constrained by the lack of domain-specific data, and collecting domain data is costly. By pre-training on large amounts of open-source code snippets, LLMs have the ability to generate code directly based on the surrounding context. Therefore, in this RQ, we aim to investigate whether LLMs can transfer their learned knowledge to LMs. Specifically, we explore if LLMs can generate data (e.g., fault data and clone data) for specific tasks to enhance the performance of LMs on downstream tasks.

Experimental Setup. To evaluate the effectiveness of using the LLM-generated data to improve the LMs. We adopt the experiment settings described in Section 4.3 and perform the evaluation on eight LMs (i.e., BERT [7], RoBERTa [36], CodeBERT [14], GraphCodeBERT [17], UniXcoder [16], PLBART [1], CodeT5 [64], and CodeT5+ [63]). In terms of fault localization, we use the score to rank the generated data (refer to Section 3.5 for more detail) and select 10%, 20%, 50%, and 100% to add to the training set of Defects4J. We also extend our evaluation by testing on the Bears and Bugs.jar datasets, which are considered to be unknown real-world datasets

(no intersection with the LLM-generated faults). For clone detection, we first calculate the average edit distance for all clone instances under each generation task. Then, we compute the edit distance between each clone instance and other clone instances. We select all clone instances with an edit distance greater than the average edit distance and add them to the training set of CodeXGLUE-POJ104 and CodeNet-Java250. Additionally, in order to comprehensively evaluate whether the new training set enhances the performance of the LMs, we use the same metrics as RQ-2.

Results. Table 11, Table 12, and Fig. 3 illustrate the performance improvements of LMs in fault localization after the incorporation of data generated by LLMs, while Table 13 shows the enhancements of these LMs in the area of code clone detection. Overall, **we find that LLM-generated data can significantly enhance the performance of LMs in both fault localization and code clone detection tasks.** We discuss the results from the aspects of fault localization and clone detection, respectively.

Table 11: Increase of F1-score, Recall, Precision, and Accuracy in fault localization (RQ-3)

Models	F1-score					Recall				
	Baseline	10%×Generated	30%×Generated	50%×Generated	All Generated	Baseline	10%×Generated	30%×Generated	50%×Generated	All Generated
BERT	0.341	**0.354 (↑3.81%)**	**0.400 (↑17.30%)**	**0.344 (↑0.88%)**	**0.357 (↑4.69%)**	0.402	0.331 (↓17.66%)	0.394 (↓1.99%)	0.307 (↓23.63%)	0.315 (↓21.64%)
RoBERTa	0.394	**0.406 (↑3.05%)**	**0.434 (↑10.15%)**	**0.432 (↑9.64%)**	0.382 (↓3.05%)	0.412	0.404 (↓1.94%)	0.397 (↓3.64%)	**0.434 (↑5.34%)**	0.309 (↓25.00%)
CodeBERT	0.428	**0.443 (↑3.50%)**	**0.478 (↑11.68%)**	**0.461 (↑7.71%)**	0.428 (∼0.00%)	0.434	**0.559 (↑28.80%)**	0.397 (↓8.53%)	**0.500 (↑15.21%)**	0.360 (↓17.05%)
GraphCodeBERT	0.384	**0.429 (↑11.72%)**	**0.550 (↑43.23%)**	**0.524 (↑36.46%)**	**0.454 (↑18.23%)**	0.353	**0.397 (↑12.46%)**	**0.522 (↑47.88%)**	**0.559 (↑58.36%)**	**0.382 (↑8.22%)**
UniXcoder	0.409	**0.486 (↑18.83%)**	**0.457 (↑11.74%)**	**0.517 (↑26.41%)**	**0.460 (↑12.47%)**	0.396	**0.511 (↑29.04%)**	**0.417 (↑5.30%)**	**0.504 (↑27.27%)**	**0.453 (↑14.39%)**
PLBART	0.395	**0.436 (↑10.38%)**	**0.460 (↑16.46%)**	**0.457 (↑15.70%)**	**0.441 (↑11.65%)**	0.348	**0.393 (↑12.93%)**	**0.400 (↑14.94%)**	**0.393 (↑12.93%)**	**0.415 (↑19.25%)**
CodeT5	0.392	**0.475 (↑21.17%)**	**0.536 (↑36.73%)**	**0.537 (↑36.99%)**	**0.457 (↑16.58%)**	0.367	**0.374 (↑1.91%)**	**0.540 (↑47.14%)**	**0.496 (↑35.15%)**	**0.475 (↑29.43%)**
CodeT5+	0.392	**0.447 (↑14.03%)**	0.387 (↓1.28%)	0.346 (↓11.73%)	**0.396 (↑1.02%)**	0.360	**0.475 (↑31.94%)**	**0.568 (↑57.78%)**	0.237 (↓34.17%)	**0.396 (↑10.00%)**

Models	Precision					Accuracy				
	Baseline	10%×Generated	30%×Generated	50%×Generated	All Generated	Baseline	10%×Generated	30%×Generated	50%×Generated	All Generated
BERT	0.297	**0.382 (↑28.62%)**	**0.407 (↑37.04%)**	**0.390 (↑31.31%)**	**0.412 (↑38.72%)**	0.843	**0.878 (↑4.15%)**	**0.880 (↑4.39%)**	**0.881 (↑4.51%)**	**0.885 (↑4.98%)**
RoBERTa	0.378	**0.407 (↑7.67%)**	**0.478 (↑26.46%)**	**0.431 (↑14.02%)**	**0.500 (↑32.28%)**	0.876	**0.884 (↑0.91%)**	**0.899 (↑2.63%)**	**0.889 (↑1.48%)**	**0.902 (↑2.97%)**
CodeBERT	0.421	0.367 (↓12.83%)	**0.600 (↑42.52%)**	**0.428 (↑1.66%)**	**0.527 (↑25.18%)**	0.887	0.863 (↓2.71%)	**0.915 (↑3.16%)**	0.886 (↓0.11%)	**0.906 (↑2.14%)**
GraphCodeBERT	0.421	**0.466 (↑10.69%)**	**0.582 (↑38.24%)**	**0.494 (↑17.34%)**	**0.559 (↑32.78%)**	0.889	**0.897 (↑0.90%)**	**0.917 (↑3.15%)**	**0.901 (↑1.35%)**	**0.910 (↑2.36%)**
UniXcoder	0.423	**0.464 (↑9.69%)**	**0.504 (↑19.15%)**	**0.530 (↑25.30%)**	**0.467 (↑10.40%)**	0.889	**0.895 (↑0.67%)**	**0.903 (↑1.57%)**	**0.908 (↑2.14%)**	**0.897 (↑0.90%)**
PLBART	0.456	**0.491 (↑7.68%)**	**0.540 (↑18.42%)**	**0.546 (↑19.74%)**	**0.471 (↑3.29%)**	0.894	**0.899 (↑0.56%)**	**0.907 (↑1.45%)**	**0.907 (↑1.45%)**	**0.896 (↑0.22%)**
CodeT5	0.422	**0.650 (↑54.03%)**	**0.532 (↑26.07%)**	**0.585 (↑38.63%)**	**0.440 (↑4.27%)**	0.889	**0.919 (↑3.37%)**	**0.908 (↑2.14%)**	**0.916 (↑3.04%)**	0.889 (∼0.00%)
CodeT5+	0.431	0.423 (↓1.86%)	0.294 (↓31.79%)	**0.635 (↑47.33%)**	0.396 (↓8.12%)	0.891	0.885 (↓0.67%)	0.824 (↓7.52%)	**0.912 (↑2.36%)**	0.882 (↓1.01%)

Improvement of Fault Localization. As shown in Table 11 and Fig. 3, we can draw the following observations:

LLM-generated data brings noticeable improvements to LMs. Overall, the addition of generated data reduces the model's average FPR. Compared to the LMs fine-tuned using only the original training set (i.e., the Baseline column in Table 11), the LMs fine-tuned with both the training set and generated data demonstrate significant improvements in F1-score, Recall, Precision, Accuracy, and FPR. For example, by adding 30% of the generated data to the training set, GraphCodeBERT's F1-score increases from 0.384 to 0.550, Recall increases from 0.353 to 0.522, Precision increases from 0.421 to 0.582, and Accuracy increases from 0.889 to 0.917.

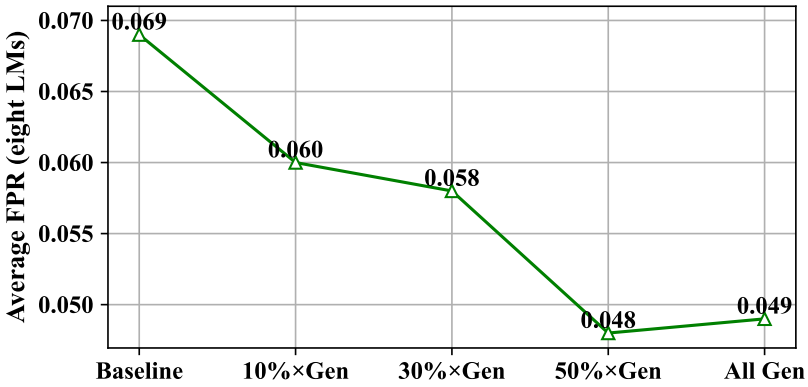

Figure 3: Average decrease (eight LMs) of FPR in fault localization (RQ-3)

Adding 30%-50% of generated data yields the best results for LMs. Generally, the performance of LMs improves with an increase in the amount of data, but it typically peaks when adding 30%-50% of generated data. For example, when adding 10% of generated data to the original training set, the Recall metric of CodeT5 improves by 1.91% (i.e., 0.374 v.s. 0.367). When adding 30% of data, the improvement is 47.14% (i.e., 0.540 v.s. 0.367). However, when add 50% and all of the generated data, the improvement diminishes (i.e., 0.496 v.s. 0.367 and 0.475 v.s. 0.367). This discrepancy arises because the data is sorted, and the quality of the data decreases progressively. Therefore, too much data (i.e., All Generated) can introduce a significant amount of noise, while too little data (i.e., 10%×Generated) is insufficient for the LLMs to learn enough defect patterns.

As shown in Table 12, **we find that LLM-generated data can enhance the fault localization capability of LMs on unknown real-world datasets.** Similar to the results on Defects4J, adding a portion of the generated data yields the best performance. For example, adding 10% of generated data to the original training set can increase the average F1-score of the LMs from 0.250 to 0.281.

Improvement of Clone Detection. As shown in Table 13, **we find that LLM-generated data can also improve the performance of clone detection.** Specifically, after learning from the LLM-generated data, LMs show an improvement of 0.72%∼6.09% on the POJ104 dataset. Similarly, on the Java250 dataset, LMs exhibit an increase of 0.12%∼3.23%. Among all the LMs, CodeT5 and CodeT5+ perform the best. For example, CodeT5+ achieves a MAP@R metric of 91.97% on the POJ104 dataset, and CodeT5

Table 12: Average increase (eight LMs) in fault localization on real-world datasets (RQ-3)

Datasets	F1-score	Recall	Precision	Accuracy
Baseline	0.250	0.196	0.361	0.852
10%×Generated	**0.281 (↑12.49%)**	**0.235 (↑20.08%)**	**0.372 (↑3.20%)**	0.848 (↓0.45%)
30%×Generated	**0.278 (↑10.99%)**	**0.243 (↑24.30%)**	**0.363 (↑0.57%)**	0.840 (↓1.37%)
50%×Generated	**0.263 (↑5.19%)**	**0.211 (↑7.82%)**	**0.371 (↑2.86%)**	0.851 (↓0.10%)
All Generated	**0.254 (↑1.64%)**	**0.197 (↑0.47%)**	**0.369 (↑2.37%)**	**0.853 (↑0.17%)**

achieves a MAP@R metric of 85.68% on the Java250 dataset. In total, the improvements for encoder-only LMs are significantly greater than those for encoder-decoder LMs. Encoder-only LMs achieve enhancements ranging from 0.44% to 6.09%, whereas encoder-decoder LMs only see improvements between 0.12% and 2.21%. We believe this disparity may be attributed to the inherently superior performance of encoder-decoder LMs compared to encoder-only LMs.

Table 13: Increase of MAP@R in clone detection (RQ-3)

Models	CodeXGLUE-POJ104		CodeNet-Java250	
	Baseline	Generated	Baseline	Generated
BERT	78.65%	**83.44% (↑6.09%)**	73.62%	**75.19% (↑2.13%)**
RoBERTa	81.65%	**84.18% (↑3.10%)**	77.13%	**79.62% (↑3.23%)**
CodeBERT	84.93%	**87.24% (↑2.72%)**	80.98%	**81.89% (↑1.12%)**
GraphCodeBERT	83.55%	**87.37% (↑4.57%)**	83.09%	**83.46% (↑0.44%)**
UniXcoder	90.50%	**91.76% (↑1.39%)**	83.45%	**85.37% (↑2.30%)**
PLBART	85.34%	**87.11% (↑2.07%)**	82.79%	**84.62% (↑2.21%)**
CodeT5	90.46%	**91.11% (↑0.72%)**	85.58%	**85.68% (↑0.12%)**
CodeT5+	90.52%	**91.97% (↑1.60%)**	82.35%	**83.88% (↑1.86%)**

***Finding 3**: Incorporating LLM-generated data into the original training set leads to substantial performance improvements for LMs across both fault localization and code clone detection tasks.*

6. Discussion

6.1. Why "Student" Surpasses "Teacher"

In our results for RQ-2 and RQ-3, we observe that the LLMs used for data generation are outperformed by the LMs fine-tuned on that LLM-generated data. We attribute this phenomenon to the following three primary reasons:

- **Low-Quality Data Filtering.** Following the initial data generation by the LLMs, we apply a filtering process to remove low-quality data. For fault localization, we use the Defects4J test suite to evaluate the generated faults. A mutation fault is categorized as "killed" by a test case, thus confirming its validity, if that test case yields a different output when executed on the mutated program compared to the original. For clone generation, we retain only the generated functions that successfully pass the entire test suite. After further filtering to remove duplicate data, this process yielded 5,000 code clones for HumanEval-X-C++ and 10,546 for HumanEval-X-Java.

- **High-Quality Data Selection.** In addition to filtering, we perform a more rigorous selection of high-quality data points. For fault localization, we compute LC, ED, and SS for each generated data and calculate a score using the formula: $Score = |(LC - LC_{ave})/LC_{ave}| + |(ED - ED_{ave})/ED_{ave}| + |(SS - SS_{ave})/SS_{ave}|$. For clone detection, we calculate the average edit distance among all generated code clones for each task in HumanEval-X. Additionally, we compute the average relative edit distance for each code with other code clones, selecting those with an average relative edit distance greater than or equal to the overall average edit distance. Based on these methods, we created data subsets corresponding to the top 10%, 30%, 50%, and 100% of the selected LLM-generated data. The experimental results are presented in Table 11, Table 12, Table 13, and Fig. 3.

- **Differentiated Strengths of LLMs and LMs.** While LLMs may not excel at non-generative tasks like vulnerability localization and clone detection, this doesn't diminish their robust data generation capabilities. On the contrary, LLMs truly shine in generative tasks. Their remarkable performance in areas such as code generation [11, 86] and program repair [77, 70] underscores their effectiveness in generating code clones and injecting defects. Conversely, pre-trained LMs inherently possess strong capabilities in non-generative tasks. They have demonstrated impressive results in fault localization [47, 75] and code clone detection [16, 8]. Therefore, by carefully filtering low-quality and selecting high-quality data generated by LLMs, LMs can learn from this refined dataset, ultimately enhancing their performance.

6.2. Data Selection Analysis

We utilize LLMs to generate a significant amount of candidate data for fault and clone generation. However, the quality and distribution of this

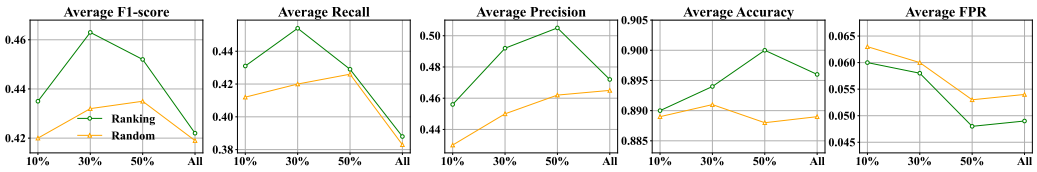

Figure 4: Average result (eight LMs) in fault localization when using ranking or random selection

data vary, so we have devised selection strategies to choose high-quality data. In this section, we discuss the role of these selection strategies. For fault localization, we compare our designed ranking selection strategy with random selection. In the random selection strategy, for each non-buggy function, we randomly select one buggy function from the corresponding buggy functions to pair with it and randomly choose 10%, 30%, and 50% of the data as needed. For clone detection, we compare the LMs fine-tuned on the selected generated data with those fine-tuned on the original generated data. Fig. 4 presents the average results (eight LMs) in fault localization when using ranking or random selection. We see that, compared to random selection, ranking selection shows better performance across five metrics. This demonstrates that our designed score (refer to Section 3 for more detail) can be an effective measure to rank potentially high-quality data. Table 14 shows the average results (eight LMs) in clone detection when adding all generated data or selected data. On the whole, the selected data performs better than the unselected data, improving by 0.78%∼2.71% compared to the baseline. Overall, our selection strategies showcase their efficacy in enhancing both fault localization and clone detection tasks, underscoring the importance of thoughtful data processing in maximizing model performance.

Table 14: Average result (eight LMs) in clone detection when adding all generated data or selection data

Datasets	**CodeXGLUE-POJ104**	**CodeNet-Java250**
Baseline	85.70%	81.82%
Generated (All)	**86.89% (↑1.39%)**	**82.15% (↑0.40%)**
Generated (Selection)	**88.02% (↑2.71%)**	**82.46% (↑0.78%)**
Generated (CoT)	**88.41% (↑3.16%)**	**82.97% (↑1.41%)**
CloneGen (Overlap)	**86.18% (↑0.56%)**	-
Generated (Overlap)	**86.99% (↑1.51%)**	-

6.3. Compare with Traditional Approaches and CoT Prompting

Compare with Traditional Approaches. Traditional generation tools can also be used to generate data. In this discussion, we aim to compare the quality of the data generated by LLMs with that of data generated by generation tools.

For fault generation, we use the Major [24] mutation tool to generate faults. Major is an efficient and flexible mutation analysis framework that supports generating mutants during compilation and exporting source-code mutants. Major supports eight mutation operators: AOR, LOR, SOR, COR, ROR, ORU, LVR, and STD. We perform mutation on all fixed versions of the programs in Defects4J and then extract the mutants (i.e., the buggy functions) that are killed by the test cases. We use two methods to select buggy functions from the mutated results. The first method employs our proposed selection strategy to choose the optimal data, while the second method involves randomly selecting one buggy function from all mutated candidate buggy functions to form a data pair with the corresponding non-buggy function. To ensure a fair comparison, we select the intersection of LLM-generated data and Major-generated data, retaining only the pairs of non-buggy and buggy functions where the non-buggy function appears in both data. This process resulted in 3,619 pairs of non-buggy and buggy functions. From the results in Table 15, we find that LLM-generated data generally outperforms mutated data. Although mutated data is better at improving the Precision of LMs, it leads to a significant drop in F1-score and Recall. This might be due to two reasons: (1) all mutation operations are implemented on a single line, whereas LLMs can insert faults across multiple lines; (2) the faults generated by mutation follow patterns, resulting in less diverse defects.

For clone generation, we use the CloneGen [81] approach to generate code clones. CloneGen applies 15 atomic transformation operators to transform the C/C++ code, generating clones while preserving the original code's semantics. We transform the ground truth for each task in HumanEval-X-C++ and perform multiple transformations; for instance, a clone code generated by the first transformation operator can be further transformed by a second operator to create additional clones.

To ensure a fair comparison, we ensure that each task had the same number of clones in both CloneGen-generated data and LLM-generated data. Table 14 shows the performance improvement of LMs using clones generated by the CloneGen method and those generated by LLMs. We find that with

the same number of clones, LLM-generated data has a better effect, improving LM performance by an average of 1.51%. In contrast, CloneGen-generated data only improved performance by 0.56%.

Compare with CoT Prompting. Specifically, as LLMs have evolved into advanced reasoning models, we believe it is crucial to further evaluate Chain-of-Thought (CoT) data derived from their reasoning processes. Therefore, we also use CoT prompting to generate data, which ultimately yielded 104,372 faults, 4,332 clones in HumanEval-X-C++, and 9,876 clones in HumanEval-X-Java. We similarly employ our selection strategy to select high-quality data from the CoT-generated data. From the results in Table 15 and Table 14, we find that LLMs with CoT prompting can generate higher quality data, thereby improving the performance of LMs trained on this CoT-generated data. The performance improvement is even more significant compared to using standard LLM-generated data.

Table 15: Average result (eight LMs) in fault localization when comparing with the mutation tool

Datasets	F1-score	Recall	Precision	Accuracy
Baseline	0.392	0.384	0.406	0.882
Mutated	0.366 (↓6.63%)	0.276 (↓28.13%)	**0.555 (↑36.70%)**	**0.906 (↑2.72%)**
Mutated*	0.360 (↓8.16%)	0.291 (↓24.22%)	**0.489 (↑20.44%)**	**0.896 (↑1.59%)**
Generated	**0.438 (↑11.73%)**	**0.391 (↑1.82%)**	**0.508 (↑25.12%)**	**0.902 (↑2.27%)**
Generated (CoT)	**0.452 (↑15.31%)**	**0.393 (↑2.34%)**	**0.515 (↑26.85%)**	**0.897 (↑1.70%)**

"Mutated*": Randomly select from Major-generated data.

6.4. Threats to Validity

Internal Validity. The first one is the design of the prompt to instruct LLMs to generate responses. Our prompt design is based on practical advice [58, 50, 31, 56, 75], which has been validated by numerous users online and has shown to elicit good responses from LLMs. The second one is about the potential mistakes in the implementation of studied LMs and LLMs. To mitigate such threats, we use the original source code provided by the corresponding authors, ensuring a direct and reliable foundation for our analysis.

External Validity. One potential threat to external validity is the generalizability of our findings across different datasets, programming languages, and model architectures. Our study involves datasets for both Java and C++, which are widely used in previous research on fault localization and clone detection [72, 16, 63]. We utilize a diverse array of pre-trained LMs, including

both encoder-only and encoder-decoder architectures, with variations in their parameter sizes and classification capabilities. Moreover, our study includes eight distinct LLMs, comprising both code LLMs and general LLMs, each offering unique attributes due to differences in their parameter sizes and generation capabilities. Despite the breadth of our experimentation, it remains uncertain whether our findings can be broadly generalized to other contexts. Thus, further investigation is necessary to validate and extend the conclusions drawn from our research.

7. Conclusion

In this paper, we leverage the powerful generation capabilities of LLMs to enhance pre-trained LMs. Specifically, we employ LLMs to produce domain-specific data, thereby bolstering the efficacy of pre-trained LMs on the target tasks. We conduct experiments by combining different LLMs in our generation phase and introducing various LMs to learn from the LLM-generated data. Then, we compare the performance of these LMs before and after learning the data. We find that LLM-generated data significantly enhances the performance of LMs. The improvement can reach up to 58.36% for fault localization and up to 6.09% for clone detection. This investigation underscores the considerable potential of leveraging LLMs for data generation to yield significant performance gains in LMs.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant No.62202419 and No. 62172214), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Zhejiang Provincial Natural Science Foundation of China (No. LY24F020008), the Ningbo Natural Science Foundation (No. 2022J184), the Key Research and Development Program of Zhejiang Province (No.2021C01105), and the State Street Zhejiang University Technology Center.

References

[1] Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W., 2021. Unified pre-training for program understanding and generation. arXiv preprint arXiv:2103.06333 .

[2] AI, D., 2023. Deepseek coder: Let the code write itself. https://github.com/deepseek-ai/DeepSeek-Coder.

[3] Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., et al., 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. arXiv preprint arXiv:2302.04023 .

[4] Briand, L.C., Labiche, Y., Liu, X., 2007. Using machine learning to support debugging with tarantula, in: The 18th IEEE International Symposium on Software Reliability (ISSRE'07), IEEE. pp. 137–146.

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Advances in neural information processing systems 33, 1877–1901.

[6] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al., 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 .

[7] Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 .

[8] Ding, Y., Chakraborty, S., Buratti, L., Pujar, S., Morari, A., Kaiser, G., Ray, B., 2023. Concord: Clone-aware contrastive learning for source code, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 26–38.

[9] Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., Lou, Y., 2024a. Evaluating large language models in class-level code generation, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13.

[10] Du, Y., Ma, T., Wu, L., Zhang, X., Ji, S., 2024b. Adaccd: adaptive semantic contrasts discovery based cross lingual adaptation for code clone detection, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 17942–17950.

[11] Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Lahiri, S.K., 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. IEEE Transactions on Software Engineering .

[12] Feng, S., Chen, C., 2023. Prompting is all your need: Automated android bug replay with large language models. arXiv preprint arXiv:2306.01987 .

[13] Feng, S., Suo, W., Wu, Y., Zou, D., Liu, Y., Jin, H., 2024. Machine learning is all you need: A simple token-based approach for effective code clone detection, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13.

[14] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 .

[15] Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction .

[16] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850 .

[17] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 .

[18] Hin, D., Kan, A., Chen, H., Babar, M.A., 2022. Linevd: Statement-level vulnerability detection using graph neural networks. arXiv preprint arXiv:2203.05181 .

[19] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H., 2024. Large language models for software engineering: A systematic literature review. ACM Transactions on Software Engineering and Methodology 33, 1–79.

[20] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al., 2022. Lora: Low-rank adaptation of large language models. ICLR 1, 3.

[21] Hugging Face Inc., 2025. Hugging face. URL: https://huggingface.co.

[22] Ibrahimzada, A.R., Chen, Y., Rong, R., Jabbarvand, R., 2023. Automated bug generation in the era of large language models. arXiv preprint arXiv:2310.02407 .

[23] Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., Casas, D.d.l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al., 2023. Mistral 7b. arXiv preprint arXiv:2310.06825 .

[24] Just, R., 2014. The major mutation framework: Efficient and scalable mutation analysis for java, in: Proceedings of the 2014 international symposium on software testing and analysis, pp. 433–436.

[25] Just, R., Jalali, D., Ernst, M.D., 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, pp. 437–440.

[26] Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and evaluating contextual embedding of source code, in: International conference on machine learning, PMLR. pp. 5110–5121.

[27] Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., Amodei, D., 2020. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361 .

[28] Khajezade, M., Fard, F.H., Shehata, M.S., 2024a. Evaluating few-shot and contrastive learning methods for code clone detection. Empirical Software Engineering 29, 163.

[29] Khajezade, M., Wu, J.J., Fard, F.H., Rodríguez-Pérez, G., Shehata, M.S., 2024b. Investigating the efficacy of large language models for code clone detection, in: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, pp. 161–165.

[30] Levenshtein, V.I., et al., 1966. Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet physics doklady, Soviet Union. pp. 707–710.

[31] Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al., 2023. Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161 .

[32] Li, X., Li, W., Zhang, Y., Zhang, L., 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization, in: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pp. 169–180.

[33] Li, X., Qiu, X., 2023. Mot: Pre-thinking and recalling enable chatgpt to self-improve with memory-of-thoughts. arXiv preprint arXiv:2305.05181 .

[34] Li, Y., Wang, S., Nguyen, T., 2021. Fault localization with code coverage representation learning, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE. pp. 661–673.

[35] Li, Y., Wang, S., Nguyen, T.N., 2022. Fault localization to detect co-change fixing locations, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 659–671.

[36] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 .

[37] Loshchilov, I., Hutter, F., 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 .

[38] Lou, Y., Zhu, Q., Dong, J., Li, X., Sun, Z., Hao, D., Zhang, L., Zhang, L., 2021. Boosting coverage-based fault localization via graph-based representation learning, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 664–676.

[39] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 .

[40] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., Jiang, D., 2023. Wizardcoder: Empowering code large language models with evol-instruct. arXiv preprint arXiv:2306.08568 .

[41] MacQueen, J., et al., 1967. Some methods for classification and analysis of multivariate observations, in: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Oakland, CA, USA. pp. 281–297.

[42] Madeiral, F., Urli, S., Maia, M., Monperrus, M., 2019. Bears: An extensible java bug benchmark for automatic program repair studies, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 468–478.

[43] Meng, X., Wang, X., Zhang, H., Sun, H., Liu, X., 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge, in: Proceedings of the 44th International Conference on Software Engineering, pp. 1169–1180.

[44] Meta, 2024. Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3.

[45] Microsoft, 2023. Phi-2: The surprising power of small language models. https://huggingface.co/microsoft/phi-2.

[46] Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing, in: Proceedings of the AAAI conference on artificial intelligence.

[47] Ni, C., Wang, W., Yang, K., Xia, X., Liu, K., Lo, D., 2022. The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization, in: Proceedings of the 2022 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM. pp. 672–683.

[48] Ni, C., Wang, X., Chen, L., Zhao, D., Cai, Z., Wang, S., Yang, X., 2024. Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation. arXiv preprint arXiv:2406.15743 .

[49] Ni, C., Yin, X., Yang, K., Zhao, D., Xing, Z., Xia, X., 2023. Distinguishing look-alike innocent and vulnerable code by subtle semantic representation learning and explanation, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1611–1622.

[50] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C., 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 .

[51] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al., 2022. Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems 35, 27730–27744.

[52] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. Pytorch: An imperative style, high-performance deep learning library, in: Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035.

[53] Peng, Z., Yin, X., Qian, R., Lin, P., Liu, Y., Ying, C., Luo, Y., 2025. Soleval: Benchmarking large language models for repository-level solidity code generation. arXiv preprint arXiv:2502.18793 .

[54] Puri, R., Kung, D.S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., et al., 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655 .

[55] Qin, Y., Wang, S., Lou, Y., Dong, J., Wang, K., Li, X., Mao, X., 2025. S oap fl: A standard operating procedure for llm-based method-level fault localization. IEEE Transactions on Software Engineering .

[56] Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al., 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 .

[57] Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., Prasad, M.R., 2018. Bugs. jar: A large-scale, diverse dataset of real-world java bugs, in: Proceedings of the 15th international conference on mining software repositories, pp. 10–13.

[58] Shieh, J., 2023. Best practices for prompt engineering with openai api. OpenAI, February https://help.openai. com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api .

[59] Tian, Z., Chen, J., Wang, D., Zhu, Q., Fan, X., Zhang, L., . Leam++: Learning for selective mutation fault construction. ACM Transactions on Software Engineering and Methodology .

[60] Tian, Z., Chen, J., Zhu, Q., Yang, J., Zhang, L., 2022. Learning to construct better mutation faults, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13.

[61] Wang, W., Li, G., Ma, B., Xia, X., Jin, Z., 2020. Detecting code clones with graph neural networkand flow-augmented abstract syntax tree. arXiv preprint arXiv:2002.08653 .

[62] Wang, X., Yu, H., Meng, X., Cao, H., Zhang, H., Sun, H., Liu, X., Hu, C., 2024. Mtl-transfer: Leveraging multi-task learning and transferred knowledge for improving fault localization and program repair. ACM Transactions on Software Engineering and Methodology 33, 1–31.

[63] Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C., 2023. Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 .

[64] Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 .

[65] Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code., in: IJCAI, pp. 3034–3040.

[66] Wei, Y., Wang, Z., Liu, J., Ding, Y., Zhang, L., 2023. Magicoder: Source code is all you need. arXiv preprint arXiv:2312.02120 .

[67] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C., 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 .

[68] Xia, C.S., Wei, Y., Zhang, L., 2023. Automated program repair in the era of large pre-trained language models, in: Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery.

[69] Xia, C.S., Zhang, L., 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, pp. 959–971.

[70] Xia, C.S., Zhang, L., 2023. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. arXiv preprint arXiv:2304.00385 .

[71] Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J., 2022. A systematic evaluation of large language models of code, in: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, pp. 1–10.

[72] Yang, A.Z., Le Goues, C., Martins, R., Hellendoorn, V., 2024a. Large language models for test-free fault localization, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–12.

[73] Yang, B., Tian, H., Pian, W., Yu, H., Wang, H., Klein, J., Bissyandé, T.F., Jin, S., 2024b. Cref: An llm-based conversational software repair framework for programming tutors, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 882–894.

[74] Yin, X., 2025. Replication. URL: `https://figshare.com/s/522d36db665ba3b9d6aa`.

[75] Yin, X., Ni, C., 2024. Multitask-based evaluation of open-source llm on software vulnerability. arXiv preprint arXiv:2404.02056 .

[76] Yin, X., Ni, C., Nguyen, T.N., Wang, S., Yang, X., 2024a. Rectifier: Code translation with corrector via llms. arXiv preprint arXiv:2407.07472 .

[77] Yin, X., Ni, C., Wang, S., Li, Z., Zeng, L., Yang, X., 2024b. Thinkrepair: Self-directed automated program repair. arXiv preprint arXiv:2407.20898 .

[78] Yin, X., Ni, C., Xu, X., Yang, X., 2025. What you see is what you get: Attention-based self-guided automatic unit test generation, in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE. pp. 1039–1051.

[79] Zhang, J., Wang, C., Li, A., Sun, W., Zhang, C., Ma, W., Liu, Y., 2024. An empirical study of automated vulnerability localization with large language models. arXiv preprint arXiv:2404.00287 .

[80] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 783–794.

[81] Zhang, W., Guo, S., Zhang, H., Sui, Y., Xue, Y., Xu, Y., 2023. Challenging machine learning-based clone detectors via semantic-preserving code transformations. IEEE Transactions on Software Engineering 49, 3052–3070.

[82] Zhang, Z., Lei, Y., Tan, Q., Mao, X., Zeng, P., Chang, X., 2017. Deep learning-based fault localization with contextual information. IEICE TRANSACTIONS on Information and Systems 100, 3027–3031.

[83] Zhang, Z., Zhang, A., Li, M., Smola, A., 2022. Automatic chain of thought prompting in large language models. arXiv preprint arXiv:2210.03493 .

[84] Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Wang, Z., Shen, L., Wang, A., Li, Y., et al., 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. arXiv preprint arXiv:2303.17568 .

[85] Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B.Y., Fu, J., Chen, W., Yue, X., 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. arXiv preprint arXiv:2402.14658 .

[86] Zhong, L., Wang, Z., 2024. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation, in: Proceedings of the AAAI conference on artificial intelligence, pp. 21841–21849.