Large Language Models for Equivalent Mutant Detection: How Far Are We?

Zhao Tian
College of Intelligence and Computing, Tianjin University
Tianjin, China
tianzhao@tju.edu.cn

Honglin Shu
Kyushu University
Fukuoka, Japan
shu.honglin.167@s.kyushu-u.ac.jp

Dong Wang*
College of Intelligence and Computing, Tianjin University
Tianjin, China
dong_w@tju.edu.cn

Xuejie Cao
College of Intelligence and Computing, Tianjin University
Tianjin, China
caoxuejie@tju.edu.cn

Yasutaka Kamei
Kyushu University
Fukuoka, Japan
kamei@ait.kyushu-u.ac.jp

Junjie Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Abstract

Mutation testing is vital for ensuring software quality. However, the presence of equivalent mutants is known to introduce redundant cost and bias issues, hindering the effectiveness of mutation testing in practical use. Although numerous equivalent mutant detection (EMD) techniques have been proposed, they exhibit limitations due to the scarcity of training data and challenges in generalizing to unseen mutants. Recently, large language models (LLMs) have been extensively adopted in various code-related tasks and have shown superior performance by more accurately capturing program semantics. Yet the performance of LLMs in equivalent mutant detection remains largely unclear. In this paper, we conduct an empirical study on 3,302 method-level Java mutant pairs to comprehensively investigate the effectiveness and efficiency of LLMs for equivalent mutant detection. Specifically, we assess the performance of LLMs compared to existing EMD techniques, examine the various strategies of LLMs, evaluate the orthogonality between EMD techniques, and measure the time overhead of training and inference. Our findings demonstrate that LLM-based techniques significantly outperform existing techniques (i.e., the average improvement of 35.69% in terms of F1-score), with the fine-tuned code embedding strategy being the most effective. Moreover, LLM-based techniques offer an excellent balance between cost (relatively low training and inference time) and effectiveness. Based on our findings, we further discuss the impact of model size and embedding quality, and provide several promising directions for future research. This work is the first to examine LLMs in equivalent mutant detection, affirming their effectiveness and efficiency.

*Dong Wang and Junjie Chen are the corresponding authors.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Mutation Testing, Equivalent Mutant Detection, Large Language Model, Empirical Study

ACM Reference Format:
Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. 2024. Large Language Models for Equivalent Mutant Detection: How Far Are We?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680395

1 Introduction

Mutation testing [5, 33] involves injecting a set of intentional artificial faults into a program being tested, to measure the effectiveness of a test suite (and further enhance it). In this context, a program with an artificial fault is referred to as a *mutant*, constructed by deliberately changing a small portion of code in the program under test. Besides measuring test effectiveness, mutation testing has been extensively extended to facilitate other software testing and debugging tasks (e.g., test case prioritization [47], bug detection [68], and fault localization [65]) achieving state-of-the-art performance.

Despite its popularity and importance, mutation testing is still plagued by high costs, notably exacerbated by the presence of equivalent mutants [63, 80], a problem known to be undecidable for over three decades. An equivalent mutant is redundant because it exhibits the same behavior as the original program for all possible test cases [33, 51]. Recent research indicates that the rate of equivalent mutants in real-world development scenarios ranges from 4% to 39% [51]. Moreover, equivalent mutants introduce significant bias into mutation-based analysis. Specifically, the widely-used metric, the *mutation score*, is calculated using only non-equivalent mutants. Therefore, the presence of equivalent mutants makes it impossible to achieve a score of 100 percent. As a result, developers might not fully trust an otherwise sufficient test suite. The cost and bias issues associated with equivalent mutants can impact the practical effectiveness of mutation testing, potentially slowing down the software development process and negatively affecting software quality. Hence, detecting redundant equivalent mutants has become increasingly critical.

Over the years, numerous equivalent mutant detection (EMD) techniques have been proposed to tackle the equivalent mutant problem [20, 37]. Traditional EMD techniques often rely on predefined rules such as constraint-based testing [9, 41, 55, 73] and compiler optimization [30, 37, 62], showing limited performance in complex practical development scenarios [53, 66]. Meanwhile, more advanced learning-based EMD techniques have been proposed, including conventional machine learning-based classifiers (e.g., KNN and SVM) [11, 14, 53] and tree-based neural network models [66]. While these learning-based EMD techniques improve upon traditional techniques by comparing extracted code features, they may not fully capture program semantics, especially when it comes to minor syntax differences. Additionally, their effectiveness is limited by the scarcity of training data for equivalent mutants and potential challenges in generalizing to unseen mutants.

Lately, large language models (LLMs) have demonstrated impressive performance in both natural language processing (NLP) [75, 81] and software engineering (SE) [70, 71]. Furthermore, given that the pre-trained corpus of these LLMs (e.g., StarCoder [42] and Code Llama [81]) contains a vast amount of code snippets, they can learn generalized knowledge, thereby, in turn, boosting a variety of code-related tasks [45, 92]. Particularly, LLMs have shown promise in diverse software testing scenarios by using different learning strategies like fine-tuning and prompt engineering for tasks such as test case generation [71, 91], program debugging [17, 43], and program repair [31]. Detecting equivalent mutants is closely associated with understanding code semantics. LLMs, pre-trained on extensive code snippets from diverse resources, possess a superior grasp of code semantics compared to the above traditional learning-based EMD techniques that lack sufficient pre-training. Consequently, LLMs are more likely to effectively address the issue of data scarcity. However, there is a lack of comprehensive understanding of how well LLMs perform in detecting equivalent mutants. Encouraged by the remarkable performance of LLMs, we conjecture that they can better understand and distinguish code semantics between equivalent mutants, even with minor syntax differences.

In our paper, we conduct an empirical study on 3,302 method-level Java mutant pairs to delve into the potential of leveraging LLMs for detecting equivalent mutants. To comprehensively assess the effectiveness and efficiency of LLM-based techniques, we formulate the following four research questions with their motivations:

RQ1: What is the performance of state-of-the-art LLMs on equivalent mutant detection? We first aim to explore the capability of LLMs in detecting equivalent mutants. Specifically, we compare LLMs with ten typical or state-of-the-art existing EMD techniques to determine whether LLMs are superior.

Results: LLM-based techniques significantly surpass all ten EMD baselines in detecting equivalent mutants, with average F1-score improvements of 75.18% for Compiler-based techniques, 19.14% for ML-based techniques, and 12.75% for Tree-based NN techniques.

RQ2: What is the best strategy using LLMs for equivalent mutant detection? Different strategies (e.g., code embedding and prompting) utilized by LLMs may influence the detection performance. Thus, we further investigate the extent of their impact, which could offer insights into the optimal selection of strategies for enhancing LLM performance on equivalent mutant detection.

Results: The fine-tuned code embedding strategy demonstrates the superiority of equivalent mutant detection. Specifically, the fine-tuned UniXCoder outperforms all the combinations of LLMs and strategies with the improvement of 1.16%~78.85% in terms of F1-score. On the other hand, LLMs based solely on prompting strategies cannot achieve comparable performance.

RQ3: What degree of orthogonality exists between our studied EMD techniques? Certain EMD categories and LLM strategies may be prone to identify mutants based on specific mutation operators. Hence, RQ3 seeks to gain an understanding of the characteristics of various EMD categories and LLM strategies by analyzing the orthogonality between them.

Results: The LLM-based techniques and the fine-tuned code embedding strategy significantly surpass the other EMD categories and LLM strategies in terms of the unique correct/incorrect detections and the detection performance on each mutation operator, reinforcing the findings of RQ1 and RQ2.

RQ4: How efficient are our studied EMD techniques? As the size of LLMs has increased exponentially recently, the cost of applying these large models has also grown significantly. While larger LLMs may perform better, the trade-off between their detection performance and the cost is crucial. In this RQ, we comprehensively investigate the time efficiency (i.e., training and inference time) of all studied EMD techniques.

Results: The inference time of the best-performing LLM-based technique (0.0431 s) exceeds that of the best-performing Compiler-based technique (2.3537 s), but is marginally longer than that of the best-performing ML-based technique (0.0019 s) and the best-performing Tree-based NN technique (0.0274 s). The results highlight the LLM's excellent balance between cost and effectiveness.

Contributions. To sum up, the contributions of this study are:

- We perform the first large-scale empirical study to assess the capability of LLMs for equivalent mutant detection, considering four perspectives (i.e., effectiveness compared to existing EMD techniques, the impact of LLM strategies, orthogonality between various EMD techniques, and time efficiency).
- The study confirms the superiority of LLM-based equivalent mutant detection techniques, yielding state-of-the-art performance.
- We provide valuable insights into the capabilities and limitations of LLMs for equivalent mutant detection. The findings will serve as essential guidance for future research aimed at enhancing LLM-based equivalent mutant detection and other aspects of software engineering. Additionally, we open source all data, code, and analysis details involved in our study [29].

2 Background and Related Work

2.1 Mutation Testing

Mutation testing is a program analysis approach that involves artificially altering the source code to inject (likely) faulty behavior [52, 74]. The changing rules are called *mutation operators*, which are typically constructed based on syntactic rules derived from the grammar of the target programming language [57]. For instance, by applying the relational operator replacement mutation operator, the code fragment `"if(x==y)"` in the original program can be changed to "`if(x!=y)`", thereby constructing a mutant. Its basic assumption is that the introduced faults can effectively represent real faults [6, 22]. Mutation testing aims to evaluate the effectiveness of the test suits [23, 67]. A mutant is referred to be "killed" if it is detected by any test case; otherwise, it is said to be "live". The key metric of mutation testing is the mutation score, also known as the percentage of killed mutants. This score is calculated by dividing the number of killed mutants, those that cause test cases to fail, by the total number of generated non-equivalent mutants.

2.2 Equivalent Mutants

The equivalent mutant problem (EMP) is a critical issue in mutation testing that has been extensively studied for decades [10, 37]. A mutant is deemed equivalent if, for all possible test cases, it exhibits the same behavior as the original program under test. These mutants are syntactically different but semantically equivalent to the original program, and cannot be killed by any test cases. Equivalent mutants are often considered one of the main causes contributing to the limited adoption of mutation testing in practice due to their high computational cost and introduction of significant bias [63, 93]. Prior research has found that the rate of equivalent mutants in real-world development scenarios might lie between 4% and 39% [51]. The generation of a high number of mutants leads to increased computational costs and bias for their evaluation [28], and a significant effort is required to detect equivalent mutants.

Detecting equivalent mutants in practice is challenging since in code mutation, program equivalence is undecidable [8, 36]. A series of EMD techniques have been developed to address this issue. In earlier times, methods such as genetic algorithms [2], constraint-based testing [9, 41, 55], coverage analysis [61, 64, 72, 73], automata language equivalence [15], software behavior graphs [21], dynamic subsumption relations [20, 24], and compiler optimization [30, 37, 62] were used to identify equivalent mutants. Recently, learning-based EMD techniques have been proposed, including conventional binary classifiers (e.g., KNN and SVM) [11, 14, 53] and tree-based neural network (NN) models [66]. In particular, an early assessment of the tree-based NN technique with 582 mutants, derived from only two mutation operators, yielded promising results with a classification accuracy of 90% [66].

Among these existing EMD techniques, certain techniques involve feature extraction through the execution of mutant programs within the context of the test suite [9, 14, 21, 53]. Although leveraging information from test suites can enhance the performance of equivalent mutant detection, generating and executing a large number of test cases in practice consumes significant time and computational resources [33]. Therefore, in this paper, we only study those techniques focusing on the semantic learning of code by examining the capability of LLMs for equivalent mutant detection.

2.3 Large Language Models

Large language models have become a dominant part of NLP because of their exceptional performance, such as Llama 2 [81] and PaLM 2 [7]. Aside from the general purposes of LLMs, many LLMs have been trained on code corpora for transferring the impressive text generation capability to the code domain, such as StarCoder [42] and Code Llama [69].

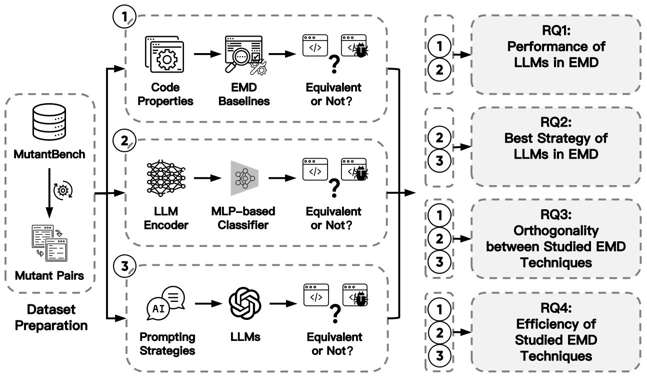


Figure 1: Overview of experimental design. ①/②/③ represents the workflow of EMD baselines/code embedding strategies/prompting strategies, respectively.

Software testing with LLMs recently has undergone significant growth [85]. Particularly, LLMs are widely used for test case generation [71, 90], program debugging [17, 43], and program repair [31] through various learning strategies such as fine-tuning and prompt engineering. To name a few, in terms of test case generation, Schäfer et al. [71] presented a large-scale empirical evaluation of the effectiveness of LLMs for automatic unit test generation with prompting strategies. Xia et al. [90] proposed Fuzz4All, which leverages LLMs as the mutation engine to produce diverse and realistic inputs for any practically relevant language, outperforming the existing language-specific fuzzers. In terms of program debugging, Feng and Chen [17] introduced a lightweight approach namely AdbGPT to reproduce the bugs from bug reports through prompt engineering. Li et al. [43] developed a technique, Differential Prompting, to effectively find failure-inducing test cases with the help of the compilable code synthesized by the inferred intention. In terms of program repair, Huang et al. [31] conducted a comprehensive study on the repair capability of five popular LLMs with the fine-tuning paradigm, suggesting that the LLM-based methods can significantly outperform previous APR techniques.

Despite these attempts, the effectiveness of LLMs in equivalent mutant detection remains largely unexplored. Recently, Ma et al. [50] conducted a preliminary study on equivalent mutant detection using a small dataset (i.e., 200 mutants) with ChatGPT. To address the limitations of dataset size and improve generalizability, our study extensively evaluates the performance of typical and state-of-the-art LLMs in equivalent mutant detection considering diverse aspects including strategies, orthogonality, and efficiency.

3 Study Design

Figure 1 illustrates the overview of our study design. Initially, based on the most widely-used MutantBench [84] dataset, we construct the training and test datasets comprising code pairs of the original program and its mutant. We first explore the effectiveness of the existing EMD techniques and state-of-the-art LLMs on equivalent mutant detection (RQ1). We also investigate the extent of impact resulting from various strategies utilized by LLMs on equivalent mutant detection (RQ2). Then, we gain an understanding of the

Table 1: Statistics of Java programs from MutantBench

Dataset	#Programs	#Methods	#EQ	#NEQ
$MutantBench_{train}$	19	328	250	1,402
$MutantBench_{test}$			249	1,401

#Programs, #Methods, #EQ, and #NEQ represent the number of programs, methods, equivalent mutants, and non-equivalent mutants, respectively.

characteristics of various EMD techniques by analyzing the orthogonality between them (RQ3). Finally, we quantitatively measure the time efficiency across the studied EMD techniques (RQ4).

3.1 Dataset Preparation

Studied dataset. To evaluate the performance of EMD techniques, we select the most widely-used MutantBench [84], containing 4,400 mutant pairs in both C/C++ and Java programming languages. More specifically, MutantBench consolidates many existing open-source datasets [9, 30, 38, 93] into one benchmark, enhancing dataset diversity and encompassing a broader spectrum of mutant types. In this study, we focus on the primary programming language, Java. Specifically, it accounts for 3,302 mutant pairs from 19 programs, as depicted in Table 1.

Data pre-processing. We carried out data pre-processing in two steps to meet the input format requirements of LLMs. Due to the constraint imposed by the maximum input token length of LLMs, we first removed all the natural language comments from the Java code utilizing pre-defined regular expressions. It is noted that natural language comments do not contribute to the equivalent mutant detection task. Second, we opted to detect the equivalence of mutant pairs at the method level instead of the program level, thereby reducing the input length. Since all mutation operators in the MutantBench dataset are applied exclusively within a single method rather than across multiple methods, ensuring that method-level mutant pairs preserve the semantic equivalence of the original mutant pairs. Hence, following existing work [79, 82], we also separated the original program and its corresponding mutant into method-level pieces. We then identified and selected the methods containing the mutation location from both the original program and its mutant. This resulted in 3,302 method-level Java mutant pairs, all of which are devoid of natural language comments.

Construction of training and test datasets. Following existing work [13, 49], we adopted a stratified sampling strategy to reduce sampling bias and ensure that both training and test datasets are representative of the entire dataset. Firstly, the 3,302 total mutant pairs were divided into two subgroups: $MutantBench_{eq}$, comprising solely equivalent mutant pairs, and $MutantBench_{neq}$, encompassing the remaining non-equivalent mutant pairs. Subsequently, ~50% of the mutant pairs from both $MutantBench_{eq}$ and $MutantBench_{neq}$ were randomly selected to construct a training dataset, denoted as $MutantBench_{train}$. Similarly, the remaining mutant pairs from both subgroups were amalgamated to form a test dataset, named $MutantBench_{test}$. Finally, we obtained 1,652 mutant pairs in $MutantBench_{train}$, comprising 250 equivalent mutant pairs and 1,402 non-equivalent mutant pairs. For $MutantBench_{test}$, we ended up with 1,650 mutant pairs, comprising 249 equivalent mutant pairs and 1,401 non-equivalent mutant pairs. In particular, we confirmed that there is no data leakage between our training and test datasets through manual inspection.

3.2 Experimented Large Language Models

In our study, we investigated the performance of ten state-of-the-art LLMs for equivalent mutant detection. These models have been widely adopted in the literature [16, 77, 85], including:

- **CodeBERT** [18] is a popular pre-trained model designed to learn from bimodal data encompassing both source code and natural languages, using a multilayer Transformer architecture.
- **GraphCodeBERT** [26] is a pre-trained model that leverages semantic-level code information to enhance code representation using a transformer-based architecture.
- **PLBART** [3] is a bidirectional and autoregressive transformer, which adopts a BART architecture and employs denoising objectives for pre-training on unlabeled data spanning source code and natural language.
- **CodeT5** [87] is a unified encoder-decoder model that incorporates token type information in code. It extends the T5 architecture, utilizing denoising sequence-to-sequence pre-training.
- **CodeT5+** [86] builds upon CodeT5. It uses a shallow encoder and a deep decoder, and is trained in multiple stages, initially with unimodal data, and later with bimodal data.
- **UniXCoder** [25] is a unified cross-modal pre-trained model that exploits multi-modal information, such as abstract syntax tree (AST) and code comments, to improve code representation. It is also based on transformer-based architecture.
- **StarCoder** [42] is based on SantaCoder architecture. It possesses its own encoder model, StarEncoder, and features infilling capabilities and rapid, large-batch inference made possible by multi-query attention.
- **Code Llama** [69] is one of the most popular LLMs for code generation and infilling derived from Llama 2 models. It is a decoder-only model and additionally fine-tuned on 500B tokens from an extra code-heavy dataset.
- **Text-Embedding Models** [60] refer to a series of new-generation embedding models developed by OpenAI [59]. These models can generate both text and code embeddings with enhanced representation capabilities. Specifically, we employed all three versions of text-embedding models (i.e., Text-Embedding-Ada-002, Text-Embedding-3-Small, and Text-Embedding-3-Large).
- **ChatGPT** [58] is a revolutionary LLM capable of transforming various fields, like software engineering. It is trained on large amounts of natural language text and code snippets, with reinforcement learning to follow human instructions. Particularly, we studied two LLMs (i.e., GPT-3.5-Turbo [58] and GPT-4 [1]).

To summarize, the studied LLMs can be divided into two types based on their architectures: encoder LLMs and decoder-only LLMs. Encoder LLMs consist of encoder-only models (i.e., CodeBERT, GraphCodeBERT, Text-Embedding Models) and encoder-decoder models (i.e., PLBART, CodeT5, UniXCoder, CodeT5+, and StarCoder). Decoder-only LLMs include Code Llama and ChatGPT.

3.3 Pre-trained Large Language Models for Code Embedding

Recently, pre-trained encoder LLMs have achieved substantial improvement in various code classification tasks, including code clone detection [35] and functionality classification [94]. Typically, these encoder LLMs are pre-trained on a large number of code snippets, learning the general-purpose code embedding knowledge. To adapt pre-trained LLMs to various downstream tasks, researchers usually train a multilayer perceptron (MLP) classifier to predict specific properties based on code embeddings produced by the pre-trained encoder LLMs [26, 54, 78]. In our study, we denote this widely-used paradigm of LLMs as the **pre-trained code embedding strategy**, representing a fundamental LLM paradigm.

Hence, we also adopted the pre-trained code embedding strategy to our studied pre-trained encoder LLMs and investigated their effectiveness in detecting equivalent mutants. As shown in Figure 1, we designed an encoder-based classifier framework, which consists of an LLM encoder and an MLP-based classifier. Specifically, we first integrated the pre-trained encoder LLMs into our designed encoder-based classifier framework for training the domain-specific classifiers to detect equivalent mutants based on our training dataset. During the training phase, we fixed the parameters of the pre-trained LLM encoder and only updated the parameters of the MLP-based classifier following existing work [26, 54]. Subsequently, multiple training iterations are performed on the training data to enable the classifier to fully learn the detection of equivalent mutants utilizing the code embeddings generated by the pre-trained LLM encoder.

In particular, for encoder LLMs (i.e., CodeBERT, GraphCodeBERT, PLBART, CodeT5, UniXCoder, CodeT5+, StarCoder, Text-Embedding-Ada-002, Text-Embedding-3-Small, and Text-Embedding-3-Large), we utilized their encoder components for obtaining embedding vectors. For the exceptional LLMs, Code Llama and ChatGPT, both of which are decoder-only architectures, are not applicable to this pre-trained code embedding strategy.

3.4 Strategies for Large Language Models

We further investigated the impact of various strategies using LLMs in equivalent mutant detection. In Section 3.3, we initially presented the fundamental pre-trained code embedding strategy. Moreover, existing studies [46, 95] have demonstrated that fine-tuning strategies can effectively adapt general LLMs to specific downstream tasks, leading to significant enhancement in LLM performance. Meanwhile, prompting strategies also have been proposed to achieve the same goal in a plug-and-play manner [39]. Thus, we devised another four LLM strategies as follows:

- **Fine-tuned code embedding strategy**: we also employed the same encoder-based classifier framework and hyper-parameter settings as those applied in the **pre-trained code embedding strategy** across all the encoder LLMs. However, we did not fix the encoder parameters; instead, we simultaneously updated the parameters of both the encoder and classifier during training.
- **Zero-shot prompting strategy**: we devised a prompt without any examples, which directly utilized a mutant pair and a structured instruction (i.e., "Please identify if the two above codes are semantically equal. Please only answer 'yes' or 'no'. 'yes' means they are semantically equal. 'no' means they are not.") to prompt LLMs for equivalent mutant detection.
- **Few-shot prompting strategy**: it enables LLMs to learn the relationship between the mutant pair and semantic equivalence based on randomly selected <*mutant pair*, *equivalence*> examples. That is, it concatenates these demonstration examples with a zero-shot prompt to form a new few-shot prompt, which is then fed to LLMs for equivalent mutant detection.
- **Fine-tuning with instruction strategy**: it enables LLMs to acquire specific knowledge through training on many more instruction-filled mutant pairs. Specifically, we used the same structured instruction that is described in the aforementioned zero-shot prompting strategy to construct an instruction-filled fine-tuning set, i.e., <*mutant pair*, *structured instruction*, *equivalence*>. Then, we fine-tuned the LLMs on the fine-tuning set to detect equivalent mutants by the zero-shot prompt.

3.5 Baselines

To fairly evaluate LLM performance, we meticulously selected the baselines by conducting a succinct literature review of relevant papers published in SE venues over the last decade. From this, we elected three widely studied techniques that rely solely on the code features without depending on the execution information of test cases, and are provided with a full replication package. These techniques encompass a total of ten baselines for comparison:

- **Compiler-based technique**. *Trivial Compiler Equivalence (TCE)* is an EMD technique based on compilation optimization [37, 62], employing the off-the-shelf compilers to compile the original program and each of its mutants into machine code, subsequently detecting mutant equivalence by comparing the equivalence of machine code pairs.
- **ML-based technique**. Brito et al. [11] extracted a set of features derived from source code properties and control flow information (e.g., mutation operator and graph distance). Based on these features, they then constructed seven ML classification models to detect equivalent mutants, including *K-Nearest Neighbors (KNN)*, *Decision Tree (DT)*, *Random Forest (RF)*, *Support Vector Machine (SVM)*, *Linear Discriminant Analysis (LDA)*, *Logistic Regression (LR)*, and *Gaussian Naive Bayes (GNB)*.
- **Tree-based neural network technique**. *Abstract Syntax Tree Neural Network (ASTNN)* takes ASTs of mutant programs as input to detect mutant equivalence [66]. This model can capture lexical-level and statement-level syntactical features of the code, as well as the code semantics by decomposing the large ASTs and encoding multi-way statement trees, significantly enhancing the detection performance.

We replicated the baseline techniques by following the implementations and parameter settings recommended in previous papers. Since the original version of the ML-based technique [11] and the Tree-based NN technique [66] only support C/C++ code, we expanded their capabilities to include Java code. Moreover, we used two variants of the most widely-used TCE baseline, namely TCE$_{Javac}$ and TCE$_{Soot}$, for comparison. Note that, we acknowledge the existence of code clone detection (CCD) techniques [35] that are built upon code similarities or patterns. However, EMD and CCD techniques serve distinct purposes and operate on different principles. Given that all mutants are inherently code snippets with minor syntactic changes (even an operator) from the original program, all the mutant pairs can be considered code clones. Therefore, we opted not to include CCD techniques as baselines in our study.

3.6 Metrics

Effectiveness. Following existing work [11, 14, 53, 66], we adopted the most widely-used metrics for the binary classification task, *Precision*, *Recall*, and *F1-score*, to assess the effectiveness of all our studied EMD techniques for equivalent mutant detection. The F1-score, being the harmonic mean of recall and precision, offers a balanced assessment of detection performance. Specifically, in our study, we used macro-averaged precision, recall, and F1-score metrics. These macro-averaged metrics are unbiased by potential class imbalances, which are calculated by finding the unweighted mean of the respective metrics for each class. For example, given two labels (i.e., negative and positive), the macro-average F1-score is the average of the F1-score for both classes (i.e., $\frac{F1_{positive}+F1_{negative}}{2}$).

Efficiency. In practice, the time spent on detecting a mutant pair is significant due to the large number of generated mutants in the mutation testing scenario. We compared the time overheads among the techniques on equivalent mutant detection to quantitatively measure their efficiency. Two types of time overheads are defined: the average time spent on detecting a mutant pair (referred to as *inference time*) and the total time spent building an EMD model based on the training set offline (referred to as *training time*).

3.7 Implementation and Environment

All the open-source pre-trained models (i.e., CodeBERT, GraphCodeBERT, PLBART, CodeT5, UniXCoder, CodeT5+, and StarCoder) are downloaded from Huggingface [89]. In particular, we used Text-Embedding-Ada-002, Text-Embedding-3-Small, Text-Embedding-3-Large, GPT-3.5-Turbo, and GPT-4 through OpenAI's APIs [59]. Concretely, we used `gpt-3.5-turbo-0125` as the specific experimental model version for GPT-3.5-Turbo and used `gpt-4-0613` as the experimental model version for GPT-4. We utilized the recommended hyper-parameters [48] for the pre-trained code embedding strategy due to their proven effectiveness. To ensure a fair comparison and reduce the complexity of hyper-parameter tuning, we reused the same hyper-parameters for the fine-tuned code embedding strategy. More detailed settings of hyper-parameters can be found in our project homepage [29]. We conducted all the experiments on an Intel Xeon CPU Gold-6342 machine with 512 GB RAM, Ubuntu 20.04.6, and two A800 GPUs.

4 Results

4.1 RQ1: Performance of LLMs in EMD

Approach. This research question offers a comparative analysis of the performance of various encoder LLMs, specifically focusing on their usage of code embeddings. The typical pre-trained code embedding strategy of LLMs for equivalent mutant detection has been provided in Section 3.3. For all eight ML-based and Tree-based NN baselines, we also trained their classifiers with our training dataset based on the same settings and process as their corresponding paper. The exceptional Compiler-based baselines (i.e., TCE$_{Javac}$ and TCE$_{Soot}$) are based on off-the-shelf compilers, requiring no training phase. Subsequently, we measured the effectiveness of 10 state-of-the-art encoder LLMs and 10 baselines in terms of precision, recall, and F1-score metrics.

Results. Table 2 shows the comparison results among LLMs and the baselines in terms of precision, recall, and F1-score. First, the evaluation results show that almost all LLM-based techniques (except CodeBERT) achieve superior effectiveness compared to all the baselines in terms of F1-score. For example, the most effective LLM-based techniques (i.e., UniXCoder and CodeT5+) achieve the F1-score of 81.88%, whereas the F1-score of ASTNN, KNN, and TCE$_{Soot}$ is 70.00%, 72.15%, and 50.80%, respectively. On average, LLM-based techniques outperform the Compiler-based, ML-based, and Tree-based NN techniques by 75.18%, 19.14%, and 12.75% in terms of F1-score, 108.27%, 15.90%, and 8.23% in terms of precision, and 62.05%, 15.25%, and 11.68% in terms of recall, respectively. It significantly demonstrates the effectiveness of LLMs on equivalent mutant detection. The prevalence of this phenomenon may stem from the fact that LLMs, typically pre-trained on extensive code snippets, exhibit enhanced capability in handling code-related downstream tasks compared to the general ML/DL models lacking such pre-training strategies. In particular, through manual analysis, we found that 363 mutant pairs could not be compiled successfully by Javac due to missing necessary configuration files. Since both TCE$_{Javac}$ and TCE$_{Soot}$ rely on the corresponding binary classfiles produced by Javac, their performance is consequently suboptimal.

Second, we observe that the performance of three state-of-the-art Text-Embedding models does not exceed other pre-trained encoder LLMs. Conversely, UniXCoder and CodeT5+, characterized by fewer parameters, demonstrate relatively superior performance in equivalent mutant detection. For instance, the F1-score of Text-Embedding-Ada-002 is 74.56% while that of UniXCoder is 81.88%. This may primarily arise because embedding models are general-purpose text embedding models trained on extensive natural language datasets. However, when employed for the code-related task (i.e., equivalent mutant detection), they are indeed affected by the data-shift problem [50]. It also suggests that smaller code-specific encoder LLMs are more inclined to produce code representations conducive to MLP-based classifiers learning the semantic differences in code.

RQ1 Summary: LLMs significantly surpass all ten EMD baselines in equivalent mutant detection. Specifically, LLMs outperform the Compiler-based, ML-based, and Tree-based NN techniques by an average improvement of 75.18%, 19.14%, and 12.75% in terms of F1-score, respectively.

4.2 RQ2: Best Strategy of LLMs in EMD

Approach. This research question aims to investigate the impact of four additional LLM strategies (designed in Section 3.4) for enhancing equivalent mutant detection compared to the pre-trained code embedding strategy in RQ1. Similarly, for the fine-tuned code embedding strategy, we selected CodeBERT, GraphCodeBERT, PLBART, CodeT5, UniXCoder, CodeT5+, and StarCoder as the base models. Besides, we selected Code Llama, GPT-3.5-Turbo, and GPT-4

Table 2: The performance of baselines and state-of-the-art LLMs on equivalent mutant detection

Technique	Precision	Recall	F1-score
Compiler-based technique			
TCE$_{Javac}$	40.55%	38.15%	39.31%
TCE$_{Soot}$	51.26%	51.81%	50.80%
ML-based technique			
KNN	77.77%	69.09%	72.15%
DT	78.20%	67.20%	70.63%
RF	78.89%	66.94%	70.53%
SVM	93.62%	58.84%	61.61%
LDA	88.02%	59.22%	62.09%
LR	90.69%	59.33%	62.30%
GNB	70.23%	62.10%	64.43%
Tree-based NN technique			
ASTNN	88.34%	65.27%	70.00%
LLM-based technique (Pre-trained code embedding strategy)			
CodeBERT (110M)	94.36%	64.26%	69.20%
GraphCodeBERT (110M)	95.81%	74.30%	80.52%
PLBART (210M)	95.81%	74.30%	80.52%
CodeT5 (210M)	95.81%	74.30%	80.52%
UniXCoder (110M)	**96.02%**	**75.70%**	**81.88%**
CodeT5+ (6B)	**96.02%**	**75.70%**	**81.88%**
StarCoder (7B)	95.99%	75.50%	81.69%
Text-Embedding-Ada-002	94.99%	68.67%	74.56%
Text-Embedding-3-Small	95.31%	70.88%	77.00%
Text-Embedding-3-Large	95.96%	75.30%	81.50%

Table 3: The performance of different LLM strategies on equivalent mutant detection

Technique	Precision	Recall	F1-score
Fine-tuned code embedding strategy			
CodeBERT (110M)	90.39%	79.74%	83.87%
GraphCodeBERT (110M)	91.54%	81.05%	85.18%
PLBART (210M)	93.24%	80.70%	85.42%
CodeT5 (210M)	90.59%	80.34%	84.37%
UniXCoder (110M)	94.33%	81.81%	**86.58%**
CodeT5+ (6B)	89.28%	**82.79%**	85.59%
StarCoder (7B)	**96.02%**	75.70%	81.88%
Zero-shot prompting strategy			
Code Llama (7B)	59.22%	50.78%	48.04%
GPT-3.5-Turbo	59.22%	59.70%	59.44%
GPT-4	67.42%	53.76%	53.61%
Few-shot prompting strategy			
Code Llama (7B)	52.85%	50.38%	47.76%
GPT-3.5-Turbo	57.04%	52.23%	51.59%
GPT-4	67.02%	55.18%	55.90%
Fine-tuning with instruction strategy			
Code Llama (7B)	93.21%	55.82%	56.79%
GPT-3.5-Turbo	92.82%	76.95%	82.31%

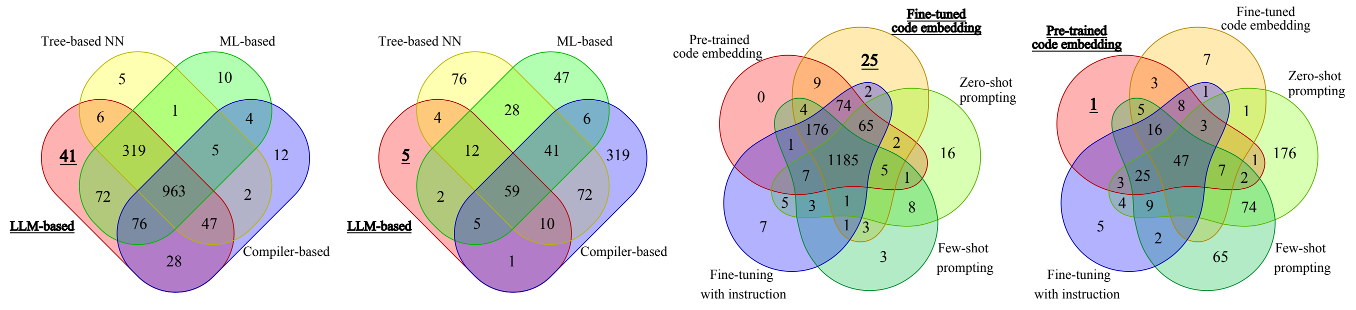
as the representative LLMs to perform zero-shot prompting, few-shot prompting, and fine-tuning with instruction strategies. Given the input length limit of LLMs, we used the 3-shot setting for the few-shot prompting strategy following the existing work [50]. In particular, we excluded GPT-4 from the fine-tuning with instruction strategy due to its unavailability. Specifically, we examined the extent of impact resulting from these four LLM strategies in terms of precision, recall, and F1-score metrics.

Results. Table 2 and Table 3 illustrate the performance of five LLM strategies on equivalent mutant detection. First, the fine-tuned UniXCoder achieves the best performance compared to all other combinations of LLMs and strategies with the improvement of 1.16%~78.85% in terms of F1-score. It demonstrates employing smaller LLMs with the fine-tuned code embedding strategy is the best approach for equivalent mutant detection.

Second, the fine-tuned code embedding strategy always outperforms the pre-trained code embedding strategy on all studied LLMs in terms of F1-score. For example, the former outperforms the latter with the improvement of 21.20%, 5.79%, 6.09%, 4.78%, 5.74%, 4.53%, and 0.23% on CodeBERT, GraphCodeBERT, PLBART, CodeT5, UniXCoder, CodeT5+, and StarCoder in terms of F1-score, respectively. It indicates that the fine-tuned code embedding strategy can significantly improve the LLM performance in equivalent mutant detection. Additionally, upon inspecting their prediction results, we observed that despite the superiority of fine-tuned LLMs over pre-trained LLMs in overall performance, a fraction of pre-training knowledge is still lost during the fine-tuning process. Specifically, among the initially correct predictions, 1.09% were erroneously transformed into incorrect predictions on average. This observation exposes the catastrophic forgetting problem [31] of LLMs under the fine-tuning paradigm to some extent.

Third, code embedding strategies (i.e., pre-trained and fine-tuned code embedding) outperform prompting strategies (i.e., zero-shot and few-shot prompting) across all three metrics (i.e., precision, recall, and F1-score). On average, code embedding strategies outperform prompting strategies by 55.81%, 41.50%, and 54.21% in terms of precision, recall, and F1-score, respectively. The main reason lies in the inherent complexity of context-understanding demanded by LLMs (with prompting strategies) for the comprehensive understanding of mutant pairs, compared to the relatively simplified process of directly comparing the embedding vectors of mutant pairs. As a result, code embedding strategies appear more straightforward in mutant understanding and comparison, thereby leading to superior detection performance.

Fourth, fine-tuning with instruction strategy significantly outperforms both zero-shot prompting and few-shot prompting strategies based on both decoder-only LLMs (i.e., Code Llama and GPT-3.5-Turbo) across all three metrics. On average, fine-tuning with instruction strategy improves 57.07% and 69.55% higher precision than zero-shot prompting and few-shot prompting, 19.41% and 29.06% higher recall, and 28.34% and 39.23% higher F1-score, respectively. It further demonstrates that the fine-tuning strategy can significantly enhance the performance of LLMs in equivalent mutant detection.



(a) Unique correct detection of four EMD categories
(b) Unique incorrect detection of four EMD categories
(c) Unique correct detection of five LLM strategies
(d) Unique incorrect detection of five LLM strategies

Figure 2: Unique correct detections (↑) and unique incorrect detections (↓) across studied EMD techniques

> **RQ2 Summary:** The fine-tuned UniXCoder significantly outperforms all other combinations of LLMs and strategies with the improvement of 1.16%~78.85% in terms of F1-score, demonstrating that fine-tuned code embedding strategy is the best strategy on equivalent mutant detection. Additionally, LLMs based solely on prompting strategies cannot achieve comparable performance.

4.3 RQ3: Orthogonality between Studied EMD Techniques

Approach. This research question aims to gain an understanding of the performance characteristics of different EMD techniques and LLM strategies. Hence, we conducted a comprehensive analysis to explore the degree of their orthogonality with two different perspectives, following the experimental design of RQ1 and RQ2:

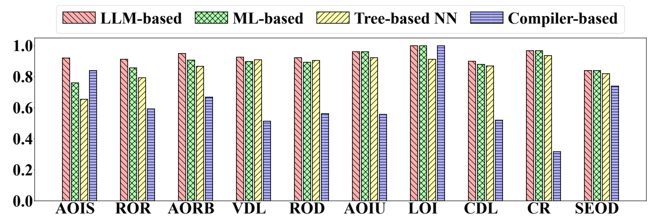
- **Between EMD categories.** Based on the findings of RQ1, we selected the best-performing EMD techniques within four EMD categories (i.e., Compiler-based, ML-based, Tree-based NN, and LLM-based techniques). Specifically, the selected techniques are TCE$_{Soot}$, KNN, ASTNN, and fine-tuned UniXCoder, each representing their respective EMD categories.
- **Between LLM strategies.** Based on the findings of RQ2, we selected the best-performing EMD techniques within five LLM strategies (i.e., pre-trained code embedding, fine-tuned code embedding, zero-shot prompting, few-shot prompting, and fine-tuning with instruction strategies). Specifically, the selected techniques are pre-trained UniXCoder, fine-tuned UniXCoder, "GPT-3.5-Turbo + zero-shot prompting", "GPT-4 + few-shot prompting", and "GPT-3.5-Turbo + fine-tuning with instruction", each representing their respective LLM strategies.

Based on the above two perspectives, we further conducted a two-level analysis: (1) **the unique correct/incorrect detections**, and (2) **the detection performance on each mutation operator**. Regarding the first level, we employed Venn diagrams to assess the unique correct/incorrect detections across various studied EMD techniques. Regarding the second level, we further investigated the detection performance of each EMD technique across various mutation operators, by disaggregating the detection results based on their respective mutation operators. In particular, the test dataset comprises 1,611 mutants, each labeled with mutation operators, alongside an additional 39 mutants unlabeled. Then the first two authors conducted a round-table discussion to manually classify these unlabeled mutants, aligning with the definition of mutation operators provided by the prior work [84]. Furthermore, we conducted a *Kruskal-Wallis test* [40], a non-parametric test for comparing differences among multiple independent groups, to assess the statistical significance between our studied EMD techniques in terms of the detection performance on each mutation operator.

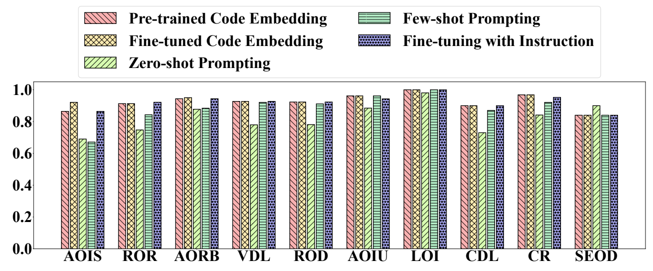
Results. Figure 2 presents the Venn diagrams that demonstrate the intersection of correct/incorrect detections among the studied EMD techniques based on two analyzed perspectives (i.e., EMD categories and LLM strategies). Overlap areas denote shared correct/incorrect detections among multiple EMD techniques, while non-overlapping areas signify the unique correct/incorrect detections of each EMD technique. Figure 3 further shows the detection performance of studied EMD techniques on each mutation operator. Due to space constraints, we only present the results on the top 10 common mutation operators. Detailed results for all 28 mutation operators are available on our project homepage [29].

Between EMD categories. From Figure 2a, we find that the LLM-based technique achieves the best performance compared to the other three EMD categories in terms of the unique *correct* detections. Specifically, the LLM-based technique exhibits 41 unique correct detections, significantly surpassing 12 by the Compiler-based technique, 10 by the ML-based technique, and 5 by the Tree-based NN technique, respectively. From Figure 2b, we find that the LLM-based technique also outperforms the other three EMD categories in terms of unique *incorrect* detections. It only has 5 unique incorrect detections, which is significantly fewer than the other EMD categories (319, 47, and 76, separately). These results prove the effectiveness of LLM-based techniques in equivalent mutant detection, further strengthening the findings in RQ1.

From Figure 3a, we further find that the LLM-based technique always outperforms the other three EMD categories in terms of detection performance on almost all mutation operators (with only one exception). For example, the LLM-based technique achieves correct detections of 339 (92.12%), 314 (91.28%), and 287 (95.03%) on the three most common mutation operators (i.e., `AOIS`, `ROR`, and `AROB`), while the suboptimal ML-based technique achieves lower



(a) Performance of 4 EMD categories on Top-10 mutation operators



(b) Performance of 5 LLM strategies on Top-10 mutation operators

Figure 3: Detection performance on Top-10 mutation operators across various EMD techniques (x-axis shows mutation operators and y-axis shows the correct detection percentage)

correct detections of 280 (76.09%), 295 (85.76%), and 274 (90.73%), respectively. For the one exceptional mutation operator (i.e., COR), the LLM-based technique exhibits slightly fewer correct detections compared to the most effective ML-based technique (13 vs 14). Furthermore, the *Kruskal-Wallis test* confirms a significant difference, with a *p-value* of 8.89e-4, suggesting that the LLM-based technique is statistically superior to all the compared EMD categories in terms of the detection performance on each mutation operator.

Between LLM strategies. From Figure 2c, we find that the fine-tuned code embedding strategy performs best compared to the other four LLM strategies in terms of the unique *correct* detections. Specifically, the fine-tuned code embedding strategy exhibits 25 unique correct detections, significantly surpassing 0 by the pre-trained code embedding strategy, 16 by the zero-shot prompting strategy, 3 by the few-shot prompting strategy, and 7 by the fine-tuning with instruction strategy. From Figure 2d, we find that both prompting strategies achieve the poorest performance among five LLM strategies regarding the unique *incorrect* detections, with 176/65 by the zero-shot/few-shot prompting strategy, compared to 1 by the pre-trained code embedding strategy, 7 by the fine-tuned code embedding strategy, and 5 by the fine-tuning with instruction strategy. It significantly suggests that LLMs based solely on prompting strategies cannot achieve comparable performance on equivalent mutant detection, further supporting the findings in RQ2.

From Figure 3b, we observe that the fine-tuned code embedding strategy always outperforms the other strategies in terms of detection performance on almost all mutation operators (with only two exceptions). For the two exceptional mutation operators (i.e., SEOD and ROR), the fine-tuned code embedding strategy exhibits slightly fewer correct detections compared to the most effective strategies (42 vs 45 and 314 vs 317, respectively). Besides, we find that both zero-shot and few-shot prompting strategies exhibit the lowest detection performance on most mutation operators. For example, the zero-shot prompting strategy achieves correct detections of 254 (69.02%), 257 (74.71%), and 265 (87.75%) on the three most common mutation operators (i.e., AOIS, ROR, and AROB), while the fine-tuned code embedding strategy achieves higher correct detections of 339 (92.12%), 314 (91.28%), and 287 (95.03%) respectively. Additionally, the *Kruskal-Wallis test* validates that there is a significant difference among all the compared strategies in terms of detection performance on each mutation operator, with a *p-value* being 1.63e-4, suggesting the superiority of the fine-tuned code embedding strategy.

RQ3 Summary: The LLM-based technique and the fine-tuned code embedding strategy significantly surpass the other EMD categories and LLM strategies regarding the unique correct/incorrect detections and the detection performance on each mutation operator.

4.4 RQ4: Efficiency of Studied EMD Techniques

Approach. This research question aims to assess the efficiency of our studied EMD techniques by calculating both training time (the total time spent building an EMD model offline) and inference time (the average time spent detecting a mutant pair). Given that the training phase is not universally applicable to all EMD techniques, such as TCE, and is conducted offline only once before the inference phase, the primary metric for evaluating efficiency across these techniques is the inference time.

Results. From Table 4, we observe that the inference time for the best-performing Compiler-based technique (i.e., TCE_{Soot}), the best-performing ML-based technique (i.e., KNN), the best-performing Tree-based NN technique (i.e., ASTNN), and the best-performing LLM-based technique (i.e., UniXCoder) are 2.3537 s, 0.0019 s, 0.0274 s, and 0.0431 s, respectively. On the one hand, compared to traditional Compiler-based techniques, LLM-based techniques have achieved significant improvements in both efficiency and effectiveness. On the other hand, while LLM-based techniques may be slightly less efficient than the ML-based and Tree-based NN techniques, their significant effectiveness makes the additional costs associated with LLMs acceptable. This indicates that LLM-based techniques provide an excellent balance between cost and effectiveness.

In addition, the pre-trained code embedding strategy significantly outperforms the fine-tuned one in terms of training time. For example, the pre-trained UniXCoder requires only 809.1785 s, whereas the fine-tuned UniXCoder requires 2566.1184 s, making the time consumption of the latter three times that of the former. It demonstrates that the limited computational resource along with the large model size of LLMs is the non-negligible factor that should be carefully considered when adopting the fine-tuned code embedding strategy to enhance LLM performance. Therefore, in practical usage, there should be a trade-off between efficiency and resource consumption for LLM-based techniques.

Table 4: Time efficiency of studied EMD techniques

Technique		Training Time (s)	Inference Time (s)
Compiler-based	TCE_{Javac}	-	1.0241
	TCE_{Soot}	-	2.3537
ML-based	KNN	298.8415	0.0019
	DT	297.3026	0.0015
	RF	300.3978	0.0081
	SVM	297.4997	0.0018
	LDA	297.7096	0.0016
	LR	296.8087	0.0016
	GNB	298.2195	0.0014
Tree-based NN	ASTNN	306.7047	0.0274
LLM-based Technique			
Pre-trained code embedding	CodeBERT (110M)	562.6160	0.0269
	GraphCodeBERT (110M)	805.1435	0.0429
	PLBART (210M)	844.1389	0.0421
	CodeT5 (210M)	1545.3771	0.0784
	UniXCoder (110M)	809.1785	0.0431
	CodeT5+ (6B)	17043.0572	0.8294
	StarCoder (7B)	16634.3038	0.9292
	Text-Embedding-Ada-002	9820.2909	0.5951
	Text-Embedding-3-Small	11346.9648	0.6876
	Text-Embedding-3-Large	19234.9228	1.1705
Fine-tuned code embedding	CodeBERT (110M)	1734.3351	0.0269
	GraphCodeBERT (110M)	2613.7416	0.0429
	PLBART (210M)	2390.2443	0.0421
	CodeT5 (210M)	4471.2962	0.0784
	UniXCoder (110M)	2566.1184	0.0431
	CodeT5+ (6B)	37286.3283	0.8294
	StarCoder (7B)	41888.5360	0.9292
Zero-shot prompting	Code Llama (7B)	-	0.2068
	GPT-3.5-Turbo	-	0.4990
	GPT-4	-	0.5808
Few-shot prompting	Code Llama (7B)	-	0.5639
	GPT-3.5-Turbo	-	0.5290
	GPT-4	-	0.6601
Fine-tuning with instruction	Code Llama (7B)	29206.5457	0.5286
	GPT-3.5-Turbo	6976.0079	0.3156

RQ4 Summary: The inference time of the best-performing LLM-based technique (0.0431 s) exceeds that of the best-performing Compiler-based technique (2.3537 s) but is marginally longer than that of the best-performing ML-based technique (0.0019 s) and the best-performing Tree-based NN technique (0.0274 s). Given the significant effectiveness of LLM-based techniques, a minor increase in inference time is deemed acceptable, highlighting their balance between cost and effectiveness.

5 Discussion

5.1 Lessons Learnt

Does the model size affect detection performance? Our study empirically validated the performance of a series of LLMs with various model sizes. The initial assumption of this study was that larger LLMs would have possessed broader prior knowledge and increased learning capacity, thereby enhancing the performance in equivalent mutant detection. However, our experimental findings indicated that the model size is not the predominant factor influencing LLM performance on equivalent mutant detection. Conversely, our findings suggest that the data modality and pre-training tasks of LLMs tend to play a more crucial role, a conclusion corroborated by existing studies [19, 25, 86]. For instance, UniXCoder surpasses all other studied LLMs, despite its smaller size as shown in RQ1 and RQ2. This superiority is likely attributed to UniXCoder leveraging AST to enhance code embeddings with rich syntax and semantic information from source code, achieved through contrastive learning involving three well-designed code-related pre-training tasks.

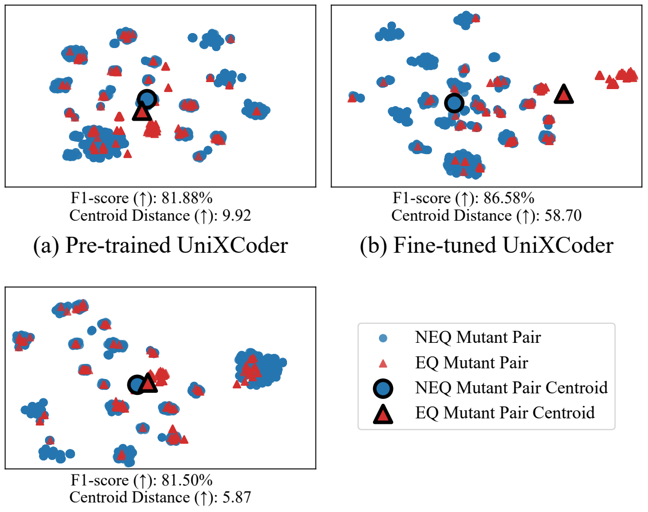


Figure 4: t-SNE plots showing the embedding of mutant pairs. EQ/NEQ represents equivalent/non-equivalent, respectively

Does the embedding quality affect detection performance? Existing studies [12, 44] indicate that the embedding quality crucially affects the ability to capture the program semantic feature for identifying an effective decision boundary. To access the embedding quality of mutant pairs, we employed t-distributed stochastic neighbor embedding (t-SNE) [83], which enables us to visually examine the relationship between code embeddings generated by various studied encoder LLMs by projecting them into a 2-dimensional space. Based on the findings of RQ1 and RQ2, we selected three best-performing LLMs within three categories (i.e., pre-trained code embedding, fine-tuned code embedding, and general text-embedding models). Specifically, the selected techniques are pre-trained UniXCoder, fine-tuned UniXCoder, and Text-Embedding-3-Large. Following existing work [4], we utilized *centroid distance* to measure the separation and quality of code embeddings. Larger centroid distance values indicate enhanced embedding quality, signifying clearer delineation in the embedding space.

Figure 4 displays the t-SNE plots for all 1,650 mutant pairs in the test set across three studied encoder LLMs. We find that fine-tuned UniXCoder (58.70) achieves better separation compared to both pre-trained UniXCoder (9.92) and Text-Embedding-3-Large (5.87) in terms of the centroid distance. The consistent performance of the three LLMs on both the centroid distance (measuring embedding quality) and F1-score (measuring detection performance) metrics suggests a strong correlation between embedding quality and detection performance.

5.2 Future Work

Across Different Programming Languages. Due to the limited computational resources and time cost, we selected the popular Java as the representative programming language for the evaluation. In the future, we will extend our experimental evaluation in other languages to comprehensively explore the performance of LLMs in equivalent mutant detection. Moreover, we plan to analyze the performance of LLMs in detecting equivalent mutants across different programming languages. This investigation will facilitate an in-depth analysis of LLMs' understanding capability of various syntaxes and structures across diverse programming paradigms.

Chain-of-Thought Prompting. In our study, we solely employed the typical prompting techniques (i.e., zero-shot prompting and few-shot prompting). Recently, Chain-of-Thought (CoT) [39, 76] prompting technique has been proposed, facilitating LLMs for tackling complex problems (e.g., mathematical reasoning and code generation), through an intermediate reasoning process to derive final solutions. Several studies have confirmed the effectiveness of CoT prompting in enhancing LLM performance across complex reasoning benchmarks [34, 88]. Hence, we can further investigate the role of CoT prompting in the task of equivalent mutant detection.

Equivalent Mutant Avoidance. Rather than detection, some research focused on avoiding the generation of equivalent mutants [51]. These avoidance techniques often involve meticulous construction of mutants employing program dependence analysis or higher-order mutation operators to reduce the number of equivalent mutants [27, 32, 56]. Equivalent mutant detection techniques and equivalent mutant avoidance techniques are orthogonal to a large extent. Future work could explore the synergy of these two categories to further enhance the mutation testing process.

Duplicated Mutant Detection. A related issue is the problem of mutant duplication. Duplicated mutants refer to mutants that are semantically equivalent to some other mutants, although both duplicated mutants may be semantically different from the original program. Duplicated mutants are also a challenge for mutation testing as they may inflate the mutant-killing effectiveness of a test suite. Kintis et al. [37] demonstrated that the equivalent mutant detection technique (i.e., TCE) can directly detect these duplicated mutants. In the future, we will delve into exploring the performance of LLMs in duplicated mutant detection.

5.3 Threats to Validity

External threat. This threat mainly lies in the equivalent mutant dataset used in our study. We only focus on programs written in Java, thus our results may not be generalized to other languages. In future work, we plan to extend our study framework to investigate LLM performance across diverse programming languages in equivalent mutant detection.

Construct threat. Three related threats are summarized. First, the construction of training and test datasets may introduce potential bias resulting from the adopted strategy. However, the stratified sampling strategy and the setting of 50% are commonly used. Second, we acknowledge that the EQ/NEQ ratio we used is not perfectly realistic. Nevertheless, our used ratio (17.80%) better reflects practical scenarios compared to the 50.00% ratio typically used in existing studies [11, 50, 66]. In future work, we will conduct a more comprehensive study to investigate the LLM sensitivity across various ratios in the more practical benchmarks. Last, due to the limited computational resources and cost, we did not run our studied EMD techniques multiple times to mitigate potential variance and randomness. Future work is encouraged to repeat the experiment multiple times and report the average results.

Internal threat. This threat mostly lies in the implementations of each studied EMD technique. To mitigate this threat, we implemented EMD techniques based on the open-source tools of each paper, and three authors have carefully reviewed the source code.

6 Conclusion

This work conducts an empirical study to extensively investigate the effectiveness and efficiency of LLMs for equivalent mutant detection. Specifically, we assess the performance of ten studied LLMs in comparison to ten existing EMD techniques, examine the various strategies of LLMs, evaluate the orthogonality between EMD techniques, and measure the time overhead of training and inference. The key findings highlight that LLM-based techniques significantly surpass all baselines, with the fine-tuned code embedding strategy being the most effective. Moreover, LLM-based techniques strike an excellent balance between cost and effectiveness. Our work also paves the way for promising future research such as the study of cross-language equivalent mutant detection, chain-of-thought prompting, combined effects with equivalent mutant avoidance techniques, and LLM application in duplicated mutant detection.

7 Data Availability

We released all the experimental data and source code on the project homepage for replication, future research, and practical use [29].

Acknowledgments

This work was supported by National Natural Science Foundation of China (Grant Nos. 62322208, 62232001), CCF Young Elite Scientists Sponsorship Program (by CAST), JSPS for the KAKENHI grants (JP21H04877, JP22K18630), Bilateral Program grant JPJSBP120239929, and the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
[2] Konstantinos Adamopoulos, Mark Harman, and Robert M Hierons. 2004. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation–GECCO 2004: Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004. Proceedings, Part II.* Springer, 1338–1349.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[4] Toufique Ahmed, Christian Bird, Premkumar Devanbu, and Saikat Chakraborty. 2024. Studying LLM Performance on Closed-and Open-source Data. *arXiv preprint arXiv:2402.15100* (2024).

[5] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering*. 402–411.

[6] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[7] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023. Palm 2 technical report. *arXiv preprint arXiv:2305.10403* (2023).

[8] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Paolo Vavassori. 2017. A novel use of equivalent mutants for static anomaly detection in software artifacts. *Information and Software Technology* 81 (2017), 52–64.

[9] Michael Baer, Norbert Oster, and Michael Philippsen. 2020. Mutantdistiller: Using symbolic execution for automatic detection of equivalent mutants and generation of mutant killing tests. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 294–303.

[10] Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. 2023. Property-based mutation testing. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 222–233.

[11] Claudinei Brito, Vinicius HS Durelli, Rafael S Durelli, Simone RS de Souza, Auri MR Vincenzi, and Márcio Eduardo Delamaro. 2020. A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 304–313.

[12] Nadia Burkart and Marco F Huber. 2021. A survey on the explainability of supervised machine learning. *Journal of Artificial Intelligence Research* 70 (2021), 245–317.

[13] Cristiano Cervellera and Danilo Macciò. 2017. Distribution-preserving stratified sampling for learning problems. *IEEE Transactions on Neural Networks and Learning Systems* 29, 7 (2017), 2886–2895.

[14] Seungjoon Chung and Shin Yoo. 2022. Augmenting Equivalent Mutant Dataset Using Symbolic Execution. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 150–159.

[15] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2018. Model-based mutant equivalence detection using automata language equivalence and simulations. *Journal of Systems and Software* 141 (2018), 1–15.

[16] Yali Du and Zhongxing Yu. 2023. Pre-training code representation with semantic flow graph for effective bug localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 579–591.

[17] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[19] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821* (2021).

[20] Rohit Gheyi, Márcio Ribeiro, Beatriz Souza, Marcio Guimarães, Leo Fernandes, Marcelo d'Amorim, Vander Alves, Leopoldo Teixeira, and Baldoino Fonseca. 2021. Identifying method-level mutation subsumption relations using Z3. *Information and Software Technology* 132 (2021), 106496.

[21] Dan Gong, Tiantian Wang, Xiaohong Su, and Yanhang Zhang. 2022. Equivalent mutants detection based on weighted software behavior graph. *International Journal of Software Engineering and Knowledge Engineering* 32, 06 (2022), 819–843.

[22] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How close are they to real faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 189–200.

[23] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2018. If You Can't Kill a Supermutant, You Have a Problem. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 18–24.

[24] Marcio Augusto Guimarães, Leo Fernandes, Márcio Ribeiro, Marcelo d'Amorim, and Rohit Gheyi. 2020. Optimizing mutation testing by discovering dynamic mutant subsumption relations. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 198–208.

[25] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[26] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[27] Mark Harman, Rob Hierons, and Sebastian Danicic. 2001. The relationship between program dependence and mutation analysis. *Mutation testing for the new century* (2001), 5–13.

[28] Dominik Holling, Sebastian Banescu, Marco Probst, Ana Petrovska, and Alexander Pretschner. 2016. Nequivack: Assessing mutation score confidence. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 152–161.

[29] Homepage. 2024. https://github.com/tianzhaotju/EMD.

[30] Mahdi Houshmand and Samad Paydar. 2017. TCE+: An extension of the tce method for detecting equivalent mutants in java programs. In *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*. Springer, 164–179.

[31] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1174.

[32] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.

[33] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[34] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689* (2023).

[35] Mohamad Khajezade, Jie Wu, Fatemeh Hendijani Fard, Gema Rodríguez-Pérez, and Mohamed Sami Shehata. 2024. Investigating the Efficacy of Large Language Models for Code Clone Detection. *arXiv preprint arXiv:2401.13802* (2024).

[36] Jinhan Kim, Juyoung Jeon, Shin Hong, and Shin Yoo. 2022. Predictive mutation analysis via the natural language channel in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–27.

[37] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2017. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering* 44, 4 (2017), 308–333.

[38] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. 2016. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 147–156.

[39] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[40] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621.

[41] Benjamin Kushigian, Amit Rawat, and René Just. 2019. Medusa: Mutant equivalence detection using satisfiability analysis. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 77–82.

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

[43] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 14–26.

[44] Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. 2022. Diffusion-lm improves controllable text generation. *Advances in Neural Information Processing Systems* 35 (2022), 4328–4343.

[45] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[46] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[47] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 46–57.

[48] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[49] Yucheng Lu, Youngsuk Park, Lifan Chen, Yuyang Wang, Christopher De Sa, and Dean Foster. 2021. Variance reduced training with stratified sampling for forecasting models. In *International Conference on Machine Learning*. PMLR, 7145–7155.

[50] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The scope of chatgpt in software engineering: A thorough investigation. *arXiv preprint arXiv:2305.12138* (2023).

[51] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering* 40, 1 (2013), 23–42.

[52] Mohsen Moradi Moghadam, Mehdi Bagherzadeh, Raffi Khatchadourian, and Hamid Bagheri. 2023. muAkka: Mutation Testing for Actor Concurrency in Akka using Real-World Bugs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 262–274.

[53] Muhammad Rashid Naeem, Tao Lin, Hamad Naeem, and Hailu Liu. 2020. A machine learning approach for classification of equivalent mutants. *Journal of Software: Evolution and Process* 32, 5 (2020), e2238.

[54] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. *arXiv preprint arXiv:2302.04026* (2023).

[55] A Jefferson Offutt and Jie Pan. 1997. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability* 7, 3 (1997), 165–192.

[56] Saeyoon Oh, Seongmin Lee, and Shin Yoo. 2021. Effectively sampling higher order mutants using causal effect. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 19–24.

[57] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–39.

[58] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt.

[59] OpenAI. 2024. https://openai.com/.

[60] OpenAI. 2024. New Generation of Embedding Model. https://openai.com/blog/new-embedding-models-and-api-updates.

[61] Mike Papadakis, Marcio Delamaro, and Yves Le Traon. 2014. Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming* 95 (2014), 298–319.

[62] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 936–946.

[63] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in computers*. Vol. 112. Elsevier, 275–378.

[64] Mike Papadakis and Yves Le Traon. 2013. Mutation testing strategies using mutant classification. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 1223–1229.

[65] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[66] Samuel Peacock, Lin Deng, Josh Dehlinger, and Suranjan Chakraborty. 2021. Automatic equivalent mutants classification using abstract syntax tree neural networks. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 13–18.

[67] James Perretta, Andrew DeOrio, Arjun Guha, and Jonathan Bell. 2022. On the use of mutation analysis for evaluating student test suite quality. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 263–275.

[68] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

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

[70] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 102–106.

[71] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[72] David Schuler and Andreas Zeller. 2010. (Un-) covering equivalent mutants. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 45–54.

[73] David Schuler and Andreas Zeller. 2013. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability* 23, 5 (2013), 353–374.

[74] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 112–122.

[75] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2998–3009.

[76] Zhao Tian and Junjie Chen. 2023. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint arXiv:2309.16120* (2023).

[77] Zhao Tian, Junjie Chen, and Zhi Jin. 2023. Code difference guided adversarial example generation for deep code models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 850–862.

[78] Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2023. On-the-fly improving performance of deep code models via input denoising. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 560–572.

[79] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[80] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empirical Software Engineering* 25 (2020), 434–487.

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

[82] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 301–312.

[83] Laurens van der Maaten and Geoffrey E. Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* (2008).

[84] Lars van Hijfte and Ana Oprescu. 2021. Mutantbench: an equivalent mutant problem comparison framework. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 7–12.

[85] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[86] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[87] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[88] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[89] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[90] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[91] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).

[92] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. An Empirical Study of Unit Test Generation with Large Language Models. *arXiv preprint arXiv:2406.18181* (2024).

[93] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th international conference on software engineering*. 919–930.

[94] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[95] Ting Zhang, Ivana Clairine Irsan, Ferdian Thung, and David Lo. 2023. Revisiting sentiment analysis for software engineering in the era of large language models. *arXiv preprint arXiv:2310.11113* (2023).