A Comprehensive Study on Large Language Models for Mutation Testing

BO WANG*, Beijing Jiaotong University, China and Beijing Key Laboratory of Traffic Data Mining and Embodied Intelligence, China
MINGDA CHEN, Beijing Jiaotong University, China
MING DENG, Beijing Jiaotong University, China
YOUFANG LIN, Beijing Jiaotong University, China and Beijing Key Laboratory of Traffic Data Mining and Embodied Intelligence, China
MARK HARMAN, University College London, United Kingdom
MIKE PAPADAKIS, University of Luxembourg, Luxembourg
JIE M. ZHANG, King's College London, United Kingdom

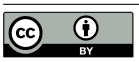
Large Language Models (LLMs) have recently been used to generate mutants both in the research community and in industrial practice. However, there has been no comprehensive empirical study of their performance for this increasingly important LLM-based Software Engineering application. To address this, we conduct a comprehensive empirical study evaluating both a wide range of traditional approaches and LLM-based approaches. Particularly, we evaluate BugFarm, LLMorpheus, and our newly designed prompt for mutant generation. The experiments cover both leading open- and closed-source LLMs, on 851 real bugs from two Java real-world bug benchmarks. Our results reveal that, compared to existing traditional approaches, LLMs generate more diverse mutants that are behaviorally closer to real bugs and, most importantly, achieve a 1.8× improvement in real bug detection, defined as the proportion of real bugs whose faulty behaviors can be mimicked by at least one generated mutant. Specifically, LLM-based approaches reach a detection rate of 77.4%, compared to 41.6% for rule-based techniques, representing an absolute gain of 35.8 percentage points. Nevertheless, our results also reveal that these impressive improvements in effectiveness come at a cost: LLM-generated mutants exhibit worse non-compilability, duplication, and equivalent mutant rates than rule-based approaches by 25.9, 7.1, and 2.6 percentage points, respectively. These findings provide actionable insights for both research and practice. They allow practitioners to have greater confidence in deploying LLM-based mutation, while researchers now have a baseline for the state-of-the-art, with which they can research techniques to further improve effectiveness and reduce cost.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Empirical Study, Mutation Testing, Mutation Analysis, Mutation Generation, Large Language Models, Coupling

*Corresponding author.

Authors' Contact Information: Bo Wang, Beijing Jiaotong University, Beijing, China and Beijing Key Laboratory of Traffic Data Mining and Embodied Intelligence, Beijing, China, wangbo_cs@bjtu.edu.cn; Mingda Chen, Beijing Jiaotong University, Beijing, China, 23120337@bjtu.edu.cn; Ming Deng, Beijing Jiaotong University, Beijing, China, 24120317@bjtu.edu.cn; Youfang Lin, Beijing Jiaotong University, Beijing, China and Beijing Key Laboratory of Traffic Data Mining and Embodied Intelligence, Beijing, China, yflin@bjtu.edu.cn; Mark Harman, University College London, London, United Kingdom, mark.harman@ucl.ac.uk; Mike Papadakis, University of Luxembourg, Luxembourg, Luxembourg, michail.papadakis@uni.lu; Jie M. Zhang, King's College London, London, United Kingdom, jie.zhang@kcl.ac.uk.



ACM 1557-7392/2026/1-ART
https://doi.org/10.1145/3805038

ACM Reference Format:
Bo Wang, Mingda Chen, Ming Deng, Youfang Lin, Mark Harman, Mike Papadakis, and Jie M. Zhang. 2026. A Comprehensive Study on Large Language Models for Mutation Testing. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 37 pages. https://doi.org/10.1145/3805038

1 Introduction

For many years, mutation testing has been the topic of scientific research [15, 25, 34, 62, 68]. Nevertheless, until the last five years, it had made little penetration into the industrial sector. In the past five years, it has become increasingly deployed in industry. For example, large tech sector companies such as Meta [4, 28] and Google [65, 67] reported on results from the deployment of mutation testing at their companies. In particular, results recently reported by Meta [28] highlighted the pivotal role played by Large Language Models (LLMs) in the successful deployment of mutation testing at scale in industry.

This recent trend towards LLM-based mutant generation traces back to work on earlier learning-based approaches, which leverage deep learning models to generate mutants by learning patterns from historical bug fixes [4, 8, 64, 74, 77]. These methods aim to introduce mutants that better resemble real faults, potentially increasing their relevance for testing.

However, despite increasing interest in LLM-based mutation testing in both research and practitioner communities, to date, no large-scale comprehensive empirical study exists. Without such a study, there is no reliable foundation for confidently deploying the technique at an industrial scale. Furthermore, the research community currently has no reliable baseline results for the state-of-the-art effectiveness and efficiency of LLM-based mutation testing compared to more traditional rule-based mutant generation techniques [34, 62].

The generation of mutants, using any technique, is challenging because of the following three important issues:

(1) **Generating effective mutants**: The ultimate goal of mutation testing is to lead to strong tests that uncover real faults, making it critical that generated mutants closely couple with real faults [6, 7, 66, 85]. While existing methods have shown promise, there is still significant room for improvement in enhancing their fault-revealing potential [14].
(2) **Generating valid mutants**: Mutants must be executable against test cases, which allows tests to assess the program's behavior in the presence of mutants, to judge whether the mutants are killed. Ensuring executable mutants is challenging because mutants must remain syntactically and semantically valid, respect type constraints, preserve control and data flow, avoid breaking external dependencies, prevent unintended side effects, and align with test case assumptions to ensure meaningful evaluation [34].
(3) **Generating diverse and killable mutants**: Determining whether a mutant is equivalent is undecidable [5], making it challenging to avoid the generation of equivalent mutants, leading to poor scalability and waste of resources [61, 81, 92, 94]. At the same time, mutants need to be diverse and cover the entire spectrum of code and program behaviors.

This paper examines how well Large Language Models (LLMs) address these challenges compared to more traditional rule-based approaches. We conducted a comprehensive empirical study that evaluates BugFarm [31] and LLMorpheus [76] (i.e., the state-of-the-art LLM-based mutant generation approaches). Moreover, we propose a novel prompt template for mutant generation, which is equipped with few-shot examples (referred to as *LLMut* for ease of presentation). All three methods are evaluated on two representative LLMs, GPT-4o and DeepSeek-V3-671b [51] (DS-671b), denoted as BugFarm (GPT-4o), BugFarm (DS-671b), LLMorpheus (GPT-4o), LLMorpheus (DS-671b), LLMut (GPT-4o), and LLMut (DS-671b), respectively. We also evaluated LLMut on five

other LLMs, i.e., GPT-3.5-Turbo, GPT-4o-Mini, DeepSeek-Coder-V2-236b, StarChat-β-16b, and CodeLlama-Instruct-13b, and report these additional results in a separate technical report [79]. We compare these LLM-based approaches with four widely-studied traditional approaches (i.e., Major [41], PIT [9], LEAM [74], and μBERT [14]), with 851 real-world bugs from two Java benchmarks (i.e., 605 bugs across 12 projects from Defects4J 2.0 [39] and 246 bugs from ConDefects [87]).

Our evaluation includes both quantitative and qualitative analysis of the mutants generated by LLMs, comparing them with existing approaches in terms of behavior similarity with real bugs (i.e., real bug detection rate, average Ochiai coefficient score, the number of high-Ochiai score bugs, coupling rate, and coupling categorization) as well as mutant validity (i.e., compilability rate, duplicate mutant rate, equivalent mutant rate, and efficiency). Additionally, we study how different prompt engineering strategies affect the effectiveness of the task. We also explore the potential of using LLMs for generating higher-order mutants. Moreover, we analyze the root causes and error types for the non-compilable mutants generated by LLMs.

Our results reveal that, compared to existing approaches, LLMs generate more diverse mutants that are behaviorally closer to real bugs. This results in a higher fault detection rate compared to traditional approaches. For instance, LLM-based approaches achieve 1.8× improvement in real bug detection rate, with an increase of 33.2 percentage points. Specifically, the mutants generated by LLMut (DS-671b) achieve fault detection rates of 91.1%, whereas those generated by LEAM, PIT, and Major yield substantially lower rates of 63.9%, 40.1%, and 66.9%, respectively. In terms of coupling rate, LLMorpheus (DS-671b) performs best, with 52.0% of its mutants coupled with real bugs. Nevertheless, we also observe that rule-based approaches, PIT and Major, outperform other approaches in terms of compilability rate, duplicate mutant rate, and equivalent mutant rate. In particular, Major excels with a 97.0% compilability rate, 0% duplicate mutants, and 2.0% equivalent mutants, while LLMut (DS-671b) exhibits a compilability rate of 77.6%, a duplicate mutant rate of 7.5%, and an equivalent mutant rate of 6.3%, respectively.

Another interesting finding of our analysis concerns the diversity of the mutants (measured in terms of newly introduced AST node types). In particular, our results show that LLMut (DS-671b) exhibits the greatest diversity, newly introducing 49 different AST node types. In comparison, rule-based approaches such as Major [41] introduce only 2 new node types, whereas LEAM [74], a state-of-the-art small model-based approach, introduces 21. The greater diversity, which subsumes all existing methods, suggests that LLMs can significantly increase mutant diversity. Moreover, the distributions of AST edit distances for mutants compared with the original code also show that LLMs generate more balanced and expressive mutants. For instance, almost 70% of the mutants of Major have an AST edit distance of one. In contrast, for the LLM-based approach, BugFarm (DS-671b), only 38% of its mutants have the same distance, indicating that LLM-based approaches tend to conduct more complex code transformations.

We further categorize non-compilable mutants generated by LLM-based approaches (i.e., the most common issue in learning-based mutation generation) and find that they fall into nine types of compilation errors, among which *Usage of Unknown Methods* and *Code Structural Destruction* are the most prevalent. Moreover, we further analyze the code context of the non-compilable mutants generated by LLM-based approaches, and find that *Method Reference* and *Method Invocation* are the most frequent code constructs. This calls for improved strategies to guide model-generated mutants toward syntactically valid and semantically meaningful variants, improving the effectiveness of AI-based mutation testing.

We analyze the surviving mutants from each approach, which are important for improving test suites. A surviving non-equivalent mutant can fall into three categories: (1) not covered, (2) covered but no state infection, or (3) state infection without detection. We analyze the distribution of surviving mutants across different approaches and find that LLMut is more likely to generate

uncovered mutants, which consequently remain undetected, providing more valuable guidance for enhancing unit tests.

In summary, our paper makes the following main contributions:

- We perform extensive and detailed comparative experiments to evaluate LLMs against existing tools/methods. Our study not only covers existing LLM-based mutant generation approaches, but also introduces a novel prompt template. Our findings indicate that LLMs excel in generating diverse mutants that closely mimic real bugs.
- We analyze the error types of non-compilable mutants and determine that member access and method invocation are more likely to lead LLMs to generate non-compilable mutants.
- We construct a high-quality dataset of Java mutants, comprehensively annotated with detailed quality and behavior metrics, which can also serve as a valuable resource for fine-tuning in future research.

The remainder of this paper is organized as follows. Section 2 presents the design of our study, including the research questions, studied approaches, datasets, and metrics. Section 3 presents the empirical evaluation of these generation approaches and answers to each research question. Section 4 discusses the influences of different experimental setups and the implications summarized from the findings. Section 5 presents the threats to the validity of our study. Section 6 discusses the related work. Section 7 concludes this work.

2 Study Design

In this section, we introduce the design and conduct of our study.

2.1 Overview and Research Questions

This paper aims to evaluate existing mutant generation approaches by answering the following research questions.

- **RQ1 (Effectiveness) How well do mutants generated by LLMs mimic real bugs compared to existing methods?** This research question investigates the effectiveness of LLM-generated mutants in representing real bugs. Specifically, it evaluates whether these mutants exhibit similar characteristics to real faults in terms of detection capability, behavioral similarity, and diversity. This RQ is divided into the following sub-RQs:
  - *RQ1.1: (Detectability) To what extent do bug-triggering tests successfully detect LLM-generated mutants?*
  - *RQ1.2 (Coupling) To what extent do LLM-generated mutants couple with real bugs?*
  - *RQ1.3 (Diversity) How diverse are LLM-generated mutants compared to traditional approaches?*
- **RQ2 (Validity) How do LLMs compare to existing approaches in terms of validity?** This research question explores the validity of LLM-generated mutants compared to traditional approaches. Validity is assessed based on key factors such as compilability, duplication, and equivalence of generated mutants. By analyzing these aspects, we aim to determine whether LLMs produce valid and meaningful mutants that contribute to practical mutation testing. This RQ is divided into the following sub-RQs:
  - *RQ2.1 (Compilability) What percentage of generated mutants are compilable across different approaches?*
  - *RQ2.2 (Duplication) What percentage of the generated mutants are duplicates of the original code across different approaches?*
  - *RQ2.3 (Equivalence) What percentage of generated mutants are equivalent across different approaches?*

Table 1. Studied Mutant Generation Approaches

Type	Name	Description
Traditional	**PIT**	Widely studied mutation testing framework with 29 mutation operators.
	Major	Widely studied mutation testing framework with nine mutation operators.
	LEAM	Custom-trained models learning from a large corpus of real bugs and patches.
	μBERT	Mutant generation with the pre-trained model CodeBERT.
LLM-based	**BF (GPT-4o)**	The existing LLM-based approach BugFarm with GPT-4o.
	BF (DS-671b)	The existing LLM-based approach BugFarm with DeepSeek-V3-671b.
	LLMph (GPT-4o)	The existing LLM-based approach LLMorpheus with GPT-4o.
	LLMph (DS-671b)	The existing LLM-based approach LLMorpheus with DeepSeek-V3-671b.
	LLMut (GPT-4o)	Our prompt with GPT-4o, trained on data before 2023/10, released in 2024/05.
	LLMut (DS-671b)	Our approach with DeepSeek-V3-671b, whose base model is DeepSeek, released in 2024/12.

- **RQ3 (Efficiency) What are the general costs of different mutant generation approaches?** This research question investigates the general costs of different mutant generation approaches. We consider time and token costs in this RQ, which are associated with the efficiency and practicality of mutant generation approaches.
- **RQ4 (Compilation Error) What are the types of compilation errors of the non-compilable mutants of different approaches?** This RQ aims to categorize and analyze the types of compilation errors that arise from non-compilable mutants generated by different approaches. Understanding these errors can guide mutant generation approaches to enhance their practical usability.
- **RQ5 (Surviving Mutants) What are the distributions of surviving mutants across different approaches?** This RQ aims to investigate the underlying reasons why mutants remain surviving, by categorizing them into three cases following the existing study [38]: (1) uncovered by tests, (2) covered but without infecting the program states, and (3) infecting the states but remaining undetected by tests. By examining how these cases are distributed across different approaches, we can better understand the effectiveness of generated mutants and their potential value for guiding test suite improvement.

2.2 Studied Mutant Generation Approaches

Our study is conducted in Java because: 1) it is one of the most widely used programming languages, shown as the TIOBE ranking[1]; 2) the majority of existing mutant generation approaches are for Java. We categorize these approaches into two groups: *traditional approaches* (i.e., those developed before the emergence of LLMs) and *LLM-based approaches* (i.e., approaches that leverage LLMs and are designed for interaction in human-like conversations), as shown below. The details of the settings for each generation approach are illustrated in Section 2.5. Table 1 summarizes the mutant generation approaches covered by this study.

2.2.1 Traditional Approaches. For Java, there exist several mutation testing frameworks, such as JavaLanche [69], MuJava [52], PIT [9], and Major [41]. As some tools are no longer maintained, following existing studies [42, 46, 60], we employ PIT [9], Major [41], LEAM [74], and μBERT [14]. Major and PIT are widely studied rule-based approaches, while LEAM and μBERT involve deep learning models.

PIT: PIT [9] is a widely used mutation testing tool on the Java bytecode level, which supports 29 well-designed mutation operators and four groups of active mutation operator sets. In our study, we employ PIT 1.14.4 and leverage the `ALL` group, which activates all operators.

Major: Major [41] is another high-impact mutation testing framework that supports nine operators. In our study, we use Major 1.3.5 and involve all mutation operators.

[1]https://www.tiobe.com/tiobe-index/

LEAM: LEAM [73, 74] is the state-of-the-art custom-trained-model-based approach, which employs sequence-to-sequence models to generate mutants. Its model is trained on a large corpus of real-world bugs and their corresponding patches, allowing it to learn patterns for generating more effective mutants.
μBERT: μBERT [14] is the state-of-the-art mutant generation approaches based on the small-scale pre-trained model, BERT [17]. Equipped with the pre-trained model, μBERT transforms the task of mutant generation to code completion under the surrounding context.

2.2.2 LLM-based Approaches. Our study incorporates two existing LLM-based mutant generation approaches (i.e., BugFarm [31] and LLMorpheus [76]), and introduces a new prompt design for mutant generation, which will be presented later.
Existing LLM-based Approaches: We adopt two state-of-the-art LLM-based mutant generation approaches, BugFarm [31] and LLMorpheus [76].

BugFarm [31] trains a small-scale model for selecting lines for mutating, and then utilizes LLMs to mutate the target line. Based on the prompt template given in the repository of BugFarm[2], we find that BugFarm numbers all lines of a Java method and uses a lightweight model to predict the target line numbers to mutate. These line numbers are then provided to the LLM. BugFarm then instructs the LLM to generate three different mutants for the Java method by changing the specified lines. Finally, BugFarm outputs each mutant as a complete mutated method, with the mutation embedded in the full method.

LLMorpheus [76] is an LLM-based mutation testing approach for JavaScript. Upon examining its prompt from its repository[3], we found it to be language-agnostic. Therefore, we reimplement the code-extraction component for Java (such as parsing ASTs and extracting the code context for mutants) while retaining its original prompt. When porting LLMorpheus to Java, we retain its default configuration. LLMorpheus masks specific syntactic elements (e.g., the conditional expression of an `if` statement) with a `PLACEHOLDER` to mark the target mutation location, and instructs the LLM to fill in this placeholder. For each masked location, LLMorpheus generates three mutants. By default, LLMorpheus provides up to 200 lines of surrounding context and instructs the LLM to return only the code fragment that should replace the placeholder.
LLMs with Our New Prompt: Different from the existing LLM-based approaches, we propose a new prompt engineering for mutant generation, shown as Figure 1. For ease of presentation, in the following part of this paper, we call our prompt **LLMut**.

To craft effective prompts, we follow the best practices which suggest that prompts should comprise four aspects: *Instruction*, *Context*, *Input Data*, and *Output Indicator* [24, 53]. In the *Instructions*, we direct the LLMs to generate mutants for the target code element. The placeholder *MUT_NUMBER* specifies the number of mutants required, while the placeholder *CODE_ELEMENT* specifies the target code snippet to be mutated. In the *Context*, we instruct that a mutant is the concept from the software engineering field, and additionally provide various sources of information to observe the impact on the performance of LLMs, including the whole Java method surrounding the target code element and few-shot examples sampled from real-world bugs from another benchmark. In the *Input Data*, we specify the code element to be mutated and the number of mutants to generate. In the *Output Indicator*, we specify the *JSON* file format for outputs, facilitating further experiments. In addition, our prompts need few-shot examples, which should originate from real bugs, enabling LLMs to learn how to mutate code from real bugs. We further explore the different prompt settings in Section 4.1.

[2]https://github.com/Intelligent-CAT-Lab/BugFarm
[3]https://github.com/githubnext/llmorpheus

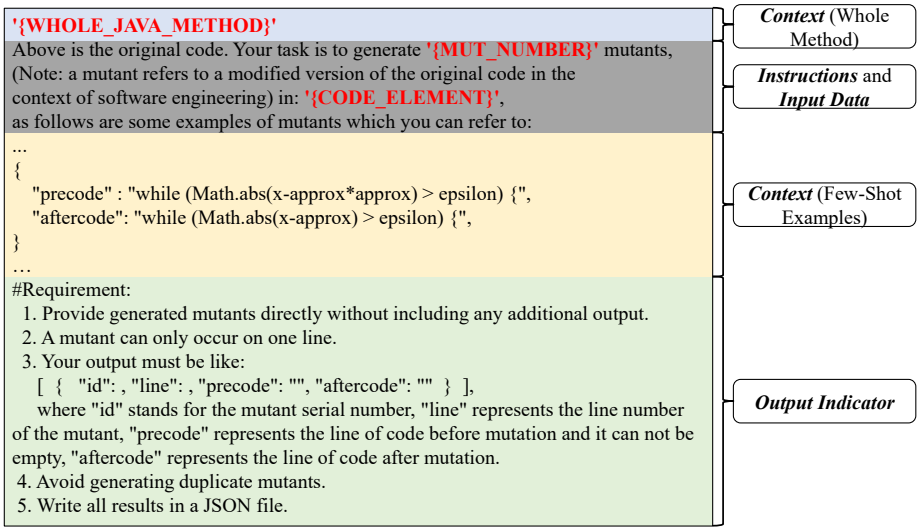


Fig. 1. Our New Prompt Template for Mutant Generation

Table 2. The Default Few-Shot Examples of LLMut from QuixBugs

Correct Version	Buggy Version
`n = (n & (n - 1));`	`n = (n ^ (n - 1));`
`while (!queue.isEmpty())`	`while (true)`
`return depth==0;`	`return true;`

`ArrayList r = new ArrayList();`
`r.add(first).addll(subset);`
`to_add(r);`
	`to_add.addAll(subset);`

`c = bin_op.apply(b,a);`	`c = bin_op.apply(a,b);`
`while(Math.abs(x-approx*approx)>epsilon)`	`while(Math.abs(x-approx)>epsilon)`

To avoid the few-shot examples leaking information, beyond the evaluation datasets (i.e., as shown later, they are Defects4J and ConDefects), we employ another benchmark, QuixBugs [49], which comprises 40 real-world Java bugs and is commonly used by the testing and debugging community [36, 48, 88, 97]. As shown later, our experimental results indicate that providing six examples results in the best overall performance, consistent with observations by Deng *et al.* [16]. Therefore, we sample six bugs from QuixBugs for few-shot learning in our prompt. To guarantee the diversity of the examples, we randomly select one bug from the dataset and assess whether its modification pattern is similar to the examples already collected. If it is different, we add it to our collection and continue until we have collected all examples, shown as Table 2. As LLM capabilities continue to advance, the optimal number of few-shot examples may vary. We therefore investigate the impact of different numbers of few-shot examples in Section 4.1.3. Note that adopting few-shot examples may also have drawbacks, which we will discuss later.

To determine the number of mutants generated in a prompt, we adopt the setting used in PIT [9]. We use PIT-generated mutants with the universal set of operators and find that PIT generates 1.15 mutants per line of code. Therefore, based on the given code context, we generate *one mutant per line*. We explore the influence of context length on mutant generation in Section 4. Additionally, for the temperature, we use the default value for each LLM.

Each LLM-based approach in our study adopts a different prompt-engineering strategy. For example, LLMut incorporates few-shot examples and uses a structured JSON output format, whereas

Table 3. Real-world Bugs Used in Our Study

Dataset	Project	# of Bugs	Time Span
Defects4J (D4J)	Math	106	2006/06/05 - 2013/08/31
	Lang	65	2006/07/16 - 2013/07/07
	Chart	26	2007/07/06 - 2010/02/09
	Time	27	2010/10/27 - 2013/12/02
	Closure	133	2009/11/12 - 2013/10/23
	Mockito	38	2009/06/20 - 2015/05/20
	Cli	39	2007/05/15 - 2018/02/26
	Codec	18	2008/04/27 - 2017/03/26
	Csv	16	2012/03/27 - 2018/05/18
	Gson	18	2010/11/02 - 2017/09/21
	JacksonCore	26	2013/08/28 - 2019/04/05
	Jsoup	93	2011/07/02 - 2019/07/04
ConDefects (CD)	—	246	2024/03/01 - 2024/06/30
Total	—	851	2006/07/16 - 2024/06/30

LLMorpheus masks high-value code fragments and instructs the LLM to fill in the masked content to ensure the quality of the output mutants. Note that the different prompt-engineering strategies are complementary, with different approaches emphasizing different aspects of the mutants.

In our study, all LLM-based approaches (i.e., BugFarm, LLMorpheus, and LLMut) are equipped with GPT-4o [1] and DeepSeek-V3-671b [51], representing the current leading closed- and open-source LLMs, respectively. The details of these models are shown as Table 1. Note that OpenAI has not officially released the parameter size of their closed-source models. The selected LLMs represent the current mainstream general-purpose LLMs, which are trained on a wide range of natural language and reasoning tasks.

2.3 Datasets

As one of the major tasks of our study is to evaluate the behavioral similarity between mutants and real-world bugs, we selected datasets that include real-world bugs along with their corresponding fault-revealing test cases. Specifically, we employ the datasets with the following properties:

- The datasets should comprise Java programs, as existing methods are primarily based on Java, and we need to compare with them.
- The bugs of the datasets should be real-world bugs so that we can compare the difference between mutants and real bugs.
- Every bug in datasets has the correctly fixed version provided by developers so that we can mutate the fixed version and compare them with the corresponding real bugs.
- Every bug is accompanied by at least one bug-triggering test because we need to measure whether the mutants affect the execution of bug-triggering tests.

To this end, we employ the Defects4J 2.0 [39] and ConDefects [87] to evaluate the mutant generation approaches, shown as Table 3. In total, we conducted experiments on 851 bugs.

Defects4J 2.0 is a widely used benchmark in the field of mutation testing [14, 40, 42, 44, 60, 74, 82], which contains historical bugs from open-source projects of diverse domains, ensuring a broad representation of real-world bugs. We select the most widely used 12 projects from Defects4J 2.0, which contain 605 bugs in total. However, from Table 1 and Table 3, we observe that the periods of the Defects4J bugs are earlier than the LLMs' training time, which may introduce data leakage. Therefore, we supplement another dataset, ConDefects [87], designed to address the data leakage concerns. ConDefects consists of tasks from AtCoder[4] programming contest. To prevent data leakage, we use data from the four most recent months following the release of the LLMs (i.e., from

[4]https://atcoder.jp

March 1, 2024, to June 30, 2024), during which we collected 246 Java bugs. Among them, only the most recent model, DeepSeek-V3-671b, cannot guarantee data leakage-free, since it was released after the ConDefects dataset.

2.4 Evaluation Metrics

In this subsection, we present the evaluation metrics used in our study to comprehensively assess the quality of the generated mutants. We evaluate the mutants from multiple perspectives, as mentioned before. Note that only relying on the mutation score is insufficient for a meaningful assessment. A higher mutation score does not necessarily indicate better quality mutants, as it only reflects the proportion of killed mutants rather than their effectiveness. However, we still report the mutation score of each mutant generation approach later in Section 3.

To better understand the metrics, we first illustrate the categories of mutants. Let all generated mutants be the set A, where LLMs may generate syntactically incorrect mutants that cannot pass compilation. Thus, we denote the compilable mutants as the set C. Note that the set C contains two types of mutants that should be removed, the duplicate mutants (denoted as D) and equivalent mutants (denoted as E). Therefore, we have $(C \supseteq D) \wedge (C \supseteq E) \wedge (D \cap E = \emptyset)$. The duplicate mutants refer to those identical to the original code or other mutants. Ideally, we should perform mutation testing against the mutant set of $(C - D - E)$, excluding all duplicated mutants (D) and equivalent mutants (E), which have no contribution to discovering the weakness of tests. Based on our definition, the duplicated mutants are syntactically identical mutants that are trivially decidable. However, determining the equivalent mutants set E is undecidable, and E is negligible compared to the universal mutant set (i.e., $|E| \ll |C - D|$). Therefore, we use the set of mutants $(C - D)$ as an approximation to compute the mutation score (i.e., $|C - D| \approx |C - D - E|$), which is the common practice of existing studies [18, 74, 82, 96]. Therefore, let K denote the set of killed mutants among the compilable and non-duplicate ones (i.e., $|C - D|$). The mutation score MS is then computed as follows:

$$MS = \frac{|K|}{|C - D|} \approx \frac{|K|}{|C - D - E|} (|E| \ll |C - D|) \tag{1}$$

Note that traditional mutant generation approaches, such as Major and PIT, produce very few duplicate mutants (i.e., $|D|$ is 0). Hence, early studies [45] defined equivalent mutants as those semantically equivalent to the original code, which may implicitly include D, which is an empty set. However, LLM-based approaches tend to generate a large number of non-mutated duplicates. Moreover, while duplicate mutants can be easily filtered, identifying equivalent mutants remains challenging [75]. Therefore, in this paper, we refine the definition of equivalent mutants to be those that are syntactically different but semantically equivalent to the original code.

2.4.1 RQ1: Effectiveness. Evaluating the effectiveness of mutant generation approaches requires assessing how well the generated mutants mimic real bugs. A key aspect of this evaluation is understanding whether LLM-generated mutants exhibit fault characteristics that align with real-world defects.

To quantify this effectiveness, we employ multiple metrics that capture different aspects of behavioral similarity between mutants and real bugs. These metrics assess whether bug-triggering tests can detect LLM-generated mutants (i.e., *RQ1.1*), how closely mutants resemble real faults based on test case failures (i.e., *RQ1.2*), and the diversity of syntactic changes introduced by different mutant generation techniques (i.e., *RQ1.3*). The metrics of these sub-RQs are listed below.

RQ1.1: Real Bug Detection Rate: Following the approach of Just et al. [40], we evaluate the effectiveness of mutant generation by measuring its ability to detect real bugs using developer-written tests. For a given real bug (which may span multiple lines), we use a mutant-generation

approach to create a set of mutants on the fixed version of the program. If the test cases that kill a mutant also fail on the real bug, we consider that the mutant generation approach has successfully "detected" this real bug. To quantify this effectiveness, we compute the *Real Bug Detection Rate*, which represents the proportion of real bugs for which at least one generated mutant is detected by the same test cases that reveal the real bug. In other words, it measures whether the generated mutants can reproduce the failing behaviors of a set of real bugs under test. This metric provides insight into how well LLM-generated mutants align with actual software defects in a testing context.

RQ1.2: Average Ochiai Coefficient: This metric estimates the semantic similarity between a mutant and its corresponding real bug based on their passing and failing test cases [59]. *Ochiai coefficient* measures how a mutation resembles its corresponding real bug. The *Ochiai coefficient* was originally introduced as a general similarity measure but later became widely adopted in spectrum-based fault localization (SBFL) [98]. In this study, we use the Ochiai coefficient in its original form to assess how closely a mutant resembles its corresponding real bug, which is consistent with the findings of Ojdanic et al. [60].

More concretely, let m and b be a mutant and its corresponding real bug, and let fT_m and fT_b be the set of their failing test cases, respectively. The coefficient of m is computed by the following formula:

$$Ochiai(m, b) = \frac{|fT_m \cap fT_b|}{\sqrt{|fT_m| \times |fT_b|}} \tag{2}$$

Therefore, if the Ochiai coefficient of m is 1, it is semantically equal to its corresponding bug b. Given a bug b, the Ochiai coefficient for b is calculated as the mean Ochiai coefficient across all its corresponding de-duplicated mutants (i.e., $(C_b - D_b)$), shown as the following formula:

$$Ochiai_b = \frac{\sum_{m \in (C_b - D_b)} Ochiai(m, b)}{|C_b - D_b|} \tag{3}$$

Given a mutant generation approach and a set of bugs B, the Average Ochiai Coefficient of the approach is calculated as the mean Ochiai coefficient across all bugs, shown as the following formula:

$$AOC = \frac{\sum_{b \in B} Ochiai_b}{|B|} \tag{4}$$

RQ1.2: High Ochiai Bug Count: Following the study [60], a bug b and a set of mutants are considered to have high Ochiai if they satisfy ($Ochiai_b \geq 0.8$). This threshold indicates a strong semantic similarity between the generated mutants and the real bug. For each mutant generation approach, we compute the number of bugs that meet this criterion.

RQ1.2: Coupling Rate: Given a real bug, one mutant is considered *coupling* if the set of test cases that kill the mutant overlaps with the bug-triggering tests. Note that each bug we analyzed has at least one failing test to reveal the bug, called *bug-triggering test*. The *Coupling Rate* refers to the proportion of coupled mutants within the set of de-duplicated mutants (i.e., $(C - D)$).

Note that *Real Bug Detection Rate* and *Coupling Rate* assess effectiveness from different granularities. *Real Bug Detection Rate* evaluates effectiveness at the bug level, i.e., **the proportion of bugs** for which at least one generated mutant reproduces the failing behavior. *Coupling Rate* evaluates effectiveness at the mutant level, i.e., **the proportion of mutants** whose failing tests overlap with the bug-triggering tests. These two metrics capture different phenomena. For example, an approach may generate many high-quality mutants for only a few bugs while producing very few mutants for the remaining bugs. In this case, the *Coupling Rate* can be high (because many mutants align well with the few covered bugs), but the *Real Bug Detection Rate* remains low (because many bugs never receive any useful mutants). Conversely, an approach might generate mutants that touch a wide range of bugs, achieving a high *Real Bug Detection Rate*. However, if per-bug mutants only

weakly reflect the bug-triggering behavior, the overall *Coupling Rate* may remain low. Therefore, **the two metrics are complementary**: *Real Bug Detection Rate* measures coverage across bugs, while *Coupling Rate* measures behavioral precision within mutants. Neither metric alone can fully represent the effectiveness of mutation generation.

RQ1.2: Coupling Categorization: We categorize each generated mutant into distinct coupling scales based on their similarity to real bugs, following the classification proposed by [23], and analyze their distribution across different approaches. Specifically, given a compilable mutant (i.e., an element from the set C) on the fixed version of a real bug (which can be uncovered by at least one triggering test), we can classify it into the following six scales based on its behavior:

- **Strong Substitution:** All triggering tests fail and no additional tests fail, which indicates that the mutant is a semantic replacement of the real bug under the given test suite.
- **Strong Substitution + Additional Tests Fail:** All triggering tests fail, but additional non-triggering tests fail.
- **Partial Substitution:** Some triggering tests fail and no additional tests fail.
- **Partial Substitution + Additional Tests Fail:** Some triggering tests fail, and some additional non-triggering tests fail.
- **Not Substitution:** Only non-triggering tests fail.
- **Not Detected:** No tests fail (i.e., the mutant is surviving under the given test suite).

Later, we will classify the mutants of each approach and analyze their distributions.

RQ1.3: AST Node Diversity: We prefer the set of mutants with diverse syntactic forms, i.e., method invocations, field accesses, etc. Therefore, we parse the code before and after mutation, and check the diversity of newly introduced AST nodes. In other words, we identify the unique AST node types introduced by each approach's generated mutants, relative to the set of node types present in the original program. The approach with a greater variety of AST node types is considered superior, as it indicates a broader exploration of the syntactic space and enhances the likelihood of detecting weaknesses in the test suite.

RQ1.3: AST Edit Distance: The AST edit distance measures the structural difference between the original and mutated code by quantifying the number of tree transformation operations (i.e., insertions, deletions, and modifications) needed to convert one AST into another. This metric evaluates the extent of syntactic changes introduced by mutations. A higher edit distance suggests more significant syntactic changes. The AST Edit Distance metric is well-suited for structured code, as it captures syntactic differences at the tree level rather than merely comparing token sequences, as in *BLEU* and similar metrics. However, it still has inherent limitations, such as its inability to fully reflect the impact of mutants on program behavior. We further discuss these limitations later in Section 5.

2.4.2 RQ2: Validity. We consider three key metrics to evaluate the validity of different mutant generation approaches: generation efficiency, compilability, and the proportion of duplicate and equivalent mutants, providing a basic understanding of the validity of each approach.

RQ2.1: Compilability Rate: the proportion of mutants that successfully compile, which is computed as:

$$CR = \frac{|C|}{|A|} \tag{5}$$

Because LLMs cannot guarantee the generation of entirely correct code and the compilation process can be time-consuming, we prefer the approach with a higher compilability rate.

RQ2.2: Duplicate Mutant Rate: the proportion of duplicate mutants, computed as:

$$DMR = \frac{|D|}{|A|} \tag{6}$$

LLMs sometimes generate duplicate mutants, i.e., mutants that are either identical to the original code or identical to each other. While traditional approaches rarely generate duplicate mutants, LLMs can produce them due to their probabilistic text-generation mechanisms and learned patterns from large-scale training data. After eliminating duplicate mutants, all remaining mutants are guaranteed to be syntactically distinct both from the original code and from each other.
RQ2.3: Equivalent Mutant Rate: the proportion of equivalent mutants, computed as:

$$EMR = \frac{|E|}{|C - D|} \tag{7}$$

Equivalent mutants are semantically equivalent to the original program, which impacts the accuracy of the mutation score. We prefer the approach with a lower *EMR*. Note that while equivalent mutants are also considered semantically duplicate, they are not easily identifiable. Therefore, we distinguish them from other duplicate mutants, which can be filtered syntactically.

2.4.3 RQ3: Efficiency. Efficiency describes how fast a mutation approach produces mutants. Mutation testing requires numerous mutants to evaluate the robustness of a test suite effectively. Therefore, the efficiency of mutant generation directly impacts the feasibility and scalability of mutation testing. To assess this, we compare these approaches based on time costs.
RQ3.1: Average Generation Time: measures the average time required to generate each mutant, providing a quantitative evaluation of the efficiency of each approach.
RQ3.2: Average Token: measures the average number of consumed tokens per mutant, reflecting both the financial cost (due to token-based pricing) and the computational time cost (since longer token sequences typically require more processing time). Since input and output token costs differ across models, we calculate them separately.

2.5 Mutant Generation Setups

In this subsection, we discuss the setups for mutant generation approaches, especially the setups for LLM-based generation.

For the well-established mutant generation approaches (i.e., PIT, Major, LEAM, and μBERT), we activate all their mutation operators. Since we try to compare the similarity between the generated mutations and real bugs, we filter the mutants within the context of real-world bugs. Note that we supply the same code snippets as context across all mutant generation approaches to ensure a fair comparison. For all LLM-based approaches, we use the default settings of each model (e.g., token limits and temperature).

3 Evaluation Results

In this section, we aim to comprehensively evaluate these approaches via various aspects. Table 4 presents the overall performance of each generation approach. In terms of the number of mutants generated (Mutant Count), PIT produces the largest number of mutants at 340,728, significantly more than other approaches, while the remaining approaches range from about twenty thousand to thirty thousand. In terms of mutation score, the traditional rule-based approaches (PIT and Major) are over 0.5, LEAM and μBERT are over 0.6, while most LLM-based approaches are over 0.7. Particularly, BugFarm (DS-671b) achieves the highest mutation score, which is nearly 0.79.

Note that the number of generated mutants and mutation scores are not quality metrics for evaluating these approaches. A large number of mutants may include redundant or equivalent

Table 4. Overall Performance of Each Mutation Generation Technique

Name	#Mut.	Mut. S.	R. B. Detec.	Avg. Ochiai	Coup. Rate	Comp. Rate	Dup. Rate	Time
PIT	340728	0.541	40.1%	23.0%	23.5%	100%	0.0%	0.02
Major	20831	0.521	66.9%	35.5%	38.9%	97.0%	0.0%	0.08
LEAM	36505	0.648	63.9%	32.4%	38.1%	44.5%	2.3%	3.06
μBERT	12946	0.677	60.5%	29.3%	47.3%	26.5%	1.7%	2.34
BF (GPT-4o)	13068	0.762	58.6%	17.7%	47.9%	74.1%	7.5%	5.35
BF (DS-671b)	13896	0.787	58.9%	17.9%	48.0%	77.5%	4.5%	7.29
LLMph (GPT-4o)	20043	0.754	68.5%	24.1%	51.7%	67.8%	7.1%	3.41
LLMph (DS-671b)	21356	0.759	69.3%	26.0%	52.0%	68.1%	6.9%	5.44
LLMut (GPT-4o)	31945	0.737	89.9%	56.2%	50.6%	76.4%	7.8%	1.31
LLMut (DS-671b)	33596	0.749	91.1%	59.8%	50.8%	77.6%	7.5%	2.57

Note: The mutation generation approach achieving the best performance in terms of a metric is highlighted in a grey background color.
BF: BugFarm. **LLMph**: LLMorpheus. **LLMut**: Our new prompt. **DS-671b**: DeepSeek-V3-671b.

Table 5. Number of Real Bugs Detected by Each Mutation Generation Approach

Proj.	Chart	Lang	Time	Math	Moc.	Clo.	Cli	Co.	Csv	Gs.	Ja.	Js.	ConD.	All
#Bug	26	65	27	106	38	133	39	18	16	18	26	93	246	851
PIT	19	23	12	40	14	38	6	6	7	6	1	48	121	341
Major	17	51	23	82	28	92	11	11	13	9	17	77	138	569
LEAM	16	27	18	44	27	122	27	12	13	11	17	90	120	544
μBERT	10	38	15	65	5	103	29	11	7	12	18	86	116	515
BF (GPT-4o)	14	37	15	57	28	93	23	6	8	11	7	83	117	499
BF (DS-671b)	16	37	15	49	25	100	22	7	8	9	7	88	118	501
LLMph (GPT-4o)	17	51	20	61	31	121	30	12	8	12	9	87	124	583
LLMph (DS-671b)	17	50	20	63	33	123	29	9	10	13	9	86	128	590
LLMut (GPT-4o)	22	55	24	94	29	131	39	16	16	16	23	92	208	765
LLMut (DS-671b)	24	61	27	101	35	131	39	13	16	16	25	92	195	775

Note: The mutation generation approaches that achieve the best performance in one project are highlighted in grey background color.
BF: BugFarm. **LLMph**: LLMorpheus. **LLMut**: our prompt. **Moc.**: Mockito. **Clo.**: Closure. **Co.**: Codec. **Gs.**: Gson. **Ja.**: JacksonCore. **Js.**: Jsoup. **ConD**: ConDefects.

mutants that do not contribute to mutation scores, while a high mutation score may be influenced by the ease with which mutants are detected rather than their resemblance to real faults.

3.1 RQ1: Performance on Effectiveness

To measure the effectiveness of mutant generation approaches, we evaluate them with three sub-RQs, which estimate their effectiveness via real bug detectability, coupling, and diversity, respectively.

3.1.1 RQ1.1 Detectability. As illustrated in Section 2.4.1, we estimate real bug detectability via the following metric.

Real Bug Detection Rate: The column *Real Bug Detec.* of Table 4 presents the proportion of real bugs that can be detected across different mutant generation approaches.

Among the traditional approaches, PIT achieves a detection rate of 40.1%, while Major, LEAM, and μBERT perform significantly better with a detection rate of 66.9%, 63.9%, and 60.5%, respectively.

For LLM-based approaches, the results are significantly more effective than those of traditional approaches. Across the evaluated models, LLMorpheus and LLMut consistently outperform all traditional approaches. For example, LLMut (DS-671b) achieves the highest detection rate at 91.1%, closely followed by LLMut (GPT-4o), with a rate of 89.9%. Moreover, we find that DS-671b outperforms GPT-4o across all three LLM-based approaches in terms of bug detection rate.

To better understand the detection of the generation approaches, we analyze their project-wise detectability, shown as Table 5. For individual projects, LLM-based approaches consistently achieve top performance across multiple projects. Notably, LLMut (DS-671b) detects the largest number of bugs in most Defects4J projects and achieves the highest overall bug detection count. Similarly,

Table 6. Number of Bugs with High Ochiai Scores (Over 0.8) Achieved by Each Mutation Generation Approach

Proj.	Chart	Lang	Time	Math	Moc.	Clo.	Cli	Co.	Csv	Gs.	Ja.	Js.	ConD.	All
#Bug	26	65	27	106	38	133	39	18	16	18	26	93	246	851
PIT	9	15	8	20	1	12	3	3	4	2	2	16	101	196
Major	9	34	14	42	12	25	3	1	2	1	3	6	150	302
LEAM	3	13	8	13	5	40	13	3	1	4	9	32	132	276
μBERT	5	18	4	27	3	14	8	3	2	3	8	20	134	249
BF (GPT-4o)	9	15	3	13	1	13	3	2	1	4	2	14	71	151
BF (DS-671b)	6	10	3	11	1	13	2	2	0	3	2	15	83	152
LLMph (GPT-4o)	6	25	7	24	6	22	9	2	3	3	5	12	81	205
LLMph (DS-671b)	6	21	8	22	5	26	11	2	3	5	6	14	92	221
LLMut (GPT-4o)	13	35	12	56	13	69	22	8	10	8	14	39	179	478
LLMut (DS-671b)	13	40	15	62	21	79	27	8	10	12	17	59	146	509

Note: The mutation generation approaches that achieve the best performance in one project are highlighted in grey background color.
BF: BugFarm. **LLMph**: LLMorpheus. **LLMut**: our prompt. **Moc.**: Mockito. **Clo.**: Closure. **Co.**: Codec. **Gs.**: Gson. **Ja.**: JacksonCore. **Js.**: Jsoup. **ConD**: ConDefects.

LLMut (GPT-4o) demonstrates strong performance, achieving the best results in projects such as ConDefects.

3.1.2 RQ1.2 Coupling. To assess the degree of coupling, we perform comprehensive experiments that include both an overall metric evaluation and a project-wise analysis.

Average Ochiai Coefficient: The column *Avg. Ochiai* of Table 4 presents the Ochiai Coefficient values across the approaches, which measures the semantic similarity between mutants and real bugs. Among the traditional approaches, Major achieves an Ochiai coefficient of 35.5%, higher than the others. LEAM and μBERT achieve coefficients of 32.4% and 29.3%, respectively, indicating that the best-performing rule-based approach, Major, outperforms both of them.

The average Ochiai coefficients of LLM-based approaches vary significantly, ranging from 17.7% to 59.8%, indicating diverse semantic distances of their mutants. Among them, LLMut (DS-671b) achieves the highest value at 59.8%, followed by LLMut (GPT-4o) at 56.2%, indicating that these mutants capture fault-triggering behaviors more closely aligned with real bugs compared to traditional approaches. Additionally, DS-671b outperforms GPT-4o in terms of average Ochiai coefficient across all three LLM-based approaches.

High Ochiai Bug Count: Following the study of Ojdanic et al. [60], the mutant set with an Ochiai coefficient over 0.8 is considered to be of high semantic similarity with its corresponding real bug. For each project, we count the number of bugs that achieved a high Ochiai coefficient under each approach, as shown in Table 6. The overall trend is similar to *Real Bug Detectability*, which indicates that the LLMut (DS-671b) achieves the highest number of bugs across most projects. Moreover, LLMut (GPT-4o) also exhibits strong performance, closely following LLMut (DS-671b) in many cases. Compared with our approach, all traditional approaches exhibit significantly fewer high Ochiai bugs. Interestingly, among traditional approaches, the rule-based approach Major outperforms the other learning-based ones, LEAM and μBERT, across most projects.

Coupling Rate: The overall coupling rate for each mutant generation approach is shown in the column *Coup. Rate* of Table 4. The traditional approaches, PIT, Major, LEAM, and μBERT, exhibit lower coupling rates than LLM-based approaches, with coupling rates of 23.5%, 38.9%, 38.1%, and 47.3%, respectively. Among the LLM-based approaches, LLMorpheus (DS-671b) and LLMorpheus (GPT-4o) achieve the highest coupling rates, reaching 52.0% and 51.7%, respectively, suggesting that LLMorpheus produces mutants whose failing behaviors align most closely with those of real bugs. LLMut (DS-671b) and LLMut (GPT-4o) closely follow LLMorpheus, with coupling rates of 50.8% and 50.6%, respectively. Interestingly, LLMut performs better in real bug detection rates, while LLMorpheus achieves higher coupling rates. This is because LLMorpheus applies mutations at specified locations (e.g., the conditional expressions of `if` statements), whereas LLMut explores

Table 7. Distribution of Coupling Categories Across Mutation Generation Approaches

Category	Strong	Strong Addi	Partial	Partial Addi	Not Subs.	Not Detected
PIT	10.1%	4.7%	1.8%	6.9%	30.6%	45.9%
Major	9.6%	5.8%	1.1%	22.4%	13.2%	47.9%
LEAM	4.9%	4.3%	0.5%	28.4%	26.7%	35.2%
μBERT	7.8%	5.2%	0.8%	33.5%	20.4%	32.3%
BugFarm (GPT-4o)	8.4%	5.4%	0.6%	33.6%	28.3%	23.8%
BugFarm (DS-671b)	10.0%	4.8%	0.7%	32.4%	30.7%	21.3%
LLMorpheus (GPT-4o)	9.3%	4.9%	2.1%	35.4%	23.6%	24.6%
LLMorpheus (DS-671b)	11.8%	4.6%	1.8%	33.8%	23.9%	24.1%
LLMut (GPT-4o)	6.8%	10.2%	1.2%	32.5%	23.1%	26.3%
LLMut (DS-671b)	11.2%	4.6%	1.9%	33.1%	24.1%	25.1%

Note: **Strong** - Strong Substitution, **Strong Addi.** - Strong Substitution + Additional Failing Tests, **Partial** - Partial Substitution, **Partial Addi.** - Partial Substitution + Additional Failing Tests, **Not Subs.** - Not Substitution.

a broader set of mutation targets without location constraints. Overall, the trends are similar to the behavior similarity discussed in Section 3.1, indicating the superiority of the LLM-based approaches, particularly LLMorpheus (DS-671b), in generating highly coupled mutants that better mimic real bugs.

Coupling Categorization: As mentioned in Section 2.4.1, we classify each mutant of each generation approach into different scales, i.e., *Strong Substitution*, *Strong + Additional Tests Fail*, *Partial Substitution*, *Partial + Additional Tests Fail*, *Not Substitution*, and *Not Detected*. Note that the first four categories are used to calculate the *Coupling Rate*, indicating these mutants semantically overlap with their corresponding real bugs.

Table 7 presents the distribution of each generation approach. Among the traditional approaches, PIT and Major exhibit a significantly higher proportion of mutants in the *Not Detected* category, with rates of 45.9% and 47.9%, respectively, indicating that a substantial number of their generated mutants fail to couple with real bugs. However, they achieve relatively higher proportions in the *Strong* category, with PIT at 10.1% and Major at 9.6%, suggesting that while their coupling ability is limited, they can still generate a small number of strongly coupled mutants. In contrast, LLM-based approaches overall demonstrate stronger coupling capabilities across multiple categories. For instance, LLMorpheus (DS-671b) achieves the highest proportion in the *Strong* category at 11.8%, closely followed by LLMut (DS-671b) with the value of 11.2%. BugFarm (DS-671b) also achieves a notable 10.0% of strongly coupled mutants. For the *Strong + Additional Tests Fail* category, LLMut (GPT-4o) leads with a proportion of 10.2%. Additionally, BugFarm and LLMorpheus exhibit the lowest proportion of *Not Substitution* and *Not Detected*, indicating that their mutation strategies target code locations that are more likely to affect execution, such as conditional expressions. Overall, LLM-based approaches generate a larger proportion of both strongly coupled and partially coupled mutants, demonstrating a stronger ability to generate diverse and highly coupled mutants.

3.1.3 RQ1.3 Diversity. We explore the syntactic diversity of mutants generated by each approach. Note that PIT works at the Java bytecode level; therefore, we skip it in this RQ.

AST Node Diversity: To assess the diversity of AST node types, we first determine if a mutant involves *deletion* (i.e., removing a code element). For the *non-deletion mutants*, we parse the code before and after mutation by the parser Javalang[5], to observe which new AST node types are introduced. A larger number of newly introduced AST node types indicates richer diversity.

Figure 2 illustrates the number of introduced AST node types, with traditional approaches highlighted in green and LLM-based approaches in blue. The figure reveals that LLM-based approaches present more types of new AST nodes than traditional approaches. The best LLM-based approach, LLMut (DS-671b), exhibits the greatest diversity, introducing 49 new AST node types, followed by

[5]https://pypi.org/project/javalang

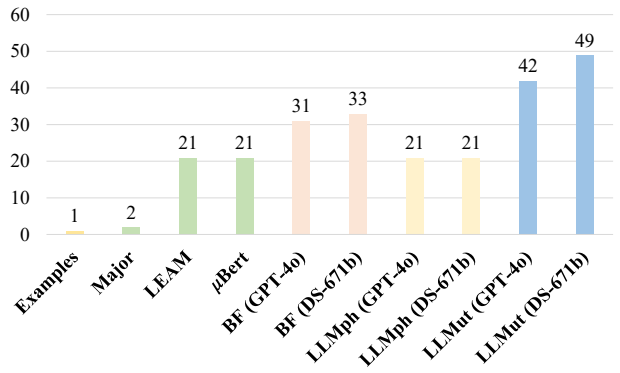


Fig. 2. Number of Newly Introduced AST Nodes of the Studied Approaches and the Few-Shot Examples

Table 8. Ratio of Deletion Mutations and the Top-3 Newly Introduced AST Node Types for Each Approach

Approach	Deletion	1st	2nd	3rd
Major	15.3%	ST (78.6%)	LT (21.4%)	——
LEAM	48.9%	LT (23.2%)	MR (14.4%)	CT (9.6%)
μBERT	0.4%	LT (29.4%)	SE (25.7%)	MR (24.5%)
BugFarm (GPT-4o)	5.5%	LT (41.9%)	BO (12.2%)	MI (9.0%)
BugFarm (DS-671b)	2.1%	LT(45.8%)	BO(14.7%)	MR(8.8%)
LLMorpheus (GPT-4o)	0.3%	LT (24.0%)	BO (23.7%)	MR (15.8%)
LLMorpheus (DS-671b)	0.3%	LT(29.5%)	BO(25.0%)	MR(15.2%)
LLMut (GPT-4o)	2.1%	LT (45.4%)	BO (16.6%)	CC (8.8%)
LLMut (DS-671b)	3.8%	LT (44.2%)	BO (15.6%)	CC (8.6%)

Deletion: The proportion of deletion mutations. **BO**: BinaryOperation. **CC**: ClassCreator. **CT**: Cast. **LT**: Literal. **MI**: MethodInvocation. **MR**: MemberReference. **SE**: StatementExpression. **ST**: Statement.

LLMut (GPT-4o), which introduces 42 new types. Meanwhile, BugFarm (DS-671b) and BugFarm (GPT-4o) present 33 and 31 new AST node types, respectively. Additionally, LLMut and BugFarm introduce significantly more AST node types than traditional approaches. We also check the AST node diversity of the examples we provided in the LLM prompts (see Table 2), as shown by the "Examples" in Figure 2. Interestingly, we find that these examples introduce only one new type of AST node and are subsumed by all other approaches. This indicates that LLM-based approaches are creative in generating mutants, producing significantly more diverse mutants than the examples provided to them.

We further analyzed the distribution of deletions and AST node types across different approaches, as shown in Table 8, which highlights the proportion of the mutants that delete the target code element and the top three most frequent AST node types. For the proportion of deletions, the traditional approach, LEAM, shows the highest deletion mutant ratio, with 48.9%, followed by Major at 15.3%. In contrast, all LLM-based approaches generate deletion mutants at a rate below 6%. This indicates that, compared to existing approaches, LLM-based approaches are less inclined to generate deletion mutants. As shown later, we find that deletions are one of the key factors contributing to the lower compilation success rate.

Regarding AST node types, the most frequent new AST node type of all the LLM-based approaches is `Literal` (LT) (e.g., $a > b \mapsto$ true). However, for the traditional approach Major, the most frequent node type is Statement (e.g., replacing a statement with an empty statement). Note that although these mutants are not with the deletion operator, a significant portion of the production of empty statements exists, leading to a large proportion. For the second most frequent new node types,

Table 9. The Distribution of AST Edit Distance for Each Approach

Distance	1	2	3	4	5	≥6
Major	69.2%	27.8%	1.8%	0.6%	0.4%	0.2%
LEAM	58.5%	23.6%	7.3%	3.4%	1.0%	6.1%
μBERT	73.8%	8.9%	2.7%	14.2%	0.1%	0.4%
BugFarm (GPT-4o)	44.7%	43.5%	8.3%	1.6%	0.6%	1.4%
BugFarm (DS-671b)	38.3%	52.1%	6.7%	1.5%	0.7	0.7
LLMorpheus (GPT-4o)	54.2%	25.7%	14.6%	3.7%	0.9%	0.8%
LLMorpheus (DS-671b)	53.9%	31.8%	10.6%	2.5%	0.4%	0.8%
LLMut (GPT-4o)	50.0%	38.8%	7.2%	2.2%	0.8%	1.0%
LLMut (DS-671b)	46.1%	44.2%	6.8%	1.5%	0.4%	0.9%

`BinaryOperation` (BO) appears most often among LLM-based approaches, indicating that LLM-based approaches frequently modify or insert binary expressions such as arithmetic or comparison operators. For the third most frequent category, LLM-based approaches introduce a variety of AST nodes, including `MethodInvocation` (MI), `MemberReference` (MR), and `ClassCreator` (CC), reflecting their broader syntactic diversity in mutant generation.

AST Edit Distance: To quantify the extent of syntactic modifications introduced by each mutant generation approach, we compute the AST edit distance for each mutant using Gumtree [20]. The distribution of edit distances for each approach is presented in Table 9, providing insights into the complexity and diversity of the generated mutants.

From the table, we find that all traditional approaches predominantly generate small-scale changes. Specifically, the proportion of mutants with an edit distance of 1 for Major, LEAM, and μBERT is 69.2%, 58.5%, and 73.8%, respectively, indicating they involve minimal syntactic changes to the original code in the majority of cases. In contrast, LLM-based approaches exhibit a broader range of edit distances, reflecting their capability to introduce more complex transformations. For example, only 38.3% of the mutants of BugFarm (DS-671b) have an edit distance of 1, and LLMut (DS-671b) generates 45.1% of mutants with an edit distance of 1 and 44.2% with a distance of 2, with an additional 6.9% corresponding to distances of 3. This pattern suggests that LLM-based methods, unlike traditional approaches, are capable of performing multi-node or compositional edits, leading to a richer diversity of syntactic structures.

Answer to RQ1: In most cases, LLM-based approaches outperform traditional methods in effectiveness. Specifically, when weighted by the number of generated mutants, rule-based approaches (i.e., PIT and Major) achieve an average real bug detection rate of 41.64% and a coupling rate of 24.37%. In contrast, LLM-based approaches reach 77.44% and 50.50%, respectively, representing significant improvements in both metrics. Among all techniques, LLMut (DS-671b) achieves the highest real bug detection rate (91.1%). Similarly, LLMorpheus (DS-671b) attains the highest coupling rate (52.0%). Moreover, LLM-generated mutants exhibit substantially greater syntactic diversity, introducing a wider range of new AST node types (e.g., 49 for LLMut (DS-671b)) and a lower proportion of deletion mutations.

3.2 RQ2: Performance on Validity

To evaluate the validity of mutant generation approaches, we consider the aspects that may impact their validity, i.e., compilability rate, duplicate mutant rate, and equivalent mutant rate. This RQ is divided into the following sub-RQs.

3.2.1 RQ2.1: Compilability. The column *Comp. Rate* of Table 4 presents the overall Compilability Rate (*CR*) of each approach. Among the traditional approaches, the rule-based approaches, PIT

and Major, demonstrate superior compilability, with PIT achieving a perfect 100% *CR*, and Major following closely at 97.60%. Note that although Major operates under simple syntax rules, it can still generate non-compilable mutants. For example, the Java compiler's analyzer rejects constructs like `while(true)`. PIT uses the mutation operators more restrictively and previously filters out non-compilable mutants. In contrast, LEAM and μBERT, have rates at 35.0% and 22.5%, respectively.

For LLM-based approaches, there is a considerable variation in compilability. Among all LLM-based approaches, LLMut (DS-671b) and BugFarm (DS-671b) attain the highest *CR* of 77.6% and 77.5%, respectively. LLMut (GPT-4o), closely follows them by achieving 76.4%. Most LLM-based approaches show significantly higher compilation success rates compared to the traditional learning-based approaches, such as LEAM and μBERT. For example, the *CR*s of BugFarm, LLMorpheus, and LLMut with GPT and DeepSeek models are over 30% higher than those of LEAM and μBERT.

3.2.2 RQ2.2: Duplicate Mutant Rate. As aforementioned, duplicate mutants refer to those that are redundant or identical to the original code, providing no meaningful contribution to the mutant testing process. Therefore, a lower Duplicate Mutant Rate (*DMR*) is desirable, as it indicates a more efficient mutant generation process with minimal redundancy. The column *Dup. Mut. Rate* of Table 4 presents the results.

The rule-based approaches (i.e., PIT and Major) achieve a perfect 0% *DMR*. This is expected, as these approaches apply well-defined mutation rules that ensure each generated mutant introduces a meaningful change to the program. Other traditional approaches, i.e., LEAM and μBERT, show relatively low *DMR*s of 2.3% and 1.7%, respectively. In contrast, LLM-based approaches generate a significant portion of duplicate mutants, exhibiting a higher *DMR*. For the LLM-based approaches evaluated with the best-performing model, DS-671b, the duplicate mutation rates (*DMR*s) are as follows: BugFarm (DS-671b): 4.5%, LLMorpheus (DS-671b): 6.9%, and LLMut (DS-671b): 7.5%. These results indicate that even equipped with the state-of-the-art LLMs, these approaches still generate a non-negligible proportion of duplicate mutants.

3.2.3 RQ2.3: Equivalent Mutants. Given the undecidable nature of identifying equivalent mutants, we manually evaluate each mutant's equivalence, following the methodology of the existing study [46]. For each mutant generation approach, we randomly sample a subset of compilable mutants, ensuring a 95% confidence level with a 10% margin of error.

Two authors independently evaluate the equivalence of the selected mutants to minimize subjectivity. The procedure for checking a mutant involves first transplanting it into the original project and executing all tests. If it passes all tests, the author then manually verifies whether the mutant is semantically equivalent. Note that PIT does not generate source-level mutants, so its mutants must be manually extracted from Java bytecode and transformed back into source-level representations, which is significantly more time-consuming.

In total, 1437 mutants were labeled, 860 of which were filtered by tests (i.e., killed), and the remaining 577 were manually checked. The entire process took approximately thirteen hours per person. The manual verification requires checking the code context, which may involve multiple classes and methods. For example, to check a mutant that replaces an integer with a method invocation, where the method is invoked by the *this* object but defined in the parent class, we should review the parent class file and verify whether its return value is consistently equal to the integer. We measure the consistency between the two authors by Cohen's kappa coefficient (κ) [78], achieving an overall κ value of 0.8, which indicates a high level of agreement. In cases of disagreement, both authors conducted thorough discussions to resolve contradictions and finally reach consistent labels. The labeling results are available at our repo.

Table 10 presents the estimated Equivalent Mutant Rate (*EMR*), reporting the number of sampled mutants (#Sampled), the number of equivalent mutants we identified (#Eq. Mut), and the *EMR* for

Table 10. Sample Compilable Mutations for Identifying Equivalent Mutations

Approach	#Sampled	#Test Filtered	#Eq. Mut.	EMR
PIT	97	26	1	1.0%
Major	96	52	2	2.1%
LEAM	96	48	2	2.1%
μBERT	94	64	3	6.4%
BugFarm (GPT-4o)	96	67	3	3.1%
BugFarm (DS-671b)	96	75	5	5.2%
LLMorpheus (GPT-4o)	96	69	4	4.2%
LLMorpheus (DS-671b)	96	65	3	3.1%
LLMut (GPT-4o)	96	64	3	3.1%
LLMut (DS-671b)	96	65	6	6.3%

each approach. Among the traditional approaches, PIT achieves the lowest *EMR* of 1.0%, followed by Major with an *EMR* of 2.1%, indicating that rule-based techniques produce fewer equivalent mutants. Overall, the LLM-based approaches exhibit higher *EMR*s, with variations across different models. LLMut (DS-671b) performs the worst by attaining the highest *EMR*s of 6.3%, while LLMut (GPT-4o), BugFarm (GPT-4o), and LLMorpheus (GPT-4o) a show relatively lower *EMR* of 3.1%. The LLM-based approaches tend to have slightly higher *EMR*s than PIT and Major, the increase remains modest and within an acceptable range.

Note that the estimated *EMR* of Major given by the recent study of Kushigian et al. [46] is 2.97%, where the authors sampled a subset of mutants for manual judgment. We adopt the same experimental process, and the result of Major is 2.0%, which is consistent with theirs, considering the randomness introduced by sampling. Moreover, the definition of equivalent mutants in this paper is the set of syntactically different but semantically equivalent mutants, which may differ from existing studies [76].

Answer to RQ2: The rule-based approaches (i.e., PIT and Major) outperform the LLM-based approaches in compilability rate, duplicate mutant rate, and equivalent mutant rate, achieving weighted averages of 99.83%, 0.00%, and 1.55%, respectively. In contrast, the weighted average values of LLM-based approaches are 73.98%, 7.10%, and 4.17%. Although rule-based approaches slightly outperform LLM-based approaches in terms of equivalent mutant rate, the equivalent mutant rates across both approaches are generally comparable.

3.3 RQ3: Performance on Efficiency

To evaluate the performance in terms of efficiency, we record the average generation time and the number of used tokens for each approach.

Average Generation Time: The column *Time* of Table 4 presents the results of *Average Generation Time* in seconds. Among the traditional approaches, PIT demonstrates the fastest generation time at 0.02 seconds per mutant, followed by Major, which takes 0.08 seconds, making them the most efficient approaches. In contrast, LEAM and μBERT take 3.06 seconds and 2.34 seconds, respectively. The LLM-based approaches present much slower generation speed, ranging from 1.31 to 7.29 seconds to generate a mutant. The results suggest that the rule-based approaches are the most time-efficient for generating mutants, which are tens to hundreds of times faster than other approaches. Note that the response time of models accessed via network APIs may vary across service providers and can be further affected by network fluctuations. We further discuss these potential impacts in Section 5.

Table 11. Average Number of Used Tokens Across All LLM-Based Approaches

Approach	Avg. Input Token	Avg. Output Token	All Input Token	All Output Token
BugFarm (GPT-4o)	200	368	2,780,213	4,807,979
BugFarm (DS-671b)	200	366	2,780,213	5,085,936
LLMorpheus (GPT-4o)	370	74	7,921,356	1,476,568
LLMorpheus (DS-671b)	370	73	7,921,356	1,558,988
LLMut (GPT-4o)	126	59	4,353,426	1,875,810
LLMut (DS-671b)	126	58	4,353,426	1,947,224

Average Token: Table 11 presents the average number of input and output tokens used by each LLM-based approach to generate a single mutant. All the token usage is computed as the official library of OpenAI, Tiktoken[6], which is the standard practice for estimating model usage.

From the table, we observe substantial differences across prompting strategies. BugFarm uses relatively short prompts (200 input tokens on average), but its output token usage is high (over 360 tokens) because it returns the *code of the entire mutated method*. Consequently, its overall token consumption needs over 2.7M input tokens and about 5M output tokens. In contrast, LLMorpheus requires the largest number of input tokens (370 on average, 7.9M in total), as its prompt includes up to 200 lines of surrounding context, yet it produces moderate output (around 74 tokens on average, 1.5M in total). Moreover, LLMut requires a moderate amount of input and output tokens, which cost 126 input tokens and 58-59 output tokens to generate one mutant. In total, LLMut needs 4.3M input tokens and 1.8M-1.9M output tokens, which are also moderate. Since LLMut asks the models to generate more mutants within a single conversation and wraps the output mutants directly with JSON tags, the average cost is significantly lower than BugFarm and LLMorpheus. These results demonstrate that various prompt engineering strategies significantly impact token costs.

Answer to RQ3: Rule-based traditional approaches, such as PIT and Major, outperform the LLM-based approaches in time cost. On average, PIT and Major generate a single mutant in just 0.02s and 0.08s, respectively, whereas the best-performing LLM, GPT-4o, requires 1.31s. In terms of token efficiency, prompt engineering plays an important role.

3.4 RQ4: Categorization of Non-Compilable Mutants

To answer this RQ, we first identify the error types of non-compilable mutants and then analyze the surrounding code contexts that are error-prone in generating them.

3.4.1 Categorization of compilation error types. Non-compilable mutants require a compilation step to filter out, which results in wasted computational resources. As mentioned in Section 3.2, the LLM-based approaches generate many non-compilable mutants. This RQ analyzes the types of errors and potential root causes of these non-compilable mutants. Following the setting of the previous steps, we first sample 384 non-compilation mutants from the outputs of LLMut, ensuring the confidence level is 95%, and the margin of error is 5% [12, 50, 54, 77]. From the manual analysis of these non-compilable mutants, we identified nine distinct error types, shown as follows.

(1) *Usage of Unknown Methods*: This type of error is due to invoking unknown methods under the code context.
(2) *Code Structural Destruction*: This type of error is due to illegal code structures, such as unmatched parentheses or braces.

[6]https://github.com/openai/tiktoken

Table 12. Error Types of Non-Compilable Mutations

ID	Error Type	%	ID	Error Type	%
1	Usage of Unknown Methods	27.3%	6	Type Mismatch	7.6%
2	Code Structural Destruction	22.9%	7	Incorrect Initialization	3.1%
3	Incorrect Method Parameters	12.8%	8	Incorrect Location	3.1%
4	Usage of Unknown Variables	11.2%	9	Incorrect Exceptions	2.3%
5	Usage of Unknown Types	9.6%	—	—	—

(3) *Incorrect Method Parameters*: This type of error is due to invoking a method with illegal parameters, including invoking with the wrong number of parameters or mismatched types.
(4) *Usage of Unknown Variables*: This type of error is due to using unknown variables in the code context.
(5) *Usage of Unknown Types*: This type of error is due to using unknown type names in the code context.
(6) *Type Mismatch*: This type of error is due to using illegal types when applying a certain operation, such as using objects to perform arithmetic operations.
(7) *Incorrect Initialization*: This type of error is due to using illegal forms of constructors or deleting necessary initialization statements.
(8) *Incorrect Location*: This type of error is due to a mutant being applied to an incorrect code location.
(9) *Incorrect Exceptions*: This type of error is due to throwing incorrect exceptions.

Table 12 presents the results. From the table, we observe that the most frequent error type, *Usage of Unknown Methods*, accounts for 27.3% of total errors, highlighting the challenge of hallucination in LLM-generated code [30]. The second most common error, *Code Structural Destruction*, makes up 22.9%, indicating that maintaining syntactic correctness remains a persistent difficulty for LLMs. These findings underscore the need for further improvements in LLM-based approaches to enhance their reliability and accuracy in code generation.

3.4.2 Categorizations of code context of non-compilable mutants. To analyze which types of code are prone to causing LLM-based approaches to generate non-compilable mutants, we examined the code locations of all non-compilable mutants generated by all nine mutant generation approaches in Table 4, as shown in Figure 3. For most approaches (except Major and BugFarm), the code locations with *MemberReference* and *MethodInvocation* are the top-2 most-prevalent AST node types. For example, for the LLM-based approaches, more than 30% of non-compilable mutants occur at the location with *MemberReference* or *MethodInvocation*. This is potentially caused by the inherent complexity of these operations, which often involve multiple dependencies and references. If any required method or member is missing or incorrectly specified, it can easily lead to non-compilable mutants. The errors highlight a need for better context-aware mutant generation, ensuring that method calls and member references align with the intended program structure. Additionally, we inspect the deletion mutants rejected by the compiler and find that they account for 1.69% (LLMut (GPT-4o)), 3.4% (LLMut (DS-671b)), 39.99% (LEAM), 0.11% (μBERT), and 16.61% (Major) of all their non-compilable mutants, respectively. Thus, for LLMs, deletion is not the major reason for non-compilation.

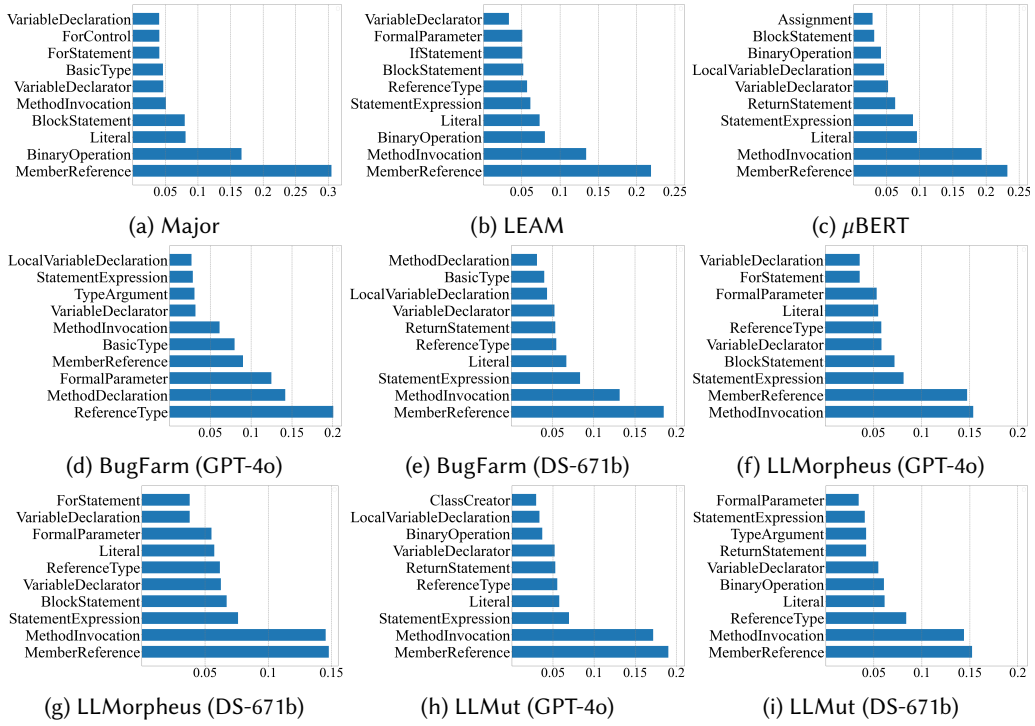


Fig. 3. Distribution of the AST Node Types of Origin Code for Non-Compilable Mutants

Answer to RQ4: The mutants generated by LLM-based approaches have nine main types of compilation errors, with *Usage of Unknown Methods* and *Code Structural Destruction* being the most prevalent. Deletion mutants are not the primary cause of non-compilable code. An analysis of the surrounding code context where LLM-based approaches produce non-compilable mutants reveals that member references and method invocations are the most frequently occurring code features associated with these errors.

3.5 RQ5: Distributions of Surviving Mutants

As shown in Table 4, the mutation scores of rule-based approaches (i.e., PIT and Major) are lower than those of LLM-based approaches, indicating their mutants are harder to detect. Therefore, we intend to analyze the reasons for surviving mutants. To address this RQ, we adopt a setting similar to our analysis of equivalent mutants. Specifically, for each approach, we randomly sample its surviving mutants and manually classify them. We employ the categories of surviving mutant causes defined in the study by Just *et al.* [38] as follows.

- (1) *Not covered*, i.e., the mutant is located in code never executed by the test suite. We refer to the mutant set of this category as *NC*.
- (2) *Covered but no state infection*, i.e., the mutant is executed but does not propagate any state change. We refer to the mutant set of this category as *NI*.
- (3) *State infection without detection*, i.e., the mutant alters the program state, but no test oracle observes the deviation. We refer to the mutant set of this category as *ND*.

Table 13. Sampled Surviving Mutants for Analyzing Survival Reasons

Approach	#Sampled	#Eq. Mut.	NC	NI	ND	NC%	NI%	ND%
PIT	96	3	43	27	23	46.24%	29.03%	24.73%
Major	96	4	42	30	20	45.65%	32.61%	21.74%
LEAM	95	7	59	16	13	67.05%	18.18%	14.77%
μBERT	88	8	37	24	19	46.25%	30.00%	23.75%
BugFarm (GPT-4o)	94	9	62	11	12	72.94%	12.94%	14.12%
BugFarm (DS-671b)	94	11	52	15	16	62.65%	18.07%	19.28%
LLMorpheus (GPT-4o)	95	11	26	44	14	30.95%	52.38%	16.67%
LLMorpheus (DS-671b)	95	13	20	47	15	24.39%	57.32%	18.29%
LLMut (GPT-4o)	95	11	53	16	15	63.10%	19.05%	17.86%
LLMut (DS-671b)	95	12	45	16	22	54.22%	19.28%	26.51%

These categories are mutually exclusive, with the following relationships: $NC \cap NI = \varnothing$, $NC \cap ND = \varnothing$, and $NI \cap ND = \varnothing$. This classification enables us to analyze the underlying reasons for surviving mutants and further evaluate the extent to which different approaches contribute to strengthening the test suite. Given a surviving mutant, we first determine whether it is equivalent; if so, it is discarded. For non-equivalent mutants, we then check whether they are uncovered. If a mutant is covered, we further examine whether it alters the program state, i.e., whether any memory values are modified. Finally, the remaining surviving mutants that are covered but do not lead to detectable differences are classified as not detected. This stepwise process ensures that every surviving mutant is uniquely assigned to one category (equivalent, uncovered, not infecting state, or not detected).

As shown in Table 13, the distributions of surviving mutants differ significantly across approaches, where the columns **NC%**, **NI%**, and **ND%** represent the proportions of NC, NI, and ND within all non-equivalent mutants, respectively.

From the table, we can make the following observations. First, BugFarm and LLMut exhibit a substantially higher proportion of NC (i.e., *not covered*) mutants, while LLMorpheus shows a higher proportion of NI (i.e., *not infected*) mutants. For instance, among the surviving mutants generated by BugFarm (GPT-4o) and BugFarm (DS-671b), 79.52% and 72.94% fall into the NC category, respectively. In contrast, the traditional approaches PIT, Major, and μBert exhibit much lower proportions of 46.24%, 45.65%, and 46.25%, respectively. This suggests that LLM-based approaches either tend to generate mutants located in previously untested code regions or, in the case of LLMorpheus, often produce mutations that do not alter the program state. This characteristic highlights the potential of LLM-based approaches in guiding test suite augmentation by directing attention to untested program areas. Notably, a recent study guides LLMs to generate unit test cases to kill surviving mutants [28, 84], indicating the usefulness of these mutants for other applications. Second, traditional approaches, especially rule-based methods such as PIT and Major, produce a larger fraction of NI (i.e., *not infected*) and ND (i.e., *not detected*) mutants compared to LLM-based methods. For instance, PIT and Major exhibit NI proportions of 29.03% and 32.61%, respectively, whereas both LLMut and BugFarm across different models remain below 20%. These types of surviving mutants require not only generating inputs that execute the mutated code but also propagating and detecting subtle semantic differences, which is inherently more challenging than increasing coverage. Therefore, the overall implication is that LLM-generated surviving mutants are more directly beneficial for strengthening unit tests, as they emphasize extending coverage, while rule-based approaches tend to stress the harder task of state propagation and detection.

Table 14. Comparative Evaluation of Different Prompts for LLMut

Metric	LLMut (GPT-4o)					LLMut (DS-671b)				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
Mut. #	3696	3692	3781	3807	3788	3878	3844	3851	3834	3847
Mut. Score	0.841	0.801	0.773	0.832	0.785	0.776	0.753	0.765	0.760	0.750
Real Bug Detec.	83.1%	83.9%	85.0%	83.1%	84.7%	83.1%	84.7%	85.5%	83.9%	85.8%
Avg. Ochiai	64.8%	65.0%	66.4%	64.8%	65.3%	66.4%	65.8%	67.5%	66.4%	65.6%
Coupling Rate	47.9%	49.7%	52.2%	48.4%	52.1%	49.1%	50.0%	53.2%	50.9%	51.7%
Compilability Rate	78.4%	81.6%	80.1%	77.4%	82.5%	79.5%	82.0%	82.8%	78.0%	83.5%
Dup. Mut. Rate	7.7%	7.9%	7.9%	8.7%	8.1%	8.2%	7.0%	7.3%	8.5%	6.4%
Avg. Gen. Time	1.607	1.584	1.389	1.017	1.282	2.596	2.601	2.573	2.488	2.571
Avg. # Token	392	218	105	44	94	392	219	104	43	92

Note: **P3** is the prompt used in the previous section. The prompt template that achieves the best performance is highlighted in grey background color.

Answer to RQ5: Compared with rule-based approaches, LLM-based approaches tend to generate mutants in code lines that are not covered or not exercised by existing test suites, providing valuable guidance for enhancing unit tests.

4 Discussion

In this section, we examine the impact of various experiment setups for LLMs and discuss their implications.

4.1 Sensitivity to Experimental Settings

We discuss the experiment setups that may affect the evaluation results, particularly the setups related to LLMs.

4.1.1 Setup of the Prompts. Prompts play a critical role in the performance of LLMs. Therefore, we aim to investigate how different prompts influence mutant generation effectiveness. Due to budget constraints, we selected a subset of 306 bugs from our dataset, comprising 10 randomly sampled bugs from each Defects4J project and all bugs from the ConDefects dataset.

Based on the prompt design of LLMut, we investigate how different sources of contextual information affect mutant generation. For this experiment, we also adopt GPT-4o and DS-671b, and vary the prompt template used for mutation generation. Starting from the prompt that includes the richest contextual information, we progressively remove specific sources of context in the *Context* section. This process yields five distinct prompt variants, summarized as follows:

(1) **Prompt 1 (P1):** P1 is the richest prompt with the code of *the whole Java source file* as the *Context* of the prompt.
(2) **Prompt 2 (P2):** P2 is the prompt with the code of *the focal method* and *its corresponding unit tests* as the *Context* of the prompt.
(3) **Prompt 3 (P3):** P3 is the prompt with the code of *the focal method* as the *Context* of the prompt, which is the default prompt in the previous section, shown as Figure 1.
(4) **Prompt 4 (P4):** P4 is the prompt with the code of the target code element to be mutated as the *Context* of the prompt.
(5) **Prompt 5 (P5):** P5 is the prompt with the code of *the focal method* as the *Context*, but removes all few-shot examples.

Table 14 presents a comparative evaluation of different prompts on two models across the main effectiveness, validity, and efficiency metrics.

Table 15. Performance of Different Context Lengths of LLMut (GPT-4o)

Metric	1-Line Context	2-Line Context	3-Line Context
Real Bug Dectc.	93.3%	91.7%	93.3%
Avg. Ochiai	46.7%	50.0%	48.3%
Coupling Rate	37.3%	36.8%	37.7%
Compilability Rate	80.7%	81.4%	82.1%
Dup. Mut. Rate	4.9%	6.6%	5.3%

For the effectiveness metrics (i.e., real bug detectability, average Ochiai, and coupling rate), on both models, **P3**, the prompt used in the previous section, delivers the best overall performance, demonstrating its strength in generating mutants that are semantically closer to real bugs. **P5**, which removes the few-shot examples, ranks as the second-best prompt overall, suggesting that few-shot examples help guide the model toward generating mutants closer to real bugs. Interestingly, **P1**, the longest prompt with the whole Java file, performs the worst in most cases, indicating that providing more information does not necessarily help LLMs generate more bug-representative mutants. **P2**, which adds unit tests to the context compared with **P3**, shows slightly reduced effectiveness, suggesting that including unit tests does not help the model generate better mutants.

For the validity metrics, different prompts express a similar trend in both models. For the compilability rate, **P5**, the one without real-world bug few-shot examples, achieves the highest compilability rate on both models. This indicates that removing few-shot examples reduces the syntactic complexity of the generated mutations, making them easier for LLMs to produce without introducing compilation errors. For the duplicated mutant rate, **P1** achieves the best performance on GPT-4o, while **P5** performs best on DS-671b, suggesting that duplication is more influenced by model behavior than by the prompts.

For the efficiency costs, the results show a clear trend: prompts with more contextual information consume more time and tokens. For example, **P1** (whole file) incurs the highest cost, followed by **P3** (whole method), while **P5** (only the code snippet to be mutated) is the most efficient. Another interesting finding is that **P1** (whole file) is significantly longer than **P2** (unit tests).

We further explore the performance similarity of different prompts. To measure their similarity in performance, we use the Spearman coefficient and Pearson correlation, as shown in the top part of Table 16. We can see that their mutual similarity exceeds 0.95, which is above the typical threshold of 0.85 [42, 57, 72], and the p-values are all below 0.05. The results suggest that on both GPT-4o and DS-671b, although **P3** achieves the most balanced overall performance, the differences among prompts are not statistically significant.

4.1.2 Experiment Setup of the Context Length. To compare the similarity to real defects, we deliberately select the context around the bug locations for generating mutants for all the approaches. In our previous experiments, the context length is set to three lines around the bug location, so we conduct experiments to compare the effects of two-line and one-line contexts, as shown in Table 15. Similarly, the middle part of Table 16 shows the performance similarity among different contexts. We can see that their mutual similarity exceeds 0.95, and the p-values are all below 0.05. Different context lengths perform similarly, thus validating the setup of our experiment.

4.1.3 Impact of Different Few-Shot Examples. To evaluate how different few-shot examples influence the results, we conduct comparative experiments by varying the few-shot examples used in the default LLMut (GPT-4o) prompt. First, we randomly select 3, 4, 5, 6, 7, 8, 9, and 10 additional examples from QuixBugs, and ensure there is no overlap with the default 6 examples. Additionally, despite the risk of data leakage, we select 6 examples from the Defects4J dataset. The results are presented in Table 17.

Table 16. Performance Similarity of Different LLM Configurations of LLMut

Settings	Spea. Coef	p-value	Pear. Coef	p-value
GPT-4o P3 v.s. GPT-4o P1	0.9500	8.76E-05	0.9969	5.37E-09
GPT-4o P3 v.s. GPT-4o P2	1.0000	0.00E+00	0.9995	8.56E-12
GPT-4o P3 v.s. GPT-4o P4	0.9500	8.76E-05	0.9998	8.32E-14
GPT-4o P3 v.s. GPT-4o P5	1.0000	0.00E+00	0.9999	5.61E-19
DS-671b P3 v.s. DS-671b P1	1.0000	0.00E+00	0.9999	1.51E-30
DS-671b P3 v.s. DS-671b P2	1.0000	0.00E+00	0.9999	4.95E-31
DS-671b P3 v.s. DS-671b P4	1.0000	0.00E+00	0.9999	2.93E-29
DS-671b P3 v.s. DS-671b P5	1.0000	0.00E+00	0.9999	1.08E-31
1-Line vs. 2-Line	0.9999	1.40E-24	0.9989	4.54E-05
2-Line vs. 3-Line	0.9999	1.40E-24	0.9995	1.46E-05
1-Line vs. 3-Line	0.9999	1.40E-24	0.9998	2.99E-06
QB-6a (Default) v.s. QB-6b	1.0000	0.00E+00	0.9999	2.99E-15
QB-6a (Default) v.s. QB-3	1.0000	0.00E+00	0.9999	1.82E-14
QB-6a (Default) v.s. QB-4	1.0000	0.00E+00	0.9999	3.97E-15
QB-6a (Default) v.s. QB-5	1.0000	0.00E+00	0.9999	7.62E-14
QB-6a (Default) v.s. QB-7	1.0000	0.0E+00	0.9999	1.12E-14
QB-6a (Default) v.s. QB-8	1.0000	0.0E+00	0.9999	1.28E-13
QB-6a (Default) v.s. QB-9	1.0000	0.00E+00	0.9999	7.03E-15
QB-6a (Default) v.s. QB-10	1.0000	0.00E+00	0.9999	7.03E-15
QB-6a (Default) v.s D4J-6	1.0000	0.00E+00	0.9999	9.81E-17

QB-N: Randomly samples N Examples from QuixBug. QB-6a is the set of default examples, while QB-6b randomly samples 6 other examples. D4J-6 randomly samples 6 examples from Defects4J.

Table 17. Impact on Different Few-Shot Examples by LLMut (GPT-4o)

Metric	QB-6a	QB-3	QB-4	QB-5	QB-6b	QB-7	QB-8	QB-9	QB-10	D4J-6
Real Bug Detec	0.933	0.900	0.917	0.933	0.917	0.900	0.883	0.933	0.933	0.933
Avg. Ochiai	0.483	0.483	0.450	0.417	0.450	0.450	0.433	0.467	0.483	0.483
Coupling Rate	0.379	0.379	0.385	0.380	0.373	0.362	0.348	0.367	0.352	0.388
Comp. Rate	82.1%	82.4%	80.8%	81.7%	81.6%	81.4%	81.8%	82.2%	81.4%	83.1%
Dup.Mut.Rate	5.3%	5.6%	6.4%	5.6%	5.8%	5.6%	7.5%	6.4%	6.2%	5.8%
Avg. # Token	106	98	102	103	106	108	109	110	113	107

QB-N: Randomly samples N Examples from QuixBug. QB-6a is the set of default examples, while QB-6b randomly samples 6 other examples. D4J-6 randomly samples 6 examples from Defects4J.

We further compute their similarity, shown as the bottom part of Table 16. We find that their similarity exceeds 0.95, significantly exceeding the typical threshold of 0.85, and the p-values are all below 0.05. Therefore, despite variations in few-shot examples, there is no statistically significant evidence indicating superior performance among the prompts.

4.1.4 Performance of Using the Same Number of Mutants. In Section 3, we did not restrict the number of mutants generated by each approach. To study the performance of each approach under fixed quantity conditions, we followed the setting of the existing studies [27, 74] and limited the number of mutants generated by all methods to the minimum produced by μBERT [14], which is 11397. This ensures a fair comparison across approaches by controlling for differences in mutant counts, thereby eliminating the potential bias that methods generating more mutants might appear to perform better simply due to quantity rather than quality. For counts exceeding this number, we randomly selected the specified number of mutants for analysis. We conducted 10 rounds of random sampling comparisons, and the results are presented in Table 18.

Under a fixed number of mutants, we observe that LLMut (DS-671b) continues to deliver the best performance in both real bug detection rate and average Ochiai coefficient, with LLMut (GPT-4o) following closely. LLMorpheus (DS-671b) and LLMorpheus (GPT-4o) achieve the highest coupling rates. In terms of validity metrics (i.e., compilability rate and duplicate mutant rate), BugFarm (DS-671b) performs the best among all LLM-based approaches. These findings indicate that, when

Table 18. Performance Under the Same Number of Mutations among All Studied Approaches

Name	R. B. Detec.	Avg. Ochiai	Coup. Rate	Comp. Rate	Dup. Rate
PIT	31.6%	12.6%	17.4%	–	–
Major	68.8%	22.2%	45.4%	96.9%	0.0%
LEAM	64.6%	14.1%	37.2%	54.1%	2.6%
μBERT	66.0%	19.0%	43.6%	26.1	1.9%
BugFarm (GPT-4o)	58.6%	17.7%	47.9%	74.2%	7.5%
BugFarm (DS-671b)	58.9%	17.9%	48.0%	77.4%	4.6%
LLMorpheus (GPT-4o)	67.5%	23.6%	50.2%	68.2%	7.3%
LLMorpheus (DS-671b)	68.5%	24.9%	51.3%	67.8%	6.8%
LLMut (GPT-4o)	87.4%	36.0%	48.8%	75.5%	7.7%
LLMut (DS-671b)	89.2%	39.4%	49.4%	76.3%	7.7%

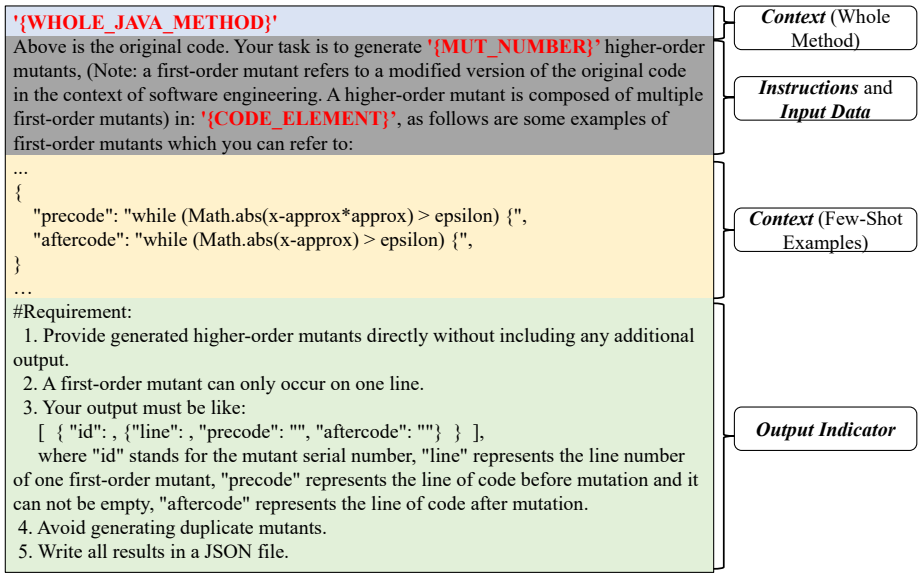


Fig. 4. The Modified Prompt Template for Higher-Order Mutant Generation

Table 19. Comparison Between First-Order Mutants and High-Order Mutants

Metric	LLMut (GPT-4o)		LLMut (DS-671b)	
	FOM	HOM	FOM	HOM
Real Bug Detec.	0.933	0.933	0.950	0.933
Avg. Ochiai	0.483	0.483	0.500	0.483
Coupling Rate	0.379	0.507	0.377	0.506
Compilability Rate	82.1%	70.2%	84.1%	73.3%
Dup. Mut. Rate	5.3%	3.5%	5.1%	3.1%

Note: The one achieving the best performance in terms of a metric is highlighted in a grey background color.

controlling for the number of generated mutants, **different LLM-based approaches (e.g., LLMut, LLMorpheus, and BugFarm) exhibit distinct strengths rather than a single universally superior approach**.

```
    // Mutant-1
-     Size2D size = this.rightBlock.arrange(g2, c4);
+     Size2D size = this.leftBlock.arrange(g2, c4);
    // Mutant-2
-     else {
-       g2.setStroke(getItemOutlineStroke(row, column, selected));
-     }
+     else if (seriesCount == 1) {
+       g2.setStroke(getItemOutlineStroke(row, column, selected));
+     }
```

Listing 1. Two Example Mutations of GPT on Chart-1

4.2 Effectiveness on Higher-Order Mutant Generation

Previously, we only evaluated LLMs in generating first-order mutants (FOMs). To explore the effectiveness of LLMs in generating higher-order mutants (HOMs) [33, 86], we conducted a preliminary experiment by evaluating LLMut on the recent mainstream models, including GPT-4o and DS-671b.

We modified the original prompt to explicitly request HOM generation, shown as Figure 4. The prompts directly instruct LLMs to generate HOMs, and each HOM is composed of several first-order mutants.

Table 19 shows the results. For the effectiveness metrics, HOMs generally achieve average Ochiai scores and higher coupling rates across these models, indicating closer semantic alignment with real bugs. However, FOMs achieve higher real bug detectability on all models. For the validity metrics, HOMs exhibit lower duplicate mutant rates, suggesting improved mutant diversity. However, FOMs exhibit significantly higher compilability rates, which aligns with the expectation that larger code changes are more prone to introducing compilation errors. Our preliminary experiments suggest that strengthening LLM-based HOM generation is promising, as HOMs are semantically closer to real bugs and face more challenges in maintaining syntactic correctness.

4.3 Implications

Based on our findings, we discuss the pros and cons of traditional and LLM-based mutant generation approaches and suggest potential improvements.

4.3.1 Advantages of LLM-based Approaches. Our study demonstrates that LLM-based approaches can generate high-quality mutants for mutation testing at an acceptable cost. The key advantage of LLM-based approaches lies in their ability to produce mutants with stronger coupling relationships with real-world bugs, which is crucial for improving test effectiveness.

Additionally, LLM-based approaches could generate syntactically diverse mutants compared with traditional approaches, which is potentially the reason for generating high-coupling mutants. For example, Listing 1 presents two compilable mutants generated by GPT for Chart-1, which are theoretically beyond the capabilities of traditional approaches. In `Mutant-1`, the LLM infers a relationship between variables (`rightBlock` implies the existence of `leftBlock`) and modifies the variable accordingly. This mutant is not killed by the original bug-triggering test, highlighting its potential for guiding test enhancement. Without natural language understanding (e.g., recognizing the correspondence between "right" and "left"), traditional approaches, both rule-based approaches and the approaches based on small-scale models, struggle to generate such context-aware mutants. In `Mutant-2`, the LLM transforms an `else` branch into an `else if` condition by synthesizing a logical condition using the integer variable `seriesCount` from the context, demonstrating an ability to introduce structurally meaningful changes that traditional approaches cannot achieve.

Additionally, our study explores how different prompt designs, model choices, and contextual information influence mutant generation performance. We find that selecting models with stronger code generation capabilities (e.g., GPT-4o) and designing prompts carefully are critical to achieving optimal results.

Moreover, our study demonstrates that LLMs have clear potential in generating HOMs. Compared with first-order mutants (FOMs), HOMs generally achieve higher coupling rates and average Ochiai scores, suggesting that they are semantically closer to real bugs, while suffering from lower compilability rates and reduced real bug detectability. The results highlight that strengthening LLM-based HOM generation is therefore a promising research direction.

4.3.2 Advantages of Traditional Approaches. Despite the advantages of LLM-generated mutants, traditional approaches still exhibit strengths in reliability and efficiency. They produce fewer non-compilable and duplicate mutants, with PIT achieving a 100% compilability rate and a 0% duplicate mutant rate, while Major follows closely with a 97.6% compilability rate and 0% duplicate mutant rate. In contrast, LLM-based approaches exhibit significantly lower compilability and higher duplicate mutant rates. Additionally, overall, traditional rule-based approaches generate a slightly lower equivalent mutant rate, further avoiding redundant costs. This suggests that traditional approaches remain highly practical when strict compilability and low redundancy are prioritized.

4.3.3 Challenges and Future Directions. Although LLMs demonstrate superior behavioral similarity to real bugs, our findings indicate that there is still substantial room for improvement. One major limitation is the high rate of non-compilable mutants, which limits their practical application in mutation testing. Our analysis reveals that common causes include syntactic errors in generated expressions, incorrect method invocations, and type mismatches. To address these challenges, future research should explore the integration of LLM-based approaches with program analysis and repair techniques to refine mutant generation. Specifically, leveraging static analysis to filter or adjust syntactically incorrect mutants before execution can enhance their reliability.

Moreover, exploring the applications of LLM-generated mutants in the downstream tasks, such as mutation-based test case prioritization (MBTCP) and mutation-based fault localization (MBFL). Our study has shown that LLM-generated mutants often have a higher similarity with real bugs (e.g., have larger Ochiai coefficients), highlighting the potential of adopting them for these tasks.

Another important direction is cross-language generalization. While our study primarily focuses on Java, mutation testing is equally critical in languages such as C/C++, Python, JavaScript, and Rust. Traditional mutation testing tools usually require language-specific implementations, leading to duplicated engineering efforts and limited portability. In contrast, modern LLMs possess inherent multilingual capabilities, as they are pre-trained on large-scale, cross-language code corpora. This enables them to naturally transfer mutation strategies across programming languages without retraining from scratch. Future work should investigate how well LLMs can exploit this cross-language knowledge, and how prompts, training data, or filtering strategies can be adapted to different language paradigms (e.g., strongly typed vs. dynamically typed). Advancing towards multilingual mutant generation would not only broaden the applicability of LLMs in software testing but also provide stronger evidence for their robustness in heterogeneous, real-world development environments.

4.4 Limitations of Our Study

Our study has the following limitations that are left for further exploration.

First, while we evaluated a comprehensive set of static and behavioral metrics, we did not investigate more advanced dynamic indicators. For instance, the phenomenon of *subsuming mutants* [18, 22], where one mutant is always killed by any test that kills another, has been studied

in the mutant testing literature as a way to assess better redundancy and the true fault-detection power of test suites. Incorporating such dynamic analyses would provide deeper insights into the effectiveness of LLM-generated mutants beyond the metrics reported in this paper. Additionally, while in the context of test suite enhancement, two mutants with identical failing test sets may be considered redundant, in mutation-based fault localization [82], the number of mutants per statement directly influences suspiciousness scores, since these are typically computed as averages across all associated mutants.

Second, our experiments are limited to the Java programming language. Although Java is the most widely studied language in mutation testing, it does not cover the diversity of programming paradigms. Mutant behaviors may differ significantly in languages such as C/C++, Python, JavaScript, and Rust, due to differences in type systems, memory models, and runtime environments. Future research should examine the generalizability of LLM-based mutant generation across multiple languages. Notably, modern LLMs are inherently multilingual, trained on large-scale cross-language corpora, which makes cross-language mutant generation a promising direction for broadening the applicability of our findings.

Third, our study relies on fixed few-shot examples, which restricts LLMs to producing context-specific mutants for the given code snippets. Moreover, constructing few-shot examples requires manual effort and may introduce unintended biases into the model's behavior during inference. In addition, we impose no restrictions on the mutation operators applied by LLMs, whereas rule-based approaches rely on predefined and relatively simple operators. Consequently, it is still an open question how LLMs would perform if constrained to the same operator set as rule-based approaches.

5 Threats to Validity

In this section, we illustrate the threats to the validity of our study.

The selected programming language, mutant generation approaches, datasets, and LLMs could be a threat to the validity of our results. To mitigate this threat, we employ the most widely studied mutation testing approaches as baselines, including learning-based and rule-based, the most widely used language (i.e., Java), and the most widely used dataset Defects4J. Moreover, our study further covers LLM-based mutant generation approaches (i.e., BugFarm and LLMorpheus), and also proposes LLMut, a novel prompt engineering approach for mutant generation. All LLM-based approaches are equipped with the recent mainstream models, GPT-4o and DS-671b.

The metrics we used may threaten the conclusion of our study. Each metric has inherent limitations. For instance, *Real Bug Detection Rate* and *Coupling Rate* capture only partial aspects of similarity to real bugs: a method may detect many bugs but with weak coupling on each bug, or achieve strong coupling concentrated on a few bugs. Likewise, *AST Edit Distance* reflects structural differences, but a higher syntactic distance does not necessarily imply larger semantic changes. Additionally, the measurement of *Average Generation Time* may be affected by external factors such as GPU type, service provider implementation, and network fluctuations, introducing unavoidable noise. To mitigate the bias of individual metrics, we report a comprehensive set of complementary measures. Particularly, besides time cost, we also report token consumption, which provides a more stable and model-independent estimate of computational and financial cost.

Another validity threat may be due to data leakage, i.e., the fact that the data in Defects4J [39] may be covered in the training set of the studied LLMs. To mitigate this threat, we employed another dataset, ConDefects [87], which includes programs and faults that were made after the release time of most LLMs we use (except DeepSeek-V3-671b) and thus have limited data leakage risk. Additionally, to increase confidence in our results, we also checked whether the tools can introduce exact matches (syntactically) with the studied faults. We hypothesize that, in case the

tools have been tuned based on specific fault instances, the tools would introduce at least one mutant that is an exact match with the faults we investigate. On Defects4J, the results are LLMut (GPT-4o) (63), LLMut (DS-671b) (72), BugFarm (GPT-4o) (59), BugFarm (DS-671b) (66), LLMorpheus (GPT-4o) (26), LLMorpheus (DS-671b) (31), LEAM (287), μBert (12), and Major (22), respectively. On ConDefects, the results are LLMut (GPT-4o) (25), LLMut (DS-671b) (35), BugFarm (GPT-4o) (22), BugFarm (DS-671b) (29), LLMorpheus (GPT-4o) (19), LLMorpheus (DS-671b) (28), LEAM (56), μBert (5), and Major (92), respectively. The comparison indicates that on Defects4J, LEAM tends to produce significantly more exact matches than the other approaches, while μBert yields significantly fewer exact matches, indicating a minimal or no advantage for all these approaches (except for LLM-based approaches with GPT-4o and LEAM) due to exact matches (in the case of Defects4J). Perhaps more interestingly, on the ConDefects dataset, which has not been seen by most of the tools, Major has the highest number of exact matches, indicating a minor influence of any data leakage on the reported results.

The different experimental settings may also threaten the validity of our results. To address the threat, we explore the impacts of prompt design, context length, few-shot examples, and mutant numbers on the performance of LLMs. The results show that different settings are highly similar. One potential threat to validity is that the metrics used in this study may not fully reflect the actual bug detection in practice. Our evaluation is conducted on the fixed versions of programs, rather than the faulty versions where real-world testing would typically be applied, which may introduce potential differences. In other words, our study operates under the *clean program assumption*, which may not fully capture the dynamics of bug detection in practical scenarios. However, to mitigate this threat, we use the widely adopted Defects4J 2.0 dataset, which consists of real bugs from well-maintained open-source projects, ensuring the relevance of our findings to real bugs.

The subjective nature of human decisions in labeling for equivalent mutants, non-compilation errors, and surviving mutants is another potential threat. To mitigate this threat, we follow a rigorous annotation process where two co-authors independently annotated each mutant. The final Cohen's Kappa coefficient indicates a relatively high level of agreement between the two authors.

6 Related Work

Mutation testing is a systematic testing approach that aims to guide the generation of high-utility test suites. It works by introducing syntactic modifications (i.e., *mutants*) into the source code of the programs under test. By observing if test suites can detect these changes [15, 25], one can identify weaknesses of their test suites, i.e., mutants that are not killed. Besides testing, mutation analysis is a tool that can support multiple applications in software engineering [50, 70]. Most commonly, mutants are employed as substitutes for real-world bugs [3, 11, 40, 58], to guide fault localization [56, 63, 82], test prioritization [71] and program repair [26, 89, 90].

All these tasks require high-utility mutants [8]. This is because it is required to compile and execute every mutant with tests, which is computationally expensive. Therefore, generating redundant or non-compilable mutants, as done by most of the existing learning-based approaches, increases the computational cost involved [22, 81, 83, 92, 93]. Additionally, when mutants are used as substitutes for real defects, the mutants must be as syntactically close to the real bugs as possible, making the mutants more natural [27, 29, 37]. Moreover, when performing tasks such as fault localization [56, 63] and program repair [21, 35, 47, 80, 91], the behavior of mutants must be coupled with real faults/bugs to make these techniques effective.

Traditional mutation testing approaches generate mutants by pre-defined program transformation rules (i.e., *mutation operators*), which are referred to as *rule-based*. For example, one widely used mutation operator is to change one arithmetic binary operator with another one (e.g., $a+b \mapsto a-b$). During the last two decades, the majority of existing mutation testing approaches have been

rule-based [2, 34]. Researchers primarily focus on approaches in Java (e.g., PIT [9], Major [41], JavaLanche [69], MuJava [52], IBIR [43], MIN [95], etc.) and C/C++ (e.g., Proteum [55], Milu [32], WinMut [81], etc.), leveraging the mature ecosystems and tooling chains of these languages for mutation testing.

Recently, with the emergence of machine learning techniques, researchers proposed using deep learning models to generate mutants. For instance, DeepMutation [77] utilizes machine sequence-to-sequence neural translation methods, while LEAM [74] employs learning-guided rule-based expansion. LEAM++, the extension of LEAM, further explores learning-based mutation selection [73]. However, these approaches suffer from challenges in generating syntactically correct mutants and mutants with diverse forms, often failing to produce syntactically valid mutants. Perhaps the closest to our work is that of Degiovanni *et al.*, who proposed μBERT [14]. μBERT generates mutants by masking the code elements to be mutated and then predicting the mutants via BERT [17].

Several recent studies have also explored the use of LLMs for mutant generation. Tip *et al.* concurrently investigated the use of LLMs for mutant generation in JavaScript [76], primarily focusing on the feasibility of leveraging LLMs to produce mutants. In contrast, our study conducts a broader comparison by evaluating LLMs against a diverse set of baselines, including both rule-based and learning-based approaches. Furthermore, we explore different prompt strategies, such as incorporating few-shot examples and enforcing structured JSON output, and we systematically examine multiple prompt templates in the context of mutation testing. Similarly, Endres *et al.* employed large models for mutant generation to aid deduction [19], while Deb *et al.* used GPT as one of the baselines in their approach [13]. In contrast to their projects, our work systematically explores a variety of LLMs and prompt designs, offering a thorough comparison with state-of-the-art approaches. Dakhel *et al.* [10] and Ibrahimzada *et al.* [31] also proposed generating artificial bugs using LLMs. These approaches mainly evaluate from a certain perspective, such as mutation score or program repair. BugFarm [31] intends to generate artificial bugs that are hard to kill, by first choosing code locations based on a trained small model and then mutating the code via an LLM. Compared with these approaches, our study is more comprehensive and evaluates existing mutant generation approaches from many aspects. Specifically, we measure the behavioral similarity between real bugs and mutants using established metrics, evaluate the syntactic diversity of mutants, and analyze the non-compilable mutants produced by LLMs. Our findings illustrate both the strengths and limitations of LLMs for mutant generation, and highlight directions for future improvement.

7 Conclusion

In this paper, we systematically investigate the performance of existing mutant generation approaches across multiple dimensions, including their effectiveness, validity, and efficiency. Our study conducts extensive and detailed comparative experiments, covering both widely studied rule-based mutant generation approaches and LLM-based methods. Particularly, we evaluate not only existing LLM-based approaches but also our new prompt for mutant generation. Our evaluation results highlight that LLMs significantly outperform traditional approaches in generating diverse mutants that closely mimic real bugs. However, the results also expose key limitations of LLMs, particularly their tendency to generate a high proportion of non-compilable mutants. Therefore, we analyze the error types of the LLM-generated non-compilable mutants and identify that member assessment and method invocation are more likely to lead LLMs to generate non-compilable mutants. Based on our findings, we emphasize the need for further research to enhance LLM-based mutation testing, addressing its current shortcomings while leveraging its strengths.

Our implementation and the experimental data are available at our artifact: **https://github.com/da-123-snake/Mutantion-Generation-Study**.

Acknowledgments

We are grateful to the anonymous reviewers. This work was supported by the Fundamental Research Funds for the Central Universities (Grant No. 2025JBMC006) and the National Natural Science Foundation of China under Grant No. 62202040.

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. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] doi:10.48550/arXiv.2303.08774

[2] Domenico Amalfitano, Ana CR Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. 2022. How do Java mutation tools differ? *Commun. ACM* 65, 12 (2022), 74–89. doi:10.1145/3526099

[3] 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. doi:10.1145/1062455.1062530

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 268–277. doi:10.1109/ICSE-SEIP52600.2021.00036

[5] Timothy A Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta informatica* 18, 1 (1982), 31–45. doi:10.1007/BF00625279

[6] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empir. Softw. Eng.* 25, 1 (2020), 434–487. doi:10.1007/s10664-019-09778-7

[7] Mingda Chen, Bo Wang, Youfang Lin, and Jie M Zhang. 2025. CAMUS: Context-Aware Neural Mutation Selection. In *2025 32nd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 11–22. doi:10.1109/APSEC66846.2025.00013

[8] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2025. Deep learning-based software engineering: progress, challenges, and opportunities. *Science China Information Sciences* 68, 1 (2025), 111102. doi:10.1007/s11432-023-4127-5

[9] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for Java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452. doi:10.1145/2931037.2948707

[10] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468. doi:10.1016/j.infsof.2024.107468

[11] Murial Daran and Pascale Thévenod-Fosse. 1996. Software error analysis: A real case study involving real faults and mutations. *ACM SIGSOFT Software Engineering Notes* 21, 3 (1996), 158–171. doi:10.1145/226295.226313

[12] Rafael Maiani de Mello, Pedro Corrêa Da Silva, and Guilherme Horta Travassos. 2015. Investigating probabilistic sampling approaches for large-scale surveys in software engineering. *Journal of Software Engineering Research and Development* 3 (2015), 1–26. doi:10.1186/s40411-015-0023-0

[13] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. 2024. Syntax Is All You Need: A Universal-Language Approach to Mutant Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 30 (July 2024), 21 pages. doi:10.1145/3643756

[14] Renzo Degiovanni and Mike Papadakis. 2022. μBert: Mutation testing using pre-trained language models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 160–169. doi:10.1109/ICSTW55395.2022.00039

[15] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623343

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186. doi:10.18653/V1/N19-1423

[18] João P Diniz, Chu-Pan Wong, Christian Kästner, and Eduardo Figueiredo. 2021. Dissecting strongly subsuming second-order mutants. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 171–181.

doi:10.1109/ICST49551.2021.00028

[19] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912. doi:10.1145/3660791

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324. doi:10.1145/2642937.2642982

[21] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27. doi:10.1145/3418461

[22] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2022. Cerebro: Static subsuming mutant selection. *IEEE Transactions on Software Engineering* 49, 1 (2022), 24–43. doi:10.1109/TSE.2022.3140510

[23] Gregory Gay and Alireza Salahirad. 2023. How Closely are Common Mutation Operators Coupled to Real Faults?. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 129–140. doi:10.1109/ICST57152.2023.00021

[24] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the potential of ChatGPT in automated code refinement: An empirical study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, Article 34, 13 pages. doi:10.1145/3597503.3623306

[25] Richard G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering* 4 (1977), 279–290. doi:10.1109/TSE.1977.231145

[26] Carol Hanna, Aymeric Blot, and Justyna Petke. 2025. Reinforcement learning for mutation operator selection in automated program repair. *Automated Software Engineering* 32, 2 (2025), 1–33. doi:10.1007/s10515-025-00501-z

[27] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. 2019. Comparing mutation testing at the levels of source code and compiler intermediate representation. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 114–124. doi:10.1109/ICST.2019.00021

[28] Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. Mutation-Guided LLM-based Test Generation at Meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 180–191. doi:10.1145/3696630.3728544

[29] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131. doi:10.1145/2902362

[30] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2, Article 42 (2025), 55 pages. doi:10.1145/3703155

[31] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2025. Challenging Bug Prediction and Repair Models with Synthetic Bugs. In *2025 IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*. IEEE, 133–144. doi:10.1109/SCAM67354.2025.00021

[32] Yue Jia and Mark Harman. 2008. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference-Practice and Research Techniques (taic part 2008)*. IEEE, 94–98. doi:10.1109/TAIC-PART.2008.18

[33] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393. doi:10.1016/j.infsof.2009.04.016

[34] 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. doi:10.1109/TSE.2010.62

[35] Jiajun Jiang, Fengjie Li, Zijie Zhao, Zhirui Ye, Mengjiao Liu, Bo Wang, Hongyu Zhang, and Junjie Chen. 2025. Boosting Redundancy-based Automated Program Repair by Fine-grained Pattern Mining. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 85–97. doi:10.1109/ICSME64153.2025.00018

[36] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1251–1263. doi:10.1109/ICSE48619.2023.00111

[37] Matthieu Jimenez, Thiery Titcheu Checkam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. 2018. Are mutants really natural? a study on how" naturalness" helps mutant selection. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10. doi:10.1145/3239235.3240500

[38] René Just, Michael D Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 315–326. doi:10.1145/2610384.2610388

[39] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440. doi:10.1145/2610384.2628055

[40] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665. doi:10.1145/2635868.2635929

[41] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 612–615. doi:10.1109/ASE.2011.6100138

[42] Samuel J Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing mutants to guide mutation testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1743–1754. doi:10.1145/3510003.3510187

[43] Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2023. IBiR: Bug-report-driven fault injection. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–31. doi:10.1145/3542946

[44] 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. doi:10.1145/3510417

[45] 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. doi:10.1109/TSE.2017.2684805

[46] Benjamin Kushigian, Samuel J Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 654–665. doi:10.1145/3650212.3680310

[47] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2011), 54–72. doi:10.1109/TSE.2011.104

[48] 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. doi:10.1109/ASE56229.2023.00089

[49] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56. doi:10.1145/3135932.3135941

[50] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 233–244. doi:10.1145/3106237.3106275

[51] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] doi:10.48550/arXiv.2412.19437

[52] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133. doi:10.5555/1077303.1077304

[53] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 883–883. doi:10.1145/3597503.3639150

[54] Finlay Macklon, Markos Viggiato, Natalia Romanova, Chris Buzon, Dale Paas, and Cor-Paul Bezemer. 2023. A Taxonomy of Testable HTML5 Canvas Issues. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3647–3659. doi:10.1109/TSE.2023.3270740

[55] José Carlos Maldonado, Márcio Eduardo Delamaro, SCPF Fabbri, A Silva Simao, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. 2001. Proteum: A family of tools to support specification and program testing based on mutation. *Mutation testing for the new century* 24 (2001), 113–116. doi:10.1007/978-1-4757-5939-6_19

[56] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162. doi:10.1109/ICST.2014.28

[57] Manish Motwani and Yuriy Brun. 2023. Better Automatic Program Repair by Using Bug Reports and Tests Together. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1225–1237. doi:10.1109/ICSE48619.2023.00109

[58] Akbar Siami Namin and Sahitya Kakarla. 2011. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 342–352.

doi:10.1145/2001420.2001461

[59] A Jefferson Offutt and Roland H Untch. 2001. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century* (2001), 34–44. doi:10.1007/978-1-4757-5939-6_7

[60] Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Syntactic Versus Semantic Similarity of Artificial and Real Faults in Mutation Testing Studies. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3922–3938. doi:10.1109/TSE.2023.3277564

[61] 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 *ICSE*. 936–946. doi:10.1109/ICSE.2015.103

[62] 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. doi:10.1016/bs.adcom.2018.03.015

[63] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628. doi:10.1002/stvr.1509

[64] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 906–918. doi:10.1145/3468264.3468623

[65] Goran Petrović and Marko Ivanković. 2018. State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 163–171. doi:10.1145/3183519.3183521

[66] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *Proceedings of the 43rd International Conference on Software Engineering*. IEEE, 910–921. doi:10.1109/ICSE43902.2021.00087

[67] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3900–3912. doi:10.1109/TSE.2021.3107634

[68] Hannah Potter, Ana CR Paiva, Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and René Just. 2025. Evaluating the Impact of Scaffolding and Visualizations for Mutation Testing Exercises in Software Engineering Education. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 823–834. doi:10.1145/3696630.3727240

[69] David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *ESEC/FSE*. 297–298. doi:10.1145/1595696.1595750

[70] 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. doi:10.1145/3293882.3330568

[71] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical evaluation of mutation-based test case prioritization techniques. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1695. doi:10.1002/stvr.1695

[72] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. 2008. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*. 351–360.

[73] Zhao Tian, Junjie Chen, Dong Wang, Qihao Zhu, Xingyu Fan, and Lingming Zhang. 2025. LEAM++: Learning for Selective Mutation Fault Construction. *ACM Transactions on Software Engineering and Methodology* 34, 8, Article 223 (2025), 41 pages. doi:10.1145/3725528

[74] 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. doi:10.1145/3551349.3556949

[75] 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*. 1733–1745. doi:10.1145/3650212.3680395

[76] Frank Tip, Jonathan Bell, and Max Schäfer. 2025. LLMorpheus: Mutation Testing Using Large Language Models. *IEEE Transactions on Software Engineering* 51, 6 (2025), 1645–1665. doi:10.1109/TSE.2025.3562025

[77] 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. doi:10.1109/ICSME.2019.00046

[78] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[79] Bo Wang, Mingda Chen, Youfang Lin, Mark Harman, Mike Papadakis, and Jie M Zhang. 2026. A Comprehensive Study on Large Language Models for Mutation Testing. arXiv:2406.09843 [cs.SE] doi:10.48550/arXiv.2406.09843

[80] Bo Wang, Ming Deng, Mingda Chen, Youfang Lin, Jianyi Zhou, and Jie M Zhang. 2026. Assessing the effectiveness of recent closed-source large language models in fault localization and automated program repair. *Automated Software Engineering* 33, 1 (2026), 1–42. doi:10.1007/s10515-025-00549-x

[81] Bo Wang, Sirui Lu, Yingfei Xiong, and Feng Liu. 2021. Faster mutation analysis with fewer processes and smaller overheads. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 381–393. doi:10.1109/ASE51524.2021.9678827

[82] Bo Wang, Jinkang Wei, Mingda Chen, Chong Chen, Youfang Lin, and Jie M Zhang. 2025. A Systematic Exploration of Mutation-Based Fault Localization Formulae. *Software Testing, Verification and Reliability* 35, 1 (2025), e1905. doi:10.1002/stvr.1905

[83] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster mutation analysis via equivalence modulo states. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 295–306. doi:10.1145/3092703.3092714

[84] Guancheng Wang, Qinghua Xu, Lionel C Briand, and Kui Liu. 2026. Mutation-Guided Unit Test Generation with a Large Language Model. arXiv:2506.02954 [cs.SE] doi:10.48550/arXiv.2506.02954

[85] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing library API misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 866–877. doi:10.1109/ICSE.2019.00093

[86] Chu-Pan Wong, Jens Meinicke, Leo Chen, João P Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. Efficiently finding higher-order mutants. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1165–1177. doi:10.1145/3368089.3409713

[87] Yonghao Wu, Zheng Li, Jie M Zhang, and Yong Liu. 2024. ConDefects: A Complementary Dataset to Address the Data Leakage Concern for LLM-Based Fault Localization and Program Repair. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 642–646. doi:10.1145/3663529.3663815

[88] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971. doi:10.1145/3540250.3549101

[89] Yuan-An Xiao, Chenyang Yang, Bo Wang, and Yingfei Xiong. 2023. ExpressAPR: Efficient patch validation for Java automated program repair systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2038–2041. doi:10.1109/ASE56229.2023.00012

[90] Yuan-An Xiao, Chenyang Yang, Bo Wang, and Yingfei Xiong. 2024. Accelerating patch validation for program repair with interception-based execution scheduling. *IEEE Transactions on Software Engineering* 50, 3 (2024), 618–635. doi:10.1109/TSE.2024.3359969

[91] Yingfei Xiong and Bo Wang. 2022. L2S: A framework for synthesizing the most probable program under a specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–45. doi:10.1145/3487570

[92] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive mutation testing. In *Proceedings of the 25th international symposium on software testing and analysis*. 342–353. doi:10.1145/2931037.2931038

[93] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive Mutation Testing. *IEEE Transactions on Software Engineering* 45, 9 (2018), 898–918. doi:10.1109/TSE.2018.2809496

[94] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 235–245. doi:10.1145/2483760.2483782

[95] Peng Zhang, Zeyu Lu, Yang Wang, Yibiao Yang, Yuming Zhou, and Mike Papadakis. 2025. Enriching mutation testing with innovative method invocation mutation: Filling the crucial missing piece of the puzzle. *IEEE Transactions on Software Engineering* 51, 7 (2025), 2125–2143. doi:10.1109/TSE.2025.3573751

[96] Peng Zhang, Yang Wang, Xutong Liu, Yanhui Li, Yibiao Yang, Ziyuan Wang, Xiaoyu Zhou, Lin Chen, and Yuming Zhou. 2022. Mutant reduction evaluation: what is there and what is missing? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–46. doi:10.1145/3522578

[97] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-aware neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1443–1455. doi:10.1109/ICSE48619.2023.00126

[98] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347. doi:10.1109/TSE.2019.2892102