Boosting LLMs for Mutation Generation

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

LLM-based mutation testing is a promising testing technology, but existing approaches typically rely on a fixed set of mutants as few-shot examples or none at all. This can result in generic low-quality mutants, missed context-specific mutation patterns, substantial numbers of redundant and non-compilable mutants, and limited semantic similarity to real bugs. To overcome these limitations, we introduce SMART (*Semantic Mutation with Adaptive Retrieval and Tuning*). SMART integrates retrieval-augmented generation (RAG) on a vectorized dataset of real-world bugs, focused code chunking, and supervised fine-tuning using mutants coupled with real-world bugs. We conducted an extensive empirical study of SMART using 1,991 real-world Java bugs from the Defects4J and ConDefects datasets, comparing SMART to the state-of-the-art LLM-based approaches, LLMut and LLMorpheus.

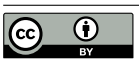
The results reveal that SMART substantially improves mutation validity, effectiveness, and efficiency (even enabling 7B-scale models to match or even surpass large models like GPT-4o). We also demonstrate that SMART significantly improves downstream software engineering applications, including test case prioritization and fault localization. More specifically, SMART improves validity (weighted average generation rate) from 42.89% to 65.6%. It raises the non-duplicate rate from 87.38% to 95.62%, and the compilable rate from 88.85% to 90.21%. In terms of effectiveness, it achieves a real bug detection rate of 92.61% (vs. 57.86% for LLMut) and improves the average Ochiai coefficient from 25.61% to 38.44%. For fault localization, SMART ranks 64 more bugs as Top-1 under MUSE and 57 more under Metallaxis.

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

Additional Key Words and Phrases: Mutation Testing, Mutation Generation, Large Language Models, RAG, Code Chunking, Fine-Tuning

*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; Ming Deng, Beijing Jiaotong University, Beijing, China, 24120317@bjtu.edu.cn; Mingda Chen, Beijing Jiaotong University, Beijing, China, 23120337@bjtu.edu.cn; Chengran Yang, Singapore Management University, Singapore, cryang@smu.edu.sg; Youfang Lin, Beijing Jiaotong University, 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 2994-970X/2026/7-ARTFSE149
https://doi.org/10.1145/3808156

ACM Reference Format:
Bo Wang, Ming Deng, Mingda Chen, Chengran Yang, Youfang Lin, Mark Harman, Mike Papadakis, and Jie M. Zhang. 2026. Boosting LLMs for Mutation Generation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE149 (July 2026), 24 pages. https://doi.org/10.1145/3808156

1 Introduction

Mutation testing [13, 19] is widely regarded as one of the most fundamental dynamic techniques in software testing and program analysis. By systematically introducing artificial faults, known as *mutants*, into the program under test, mutation testing enables researchers and practitioners to assess the adequacy of test suites [25], prioritize effective test suites [6, 16, 33, 44], or localize buggy statements/methods [35, 39, 53]. In recent years, mutation testing has gained widespread adoption in the industry. Google and Meta have reported their successful experience in deploying mutation testing [3, 21, 40]. The effectiveness of mutation testing, however, fundamentally depends on the quality of the generated mutants. Generating effective and diverse mutants is thus critical not only for evaluating test suites, but also for supporting downstream applications such as test case prioritization and fault localization.

Recently, rapid advances in Large Language Models (LLMs) have created new opportunities for mutation testing [7, 57]. Due to their strong code understanding and generation capabilities, LLMs have been explored to automatically generate more realistic and semantically meaningful mutants. Both academic research [12, 23, 48, 50, 56] and industrial practice [21] have begun to investigate LLM-based mutation generation as a promising alternative to traditional approaches. Despite encouraging progress, existing LLM-based approaches still suffer from several major limitations. First, they rely on either a fixed set of few-shot examples as in-context examples or no examples at all, making it difficult for models to adapt to diverse program contexts. For example, BugFarm [23] and LLMorpheus [48] rely solely on prompts, whereas LLMut [50] incorporates a fixed set of real-world bugs as few-shot examples to instruct LLMs. Second, they typically restrict the code context to the surrounding focal method, which blinds the model to crucial program-wide dependencies. All existing LLM-based mutation generation approaches directly adopt the whole focal method as context. Due to these limitations, the generated mutants still contain a substantial proportion of redundant and non-compilable cases, with limited semantic similarity to real bugs [50, 55].

To address these limitations, we present **SMART** (**S**emantic **M**utation with **A**daptive **R**etrieval and **T**uning), a novel LLM-based mutation generation approach that integrates retrieval-augmented generation (RAG) [32], code chunking, and fine-tuning [5]. Our approach is designed to enhance both the relevance and effectiveness of generated mutants. First, to select context-aware few-shot examples, we build a vectorized dataset of real-world bugs and employ RAG to retrieve the most relevant "bug–fix" pairs for the target method. We collect 130,000 real-world single-hunk bugs with corresponding patches, and embed the patched versions as the keys for retrieval. This ensures that the LLM benefits from in-context examples that are semantically aligned with the mutation task at hand. Second, rather than mutating the entire method in a monolithic way, we partition the focal method into function-coherent code chunks and generate mutants at the chunk level, which improves focus and diversity. Finally, to better adapt LLMs to the task of producing semantically effective mutants, we apply supervised fine-tuning (SFT) using effective mutants coupled with real-world bugs. The training set comprises 13,760 mutants that are coupled with real bugs from the prior study [50]. To prevent data leakage, all training mutants are drawn from projects outside the evaluation set. By combining retrieval, structural code decomposition, and model adaptation, SMART aims to advance the state of LLM-based mutation generation significantly.

To assess the quality of SMART-generated mutants, we conducted an extensive empirical study on 1,991 real-world Java bugs, including 701 from Defects4J [27] and 1,290 from ConDefects [62]. We

evaluated SMART using five open-source LLMs (DeepSeek-Coder-6.7B, Llama-3.1-8B, Qwen-2.5-7B, Qwen-2.5-14B, and Qwen-2.5-32B), and additionally tested GPT-4o with RAG and code chunking enabled. For comparison, we included two state-of-the-art LLM-based mutation generation approaches: LLMorpheus [48] and LLMut [50]. Our evaluation covers a wide range of dimensions. We first examine the effectiveness of generated mutants, i.e., how closely they couple with real-world bugs. We then assess their validity, since LLMs inevitably produce redundant or non-compilable mutants. Finally, we evaluate their impact on downstream applications, including mutation-based test case prioritization and fault localization.

The results show that SMART substantially improves both the *validity* and *effectiveness* of LLM-generated mutants. For validity, SMART raises the weighted average generation rate from 42.89% (LLMut) to 65.6% (↑ 52.95%; +22.71 percentage points (PPs)), increases the non-duplicate rate from 87.38% (LLMut) and 85.87% (LLMorpheus) to 95.62% (↑ 9.42%–11.36%), and improves the compilable rate from 88.85% (LLMut) and 78.43% (LLMorpheus) to 90.21% (↑ 1.53%–15.0%). For effectiveness, SMART achieves a weighted average real bug detection rate of 92.61%, compared to 57.86% for LLMut and 31.99% for LLMorpheus, and raises the average Ochiai coefficient to 38.44% (↑ 50.1% and 277.6% over baselines). On downstream applications, SMART enhances mutation-based fault localization by ranking more bugs in Top-k and reducing MAR/MFR (e.g., 143 Top-1 bugs by MUSE vs. 106 for LLMut and 11 for LLMorpheus). Notably, our approach enables 7B-scale models to achieve performance competitive with GPT-4o, making it suitable for scenarios with limited computational resources.

In the discussion section, we further analyze the computational cost and robustness of SMART. Specifically, we report token consumption during mutation generation, as well as the overhead introduced by RAG and fine-tuning. Building the RAG index requires approximately 10 minutes as a one-time offline cost, while retrieval incurs an average overhead of 6.5 seconds per bug when chunking is enabled. We also examine the impact of different similarity metrics used in RAG and compare cosine similarity, dot-product similarity, and Euclidean distance. Our results show that Euclidean distance consistently achieves the best performance on semantic-related metrics and downstream tasks. These analyses provide a more comprehensive understanding of its practical cost and sensitivity to configuration choices.

In summary, our paper makes the following contributions:

- We propose a novel LLM-based mutation generation approach, SMART, which incorporates retrieval-augmented generation (RAG), code chunking, and fine-tuning.
- We conduct an extensive study to evaluate SMART, measuring its mutation validity, effectiveness, and utility in mutation-based test case prioritization and fault localization. The results demonstrate the superiority of SMART over the two state-of-the-art LLM-based approaches.
- Our implementation, the checkpoints of fine-tuned models, and high-quality Java mutants are publicly available.

2 Background

In this section, we introduce the necessary concepts of mutation testing, including metrics for evaluating the quality of mutants and their downstream applications.

2.1 Mutation Testing

Mutation testing is a fault-injection technique that assesses test quality by injecting small syntactic changes, called *mutants*, into the program [13, 19, 25, 38]. A mutant is *killed* if test outputs differ from the original program; otherwise it *survives*, exposing weaknesses in the test suite. The *mutation score*, i.e., the proportion of killed mutants, is a standard measure of test effectiveness.

Traditional mutation testing relies on rule-based operators (e.g., replacing arithmetic or relational operators) [8, 29], which, while effective, often fail to capture the diversity of real faults. To overcome this, learning-based approaches such as DeepMutation [49] and LEAM [45, 46] leverage deep learning to guide mutation generation. More recently, LLMs have emerged as a new direction, offering the potential to generate semantically richer and more realistic mutants. BugFarm [23] leverages LLMs to synthesize hard-to-kill artificial bugs for test evaluation. LLMorpheus [48] explores the feasibility of using LLMs for mutation generation. LLMut [50] systematically studies the validity and effectiveness of LLM-generated mutants in Java.

2.2 Evaluating Mutations

The usefulness of mutation testing largely depends on the quality of the generated mutants. To better characterize this quality, we first categorize the mutants generated by LLMs [50]. Let all LLM-generated mutants be denoted as the set A, which may include invalid mutants. We denote the subset of compilable mutants as C. Within C, two subsets of mutants are useless: duplicate mutants (D), which are syntactically identical to the original program or to other mutants, and equivalent mutants (E), which are syntactically different but semantically indistinguishable from the original program. Formally, $(C \supseteq D) \wedge (C \supseteq E) \wedge (D \cap E = \phi)$. Therefore, the ideal set of useful mutants for mutation testing is $(C - D - E)$, excluding both duplicates and equivalents that do not contribute to exposing weaknesses in the test suite. However, because determining whether a mutant is equivalent is undecidable [4], and the proportion of equivalent mutants is typically negligible in practice ($|E| \ll |C - D|$) [47, 48, 50], we approximate the ideal set of useful mutants by $(C - D)$ for downstream mutation testing applications. For instance, the mutation score MS is calculated as follows:

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

where K is the set of killed mutants.

2.2.1 Validity. Invalid mutants (e.g., non-compilable code or duplicates of the original program) provide little value and may bias evaluation results. Therefore, validity is commonly measured using two indicators: (1) the *compilability rate*, i.e., the proportion of mutants that can be successfully compiled (i.e., $CR = |C|/|A|$); (2) the *duplication rate*, i.e., the proportion of mutants that are syntactically identical to the original program or other existing mutants (i.e., $DR = |D|/|A|$). An effective approach should maximize compilability while minimizing duplication and equivalence.

2.2.2 Effectiveness. The effectiveness of mutants examines whether the introduced changes capture characteristics commonly observed in real bugs. Evaluating effectiveness is crucial, since only mutants that resemble real bugs can provide meaningful insights into the fault-detection capability of a test suite and the practical value of mutation testing. Particularly, for LLM-generated mutants, effectiveness involves examining whether the introduced changes capture fault characteristics that are commonly observed in real defects.

Specifically, mutant effectiveness comprises three aspects. First, the **real-bug detection capability**, which evaluates whether bug-triggering test cases that reveal real faults are also able to detect the generated mutants [28]. Second, the **fault coupling rate**, i.e., the proportion of generated mutants that couple with real bugs. Third, the **semantic similarity** between a mutant and its corresponding real bug, which can be quantified using the **Ochiai coefficient** [37, 50].

Formally, let m and b be a mutant and its corresponding real bug, and let fT_m and fT_b be the sets of failing test cases for m and b, respectively. The Ochiai coefficient between m and b is defined as:

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

A coefficient closer to 1 indicates that the mutant is more semantically similar to the real bug. For a real bug b, its Ochiai coefficient is calculated as the mean coefficient across all de-duplicated mutants $(C_b - D_b)$ generated for it:

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

where C_b and D_b are the corresponding sets C and D of the bug b, respectively. Given a mutation generation approach and a set of bugs B, the overall effectiveness is quantified by the Average Ochiai Coefficient (AOC):

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

2.3 Downstream Applications of Mutation Testing

In this paper, we consider the two widely studied applications of mutation testing, *mutation-based test case prioritization* and *mutation-based fault localization*.

2.3.1 Mutation-Based Test Case Prioritization. Test case prioritization (TCP) aims to schedule the execution order of test cases so that faults can be detected as early as possible, and has been extensively studied in the literature [42, 66].

Mutation-based TCP [16, 33, 44], which leverages information from the kill matrix, has shown its effectiveness compared with coverage-based methods. Recently, the TCP approaches, GRK, GRD, and HYB-ω, have been widely studied [16, 45, 46].

GRK (Greedy Kill) iteratively selects the test case that kills the maximum number of additional mutants, aiming to separate faulty behavior from the original program as early as possible. **GRD** (Greedy Distinguish) iteratively selects the test case that distinguishes the maximum number of additional mutants, where two mutants are distinguished if their execution produces different outputs. Thus, GRD attempts to maximize the diversity of fault-revealing behaviors. **HYB-ω** combines both strategies by selecting, at each step, the test case that maximizes a weighted sum of additionally killed and additionally distinguished mutants. When $\omega = 1$, HYB-ω degenerates to GRK; when $\omega = 0$, it degenerates to GRD.

2.3.2 Mutation-Based Fault Localization. Mutation-based fault localization (MBFL) techniques leverage the behavior of mutants to estimate the suspiciousness of program entities, i.e., their likelihood of being buggy [35, 39, 53, 61, 70]. The intuition is that if the execution of a mutant impacts failing test cases more often than passing ones, the associated statement is likely to be faulty. MUSE [35] and Metallaxis [39] are two widely studied MBFL approaches.

MUSE: MUSE first assigns a suspiciousness score to each mutant m using both failing and passing test cases, shown as follows:

$$S(m) = failed(m) - \frac{f2p}{p2f} \cdot passed(m), \tag{5}$$

where $failed(m)$ and $passed(m)$ denote the number of test cases that fail or pass on the original program but yield the opposite outcome on mutant m. $f2p/p2f$ represents the number of test cases that flip from failing to passing or from passing to failing due to any mutant. The suspiciousness of a statement s is then calculated as the average suspiciousness of all mutants located at s.

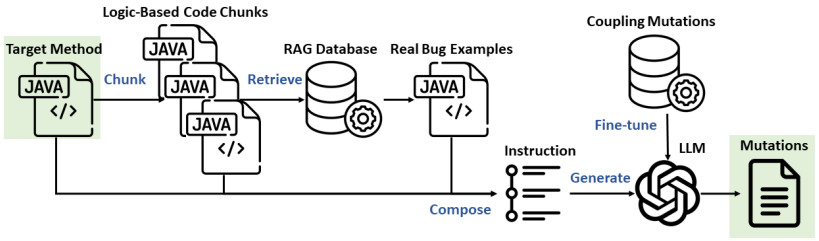


Fig. 1. Overview of SMART

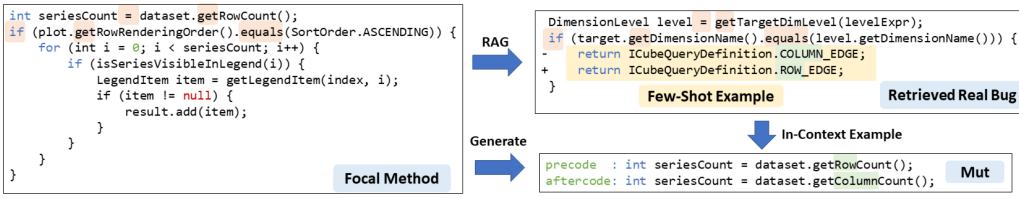


Fig. 2. Example Mutant Generated by SMART

Metallaxis: In contrast, Metallaxis computes suspiciousness by normalizing with respect to the total number of failing test cases, shown as follows:

$$S(m) = \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m))}}. \tag{6}$$

Here, $totalfailed$ is the number of failing test cases on the original program, while $failed(m)$ and $passed(m)$ are defined similarly as in MUSE. The suspiciousness of a statement s is determined as the maximum suspiciousness score among all mutants associated with s.

3 Methodology

In this section, we present SMART, a novel LLM-based mutation generation approach with logic-based code chunking (Section 3.2), RAG-based few-shot example retrieval (Section 3.3), task-specific prompt engineering (Section 3.4), and supervised fine-tuning (Section 3.5).

3.1 Overview

Figure 1 illustrates the overall workflow of SMART, which integrates three key techniques (i.e., logic-based code chunking, retrieval-augmented generation (RAG), and supervised fine-tuning (SFT)), into a unified pipeline.

Given a focal Java method, we first apply logic-based code chunking to partition it into semantically coherent segments, reducing mutation generation complexity and enabling finer-grained reasoning. We then use RAG over a vectorized repository of real-world bug–fix pairs to retrieve context-aware few-shot examples, which are incorporated into the prompt to guide mutant generation. In parallel, we further adapt LLMs through supervised fine-tuning on mutants semantically coupled with real-world bugs. Finally, the fine-tuned model generates chunk-level mutants, which are aggregated into a candidate set for further evaluation.

Figure 2 illustrates an example of SMART. Given a code chunk from the focal method (left), our RAG component retrieves a real bug (right) whose code is similar to the chunk: both contain (1) an assignment to a local variable, and (2) an `if`-condition composed of chained `get*()`

and `equals()` calls. In the retrieved bug, the fix swaps the semantic roles of `COLUMN` and `ROW` (`COLUMN_EDGE` → `ROW_EDGE`). We feed this instance as a few-shot example to the LLM, which then produces a semantically analogous mutant for the target chunk, which changes `int seriesCount = dataset.getRowCount()` to `int seriesCount = dataset.getColumnCount()`. This mutant captures a realistic *row/column* semantic swap rather than a superficial token edit, exposing tests that implicitly assume a specific orientation (*row* vs. *column*). The example highlights SMART's capability to generate semantically effective mutants.

3.2 Logic-Based Code Chunking

Code chunking has proven effective in LLM-related tasks [58, 60], as it reduces input size and helps LLMs focus more accurately on relevant code context. However, naively splitting a focal method by a fixed chunk size may break the semantic structure of the code and lead to incoherent inputs. To address this issue, we design a *logic-based code chunking* strategy that partitions methods along their natural control-flow and declaration boundaries. The procedure is illustrated in Algorithm 1.

The algorithm takes a focal method M as input and outputs a set of chunks C. We first extract the line numbers of all statements in M and parse the method into its abstract syntax tree (AST). Next, we collect the target nodes that represent control-flow structures (`if`, `for`, `while`, `do-while`, and `try-catch`). These nodes are sorted in descending order of their line numbers to ensure bottom-up processing of nested statements.

For each target node t, we identify the corresponding code lines l_t. If these lines have not yet been collected, we create a chunk containing the code of t and extend it with the declaration statements that immediately precede t to preserve semantic completeness. We then mark these lines as processed by removing l_t, and the declaration lines l_d from the set $\mathcal{L}$.

After handling all control-flow nodes, the remaining lines in $\mathcal{L}$ are grouped into consecutive segments. Each segment forms a separate chunk, ensuring that every line of the focal method is included in exactly one chunk. Finally, the algorithm returns the set of all chunks C, preserving the logical structure of the original method while reducing the complexity of mutation generation.

3.3 Context-Aware Few-Shot Examples Retrieval via RAG

To provide the LLM with relevant guidance when generating mutants, we design a RAG component that retrieves context-aware few-shot examples from a repository of real-world bugs.

We collect 130,000 single-hunk Java bugs from open-source repositories. Each bug is represented as a pair of *pre-fix* and *post-fix* code, i.e., the buggy code before the patch and the corresponding fixed code after the patch. These bug–fix code pairs serve as the source of potential few-shot examples. To enable efficient semantic retrieval, we embed each buggy method using the encoder of CodeT5 [59], a pre-trained model specialized for code representation learning. The embeddings are stored in a vectorized dataset, enabling scalable similarity search.

Given a code snippet (e.g., an entire method or a code chunk) to be mutated, we compute its embedding using the same encoder. We then perform a similarity search against the repository using *Euclidean Distance* to measure the closeness between embeddings. The Top-N most similar bug–fix pairs are selected as context-aware few-shot examples, where N is a hyperparameter.

3.4 Prompt Design

To guide LLMs in generating high-quality mutants, we design a structured prompt template, as illustrated in Figure 3. The prompt consists of four main components: *Instruction*, *Focal Method*, *Code Chunk*, and *Few-Shot Examples*, together with explicit *Output Instructions*.

We first define the task by natural language instruction. To provide the global context, we include the entire focal method. This ensures that the LLM can understand the broader semantics of the

Algorithm 1: The Chunking Algorithm of SMART

Input: The focal method: M
Output: The set of code chunks: C

$C \leftarrow \phi$
$\mathcal{L} \leftarrow$ get_line_num_set(M)
$\mathcal{A} \leftarrow$ parse_ast(M)
$\mathcal{T} \leftarrow$ get_target_nodes($\mathcal{A}$, {"IfStmt", "ForStmt", "WhileStmt", "DoWhileStmt", "TryStmt"})
$\mathcal{T} \leftarrow$ sort_by_line_descending($\mathcal{T}$)
for $t \in \mathcal{T}$ **do**
    $l_t \leftarrow$ get_line_num_set(t)
    if $\mathcal{L} \cap l_t \neq \phi$ **then**
        $c \leftarrow$ get_code_by_lines($\mathcal{L} \cap l_t$)
        $d \leftarrow$ get_preceding_decl_stmts(t)
        $l_d \leftarrow$ get_line_num_set(d)
        $c \leftarrow c \cup$ get_code_by_lines($\mathcal{L} \cap l_d$)
        $C \leftarrow C \cup \{c\}$
        $\mathcal{L} \leftarrow (\mathcal{L} - l_t - l_d)$
for $seg \in$ split_into_consecutive_segments($\mathcal{L}$) **do**
    $c \leftarrow$ get_code_by_lines(seg)
    $C \leftarrow C \cup \{c\}$
return C

[Instruction]: Below is the original Java method, followed by a specific code chunk extracted from it. Your task is to generate **{N}** mutant versions by applying single-line mutations only within the code chunk. Note: In software engineering, a mutant refers to a variant of the original program created by introducing small syntactic changes, which are typically used for mutation testing.
[Entire Focal Method]: **{JAVA METHOD}**
[The Current Chunk]: Only mutate these lines: **{Code Chunk}**
[Few-Shot Examples]: <json> **{Examples}** </json>
[Output Instructions]:
1. A mutant can only occur on one line.
2. Your output must be like: <json> [{ "precode": "", "aftercode": "" }] </json>. The "precode" represents the line of code before mutation, and it can't be empty, "aftercode" represents the line of code after mutation. Note that you may need to generate multiple pairs of "precode" and "aftercode".
3. Prohibit generating mutants that are identical to the original code (precode) or duplicate any previously generated mutants.
4. Output all mutants in JSON format, ensuring they are wrapped in <json></json> tags.

Fig. 3. The Prompt Template of SMART for Mutation Generation

code and avoid generating inconsistent or irrelevant changes. Then, we highlight the specific code chunk to be mutated. By restricting modifications to this chunk, we reduce ambiguity in the mutation task and force the LLM to focus on a semantically coherent portion of the method. To further enhance context-awareness, we integrate bug–fix pairs retrieved by RAG as few-shot examples. These examples are serialized in JSON format to maintain consistency and to make it easier for the LLM to learn the structure of valid input–output mappings. Finally, we explicitly constrain the output format. The LLM must generate mutants such that each mutation changes

only one line, and the output must follow a JSON-based structure containing the `precode` and `aftercode` fields. This design facilitates automatic parsing and integration of generated mutants into downstream mutation-testing pipelines.

3.5 Fine-Tuning

In addition to the RAG component, which provides relevant, context-aware examples at inference time, we employ supervised fine-tuning (SFT) to specialize the model parameters for mutation generation. The primary goal of SFT in SMART is to align the model more closely with producing semantically effective mutations by training it on mutations that have been empirically validated to couple with real-world bugs.

Each mutation pair, consisting of the original code ("precode") and the corresponding mutated code ("aftercode"), serves as a golden example of a meaningful semantic change (details of the training data are presented in Section 4.3). We format these pairs into high-quality training instances that follow the same structure as our final prompt design (see Section 3.4).

For the SFT process, SMART uses a standard causal language modeling objective, which is mathematically formulated as minimizing the cross-entropy loss. The goal is to adjust the model's parameters, denoted by θ, to maximize the conditional probability of generating a target sequence Y (the ground-truth mutant) given an input prompt X. For a single training instance (X, Y), the loss function $\mathcal{L}(\theta)$ is defined as:

$$\mathcal{L}(\theta) = -\sum_{t=1}^{n} \log P(y_t | X, y_{1:t-1}; \theta), \tag{7}$$

where X is the input prompt, which includes the instruction, the entire focal method, and the specific code chunk to mutate; $Y = (y_1, y_2, \ldots, y_n)$ is the target output sequence, which is the ground-truth JSON string specifying the mutation (e.g., `{"precode": "...", "aftercode": "..."}`); y_t denotes the t-th token in the target sequence Y; and $P(y_t|X, y_{1:t-1}; \theta)$ is the probability assigned by the model with parameters θ to the correct token y_t at position t, given the input prompt X and all preceding ground-truth tokens $y_{1:t-1}$. By minimizing this loss over the entire training dataset, the model parameters θ are updated via backpropagation to make the ground-truth outputs more probable, thereby aligning the model with the task of generating realistic mutants.

4 Evaluation Design

In this section, we elaborate on the details of our evaluation design.

4.1 Research Questions

We aim to evaluate SMART by answering the following research questions.

- **RQ1 (Validity): Does SMART generate more valid mutants than existing approaches?**
- **RQ2 (Effectiveness): How well do SMART-generated mutants resemble real bugs compared with those generated by existing approaches?**
- **RQ3 (Mutation-Based Test Case Prioritization): How effective is SMART for mutation-based test case prioritization compared with existing approaches?**
- **RQ4 (Mutation-Based Fault Localization): How effective is SMART for mutation-based fault localization compared with existing approaches?**
- **RQ5 (Ablation Study on Components): How useful is each component for generating effective mutants?**

Table 1. The Statistics of the Used Datasets

Name	Bug #	Avg. Test #	Avg. Buggy Mtd. #	Avg. Mtd. LOC
Defects4J	701	250.5	2.02	34.15
ConDefects	1290	26.1	1	26.82

4.2 Studied Mutation Generation Approaches

In this study, we compare SMART with two representative state-of-the-art LLM-based mutation generation approaches: LLMorpheus [48] and LLMut [50]. LLMorpheus represents one of the most recent frameworks for leveraging LLMs in mutation testing, while LLMut conducts a systematic study of mutation effectiveness and validity. Together, these two approaches serve as strong baselines for evaluating the performance of SMART.

Beyond these baselines, we further conduct an ablation study to isolate the contributions of individual components in our design. Specifically, we explore the following variants:

- **SMART**$_R$, which augments the vanilla LLM with RAG to retrieve in-context few-shot examples;
- **SMART**$_C$, which applies logic-based code chunking without retrieval or fine-tuning;
- **SMART**$_{RC}$, which combines retrieval with chunk-level mutation generation;
- **SMART**$_{CF}$, which applies supervised fine-tuning while using chunking; and
- **SMART**, our complete approach that integrates RAG, chunking, and fine-tuning.

For mutation generation, we evaluate a diverse set of LLMs. Specifically, we employ five open-source models: **DeepSeek-Coder-6.7B** [69], **Llama-3.1-8B** [18], **Qwen-2.5-7B**, **Qwen-2.5-14B**, and **Qwen-2.5-32B** [65]. The three Qwen-2.5 variants share the same architecture but differ in parameter scales, which allows us to study the effect of model size under a controlled setting. Additionally, for experiments that do not involve fine-tuning, we also include **GPT-4o** [1] (i.e.,SMART$_R$, SMART$_C$, SMART$_{RC}$, LLMut, and LLMorpheus). Note that for fairness and consistency, we adopt GPT-4o as the backend model for LLMorpheus [48] and LLMut [50], since GPT-4o is the most widely studied model in existing LLM-based mutation testing research [48, 50].

Through these comparisons, we aim to examine not only whether SMART outperforms existing methods, but also how each of its components contributes to improving the quality of generated mutants and their downstream utility in mutation testing.

4.3 Training Data and Subjects Under Test

We conduct our study on two widely used benchmarks of real-world Java bugs: **Defects4J 2.0** [27] and **ConDefects** [62]. Defects4J 2.0 is a widely used benchmark in mutation testing research [6, 12, 28, 30, 31, 37, 46, 50, 68]. It provides a collection of historical bugs from open-source projects across diverse domains, offering a broad and representative set of real-world software faults. In contrast, ConDefects is constructed from tasks in the AtCoder programming contest[1], and thus complements Defects4J with competitive programming–style defects. Both datasets contain real faults together with developer-written bug-revealing test cases, which provide a reliable basis for evaluating mutation-based techniques. Table 1 presents the statistics of the two datasets. We observe that bugs from Defects4J exhibit greater complexity compared to those in ConDefects. Specifically, they are subjected to far more developer-written tests (250.5 on average vs. 26.1 in ConDefects) and are located in methods with longer code bodies (34.15 LOC vs. 26.82 LOC), making them more challenging for mutation generation and analysis.

[1] https://atcoder.jp

For the experiments on *mutation effectiveness*, *mutation validity*, and *mutation-based TCP*, we generate mutants from the *fixed* versions of each bug. This allows us to measure (1) the similarity between generated mutants and their corresponding real faults, (2) the validity of the generated mutants, and (3) whether bug-revealing test cases are prioritized earlier by mutation-based TCP techniques. For the experiments on *MBFL*, we instead generate mutants from the *buggy* versions of the programs. This setting enables us to evaluate the impact of mutants on MBFL techniques such as MUSE [35] and Metallaxis [39].

To avoid potential data leakage, we randomly select two projects from Defects4J 2.0, namely *Lang* and *Cli*, and exclude them from our evaluation. For training, we adopt the mutant dataset constructed by LLMut [50]. In particular, to adapt LLMs toward generating semantically meaningful mutants, we use mutants that are *coupled with real-world bugs*, i.e., mutants that can be killed by at least one bug-revealing test. Our training set consists of 13,760 such coupled mutants, which are collected from diverse sources, including LLMs (e.g., DeepSeek[69], GPT-4o [1], and CodeLlama [43]) and mutation testing tools (e.g., LEAM [46], Major [29], and μBERT [12]).

4.4 Mutation Generation Setups

For each real bug in the studied datasets, we generate mutants from the corresponding methods. For RQ1–RQ3, we use fixed versions, while for RQ4, we use buggy versions. If a bug spans multiple methods, we invoke the generation procedure independently for each method. Given a focal method (or its logic-based chunks, in the case of SMART), we configure the LLM to generate N mutants, where N equals the number of lines of code in the input, following common practice [50]. Following the practice of [14, 50], we use the *Top-6* few-shot examples. During mutation generation, all LLMs are run with their default temperature and token limitations.

4.5 Fine-tuning Settings

We implement SMART using Hugging Face Transformers[2] and perform SFT with full-parameter updates for all LLMs. The loss is computed only on the model outputs. To avoid out-of-memory errors, we truncate input sequences to a maximum length of 4096 tokens. This cutoff balances context retention and computational efficiency, while covering all training instances. Following standard practice, we fine-tune each selected LLM for 3 epochs with cosine learning rate decay and a 10% warmup ratio. The base learning rate is set to 3.0×10^{-5}. The per-device batch size is 4, with gradient accumulation steps set to 4.

5 Results and Analysis

In this section, we present the experimental results and answer the research questions.

5.1 RQ1: Comparison of Mutant Validity

5.1.1 Metrics. As introduced in Section 2.2, mutant validity reflects whether the generated mutants are well-formed and practically usable. We evaluate the validity of mutants generated by different LLM-based mutation approaches from three complementary perspectives:

- **Generation Rate (Ge. R.).** In both SMART and LLMut, the number of requested mutants is explicitly specified in the prompt (denoted as *Exp.*). We measure the proportion of successfully parsed mutants (denoted as *Gen.*) relative to the expected number, i.e., Gen./Exp.. A higher generation rate indicates that the model follows the generation instructions more reliably.
- **Non-duplicate Rate (ND. R.).** We compute the proportion of generated mutants that are non-redundant, i.e., not identical to the original program or to other generated mutants, given by

[2] https://huggingface.co/docs/transformers/en/index

Table 2. The Validity Comparison

Approaches	Defects4J (701 Bugs)					ConDefects (1290 Bugs)				
	Exp.	Gen.	Ge. R.	ND. R.	Com. R.	Exp.	Ge.	Ge. R.	ND. R.	Com. R.
LLMorpheus (GPT-4o)	-	4092	-	88.25%	58.71%	-	3861	-	83.35%	99.32%
LLMut (GPT-4o)	46873	6519	13.91%	95.77%	83.05%	33238	1900	5.72%	97.84%	99.95%
LLMut (DS-6.7B)	46873	3142	6.70%	55.25%	68.26%	33306	4661	13.99%	72.19%	99.58%
LLMut (LM3.1-8B)	46873	3507	7.48%	67.49%	66.50%	33306	5530	16.60%	84.25%	99.25%
LLMut (QW2.5-7B)	46873	7623	16.26%	83.13%	70.79%	33306	7413	22.26%	88.20%	99.51%
LLMut (QW2.5-14B)	46873	10501	22.40%	86.93%	75.82%	33306	19877	59.68%	89.69%	99.73%
LLMut (QW2.5-32B)	46873	23708	50.58%	84.96%	78.63%	33306	25265	75.86%	96.37%	99.70%
SMART$_{RC}$ (GPT-4o)	35979	17579	48.86%	97.98%	85.65%	26980	11680	43.29%	99.32%	99.65%
SMART (DS-6.7B)	35979	22012	61.18%	92.44%	82.84%	26980	21841	80.95%	97.81%	99.78%
SMART (LM3.1-8B)	35979	20030	55.67%	91.04%	76.09%	26980	18686	69.26%	97.03%	99.74%
SMART (QW2.5-7B)	35979	19971	55.51%	92.22%	79.43%	26980	20375	75.52%	97.49%	99.76%
SMART (QW2.5-14B)	35979	22574	62.74%	93.71%	81.51%	26980	21492	79.66%	98.37%	99.79%
SMART (QW2.5-32B)	35979	23083	64.16%	93.80%	83.32%	26980	20828	77.20%	98.41%	99.72%

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

$|A - D|/|A|$. This metric evaluates whether the model produces diverse outputs rather than trivial or repeated ones.

- **Compilable Rate (Com. R.).** We calculate the proportion of generated mutants that compile successfully, i.e., $|C|/|A|$. This metric reflects the syntactic correctness of the generated code and its practical utility in downstream mutation testing tasks.

5.1.2 Results. Table 2 summarizes the validity results of different approaches. For each dataset, we report the expected number of mutants (*Exp.*), the number of successfully parsed mutants (*Gen.*), and three validity metrics: generation rate (*Ge. Rate*), non-duplicate rate (*ND. Rate*), and compilable rate (*Com. Rate*). Since GPT-4o is a proprietary model and does not support SFT, we report only the results of SMART$_{RC}$ for GPT-4o, which incorporates RAG and code chunking without fine-tuning.

Across both datasets, SMART consistently delivers higher validity than LLMut across all five open-source LLMs. On Defects4J, for example, SMART improves LLMut's generation rate by more than 40 PPs on Qwen-2.5-14B (62.74% vs. 22.40%), while also achieving higher non-duplicate and compilable rates. Even in the restricted GPT-4o setting, SMART$_{RC}$ substantially outperforms LLMut and LLMorpheus on Defects4J, attaining the highest non-duplicate and compilable rates. On ConDefects, SMART achieves generation rates above 60% across all open-source models, whereas LLMut remains below 30%, and it also achieves non-duplicate and compilable rates comparable to those of LLMut (GPT-4o) while outperforming LLMorpheus (GPT-4o).

We also observe systematic differences between the two datasets. ConDefects generally yields much higher generation and compilable rates than Defects4J, largely because it consists of algorithmic tasks implemented as small single-file programs, which are structurally simpler and easier for LLMs to mutate correctly.

Answer to RQ1: SMART substantially improves mutant validity. It raises the weighted-average **generation rate** from 42.89% (LLMut) to 65.6% (+22.71 PPs; ↑ 52.95%). The **non-duplicate rate** reaches 95.62% (+8.24–9.75 PPs over the baselines), and the **compilable rate** reaches 90.21% (up to +11.78 PPs). Moreover, after applying SMART, all open-source LLMs achieve performance competitive with GPT-4o.

5.2 RQ2: Comparison of Mutant Effectiveness

5.2.1 Metrics. To evaluate the effectiveness of LLM-generated mutants, we adopt several complementary metrics to assess how closely the generated mutants resemble real bugs. Following the definitions in Section 2, these metrics capture detectability, coupling, and semantic similarity:

Table 3. The Effectiveness Comparison

Approaches	Defects4J (701 Bugs)					ConDefects (1290 Bugs)				
	MS	R. B. D.	Coup.	Avg. O.	O.≥ 0.8	MS	R. B. D.	Coup.	Avg. O.	O.≥ 0.8
LLMorpheus (GPT-4o)	74.20%	51.36%	48.61%	16.14%	90	63.47%	11.47%	56.09%	3.87%	36
LLMut (GPT-4o)	67.78%	29.91%	37.89%	10.37%	90	79.71%	7.36%	63.29%	3.79%	66
LLMut (DS-6.7B)	70.71%	25.25%	45.82%	8.55%	60	79.34%	30.00%	65.07%	15.67%	203
LLMut (LM3.1-8B)	58.23%	22.54%	32.09%	5.53%	33	79.86%	19.15%	66.85%	9.51%	123
LLMut (QW2.5-7B)	59.95%	44.51%	36.53%	14.58%	103	81.80%	22.64%	66.32%	11.24%	188
LLMut (QW2.5-14B)	65.15%	58.92%	39.78%	22.69%	205	80.67%	65.58%	66.31%	35.15%	576
LLMut (QW2.5-32B)	66.51%	77.75%	37.32%	25.26%	278	81.24%	80.62%	66.04%	43.01%	730
SMART$_{RC}$ (GPT-4o)	69.41%	89.44%	37.24%	27.61%	288	76.93%	78.76%	62.94%	41.18%	530
SMART (DS-6.7B)	74.72%	92.72%	40.21%	28.70%	298	84.42%	95.74%	69.51%	51.05%	739
SMART (LM3.1-8B)	65.46%	91.16%	37.54%	27.15%	250	84.62%	93.41%	70.66%	49.41%	657
SMART (QW2.5-7B)	76.12%	93.58%	43.04%	29.30%	275	85.56%	94.19%	72.16%	50.12%	697
SMART (QW2.5-14B)	75.76%	93.87%	42.14%	29.23%	289	87.32%	94.26%	72.23%	50.22%	709
SMART (QW2.5-32B)	75.99%	92.87%	42.77%	29.16%	300	87.44%	94.50%	72.37%	49.63%	712

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

- **Real-Bug Detection (R.B.D.).** The percentage of bug-revealing tests, i.e., tests that fail on the buggy version, that also kill at least one generated mutant. This metric measures whether the generated mutants can be detected by the same tests that expose real bugs.
- **Coupling Rate (Coup.).** The proportion of generated mutants that are coupled with real bugs, i.e., mutants that can be killed by at least one bug-revealing test. This metric quantifies how closely the generated mutants align with real bugs.
- **Average Ochiai Coefficient (Avg. O.).** To further assess semantic similarity, we compute the Ochiai coefficient between each mutant and its corresponding real bug based on their sets of failing test cases. The calculation of *Avg. O.* is given in Formula 4. A higher score indicates stronger semantic similarity between generated mutants and real bugs.
- **High-Similarity mutants (O. ≥ 0.8).** Following Ojdanic et al. [37], we also report the number of real bugs whose average Ochiai coefficient with the generated mutants is at least 0.8. This stricter metric highlights how many real bugs are closely approximated by generated mutants.

We also report the **Mutation Score (MS)** for completeness, but note that it is insufficient for assessing mutation generation effectiveness, as a high score may simply indicate easy-to-kill mutants rather than realistic ones. The closeness metric measures the semantic distance between a generated mutant and its corresponding historical real bug; thus, all baselines use the same ground-truth bug as the reference.

5.2.2 Results. Table 3 presents the results of the effectiveness evaluation on both Defects4J and ConDefects. We can make the following key observations.

First, SMART consistently outperforms the baselines on nearly all metrics. On Defects4J, SMART (Qwen-2.5-7B) achieves the highest real-bug detection rate (93.58%), surpassing the strongest LLMut variant by more than 15 PPs. Similarly, SMART (Qwen-2.5-32B) yields the largest number of high-Ochiai bugs (300), indicating a stronger ability to generate mutants closely coupled with real bugs. In terms of semantic similarity, SMART across different models achieves higher average Ochiai coefficients (around 27–29%) than LLMut (mostly below 23%).

Second, SMART shows an even larger advantage on ConDefects. For example, SMART (DS-6.7B) achieves an average Ochiai coefficient of 51.05%, compared with only 15.67% for LLMut (DS-6.7B). Similarly, SMART with Qwen-2.5-14B and Qwen-2.5-32B achieves the highest real-bug detection rates (over 94%) and coupling rates (over 72%), along with more than 700 high-Ochiai bugs, substantially outperforming all baselines.

Third, while LLMorpheus (GPT-4o) shows relatively high mutation scores (74.20% on Defects4J), it performs poorly in semantic similarity (16.14% Avg. O.) and high-Ochiai bug count (90). In

contrast, SMART$_{RC}$ (GPT-4o), even without fine-tuning, improves the average Ochiai to 27.61% and detects 288 high-Ochiai bugs.

Finally, by comparing SMART$_{RC}$ (GPT-4o) with the full SMART using open-source LLMs, we find that **fine-tuning enables 7B-scale models to achieve performance comparable to, and sometimes even better than, GPT-4o**. This result highlights the practical value of our approach by showing that strong effectiveness can be achieved even with relatively small open-source models.

Overall, these results show that SMART consistently generates more effective mutants than both LLMut and LLMorpheus. By improving real-bug detection, fault coupling, and semantic similarity, SMART provides a stronger basis for mutation-based testing applications.

Answer to RQ2: SMART consistently improves the effectiveness of LLM-generated mutants over LLMut and LLMorpheus. For **real bug detection**, SMART achieves a weighted average of 92.61%, compared with 57.86% for LLMut (↑ 60.1%; +34.75 PPs) and 31.99% for LLMorpheus (↑ 189.5%; +60.62 PPs). For the **coupling rate**, SMART reaches 54.94%, slightly above LLMut (53.12%) and LLMorpheus (52.23%). For the **average Ochiai coefficient**, SMART achieves 38.44%, versus 25.61% for LLMut (↑ 50.1%; +12.83 PPs) and 10.18% for LLMorpheus (↑ 277.6%; +28.26 PPs). These results show that SMART generates mutants more closely aligned with real-world faults, while enabling fine-tuned 7B models to achieve GPT-4o-level performance.

5.3 RQ3: Comparison of Mutation-Based Test Case Prioritization

5.3.1 Metrics. In mutation-based test case prioritization (TCP), mutant quality is indirectly reflected by the resulting prioritization effectiveness. Following prior work [6, 16, 45, 46], we adopt three widely studied mutation-based TCP techniques, **GRK** (Greedy Kill), **GRD** (Greedy Distinguish), and **HYB-ω**, which have been introduced in Section 2.3.1. For each TCP technique, we measure effectiveness using the **Average Percentage of Faults Detected (APFD)** [41], a standard metric in TCP research, computed as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_n}{nr} + \frac{1}{2n}, \tag{8}$$

where r denotes the number of bugs, n denotes the number of test cases to be prioritized, and TF_i denotes the ranking of the first test case prioritized by a TCP approach that detects the i^{th} bug. A higher APFD indicates earlier bug detection.

5.3.2 Results. Table 4 reports the APFD results of three mutation-based TCP techniques (GRK, GRD, HYB-ω) using mutants generated by different approaches.

On Defects4J, SMART consistently outperforms LLMut and LLMorpheus across all three techniques. For example, with Qwen-2.5-32B, SMART achieves the highest APFDs: 0.867 (GRK), 0.8805 (GRD), and 0.8805 (HYB-ω), exceeding the corresponding LLMut scores (0.8181/0.8312/0.8312). Compared with LLMut under the same model, SMART consistently outperforms on all metrics, showing the benefit of chunking, RAG, and fine-tuning. On ConDefects, SMART remains competitive but does not surpass the best LLMut variants. The highest scores are achieved by LLMut (Qwen-2.5-32B) for GRK (0.8823) and by LLMut (Qwen-2.5-14B) for GRD/HYB-ω (0.8684). SMART reaches 0.869 on GRK and 0.865 on GRD/HYB-ω with DeepSeek-6.7B, within 1.33 PPs of the best LLMut results, while still outperforming LLMorpheus and LLMut with GPT-4o.

Table 4. Mutation-Based Test Case Prioritization

Approaches	Defects4J (701 Bugs)			ConDefects (1290 Bugs)		
	GRK	GRD	HYB-ω	GRK	GRD	HYB-ω
LLMorpheus (GPT-4o)	0.6707	0.6746	0.6746	0.8445	0.844	0.844
LLMut (GPT-4o)	0.6246	0.6295	0.6295	0.8521	0.8649	0.8649
LLMut (DS-6.7B)	0.6	0.5995	0.5995	0.8509	0.8523	0.8523
LLMut (LM3.1-8B)	0.565	0.5657	0.5657	0.8471	0.8398	0.8398
LLMut (QW2.5-7B)	0.6736	0.677	0.677	0.8551	0.8637	0.8637
LLMut (QW2.5-14B)	0.7436	0.7493	0.7493	0.8768	0.8684	0.8684
LLMut (QW2.5-32B)	0.8181	0.8312	0.8312	0.8823	0.876	0.876
SMART$_{RC}$ (GPT-4o)	0.8399	0.8552	0.8552	0.8675	0.8623	0.8623
SMART (DS-6.7B)	0.8522	0.8793	0.8793	0.869	0.865	0.865
SMART (LM3.1-8B)	0.8285	0.849	0.849	0.8585	0.8523	0.8523
SMART (QW2.5-7B)	0.8442	0.8635	0.8635	0.8658	0.8636	0.8636
SMART (QW2.5-14B)	0.8563	0.8735	0.8735	0.8595	0.8577	0.8577
SMART (QW2.5-32B)	0.867	0.8805	0.8805	0.8658	0.8609	0.8609

Answer to RQ3: On Defects4J, SMART achieves the best APFD across GRK/GRD/HYB-ω, clearly outperforming LLMut and LLMorpheus; on ConDefects, it remains highly competitive (within 0.3–1.3 PPs of the best LLMut variants) while SMART$_{RC}$ (GPT-4o) also surpasses prior GPT-4o baselines.

5.4 RQ4: Comparison of Mutation-Based Fault Localization

5.4.1 Metrics. In mutation-based fault localization (MBFL), we evaluate the quality of generated mutants by examining their impact on bug localization accuracy. Given a buggy method, each statement is ranked by its suspiciousness score computed from mutation analysis (Section 2.3.2). We then assess fault localization effectiveness using the following standard metrics:

- **Top-k.** The proportion of bugs for which at least one buggy statement is ranked within the top k positions. Higher Top-k values indicate that buggy statements are placed earlier, reducing debugging effort. In our study, we report Top-1, Top-3, and Top-5, following common practice in MBFL research [46].
- **Mean Average Rank (MAR).** For each bug, we compute the average rank of all buggy statements and then average this value over all bugs. A lower MAR indicates that buggy statements tend to appear closer to the top of the ranking overall.
- **Mean First Rank (MFR).** For each bug, we record the rank of the first buggy statement and then average this value over all bugs. A lower MFR indicates less effort is needed to inspect the ranked list before reaching a buggy statement.

If multiple statements share the same suspiciousness score, we follow prior work [46, 70] and use $E_{inspect}$ to compute their expected rank. In MBFL, when the mutants produced by one approach lead to larger Top-k scores and smaller MFR or MAR values compared to another, the former is regarded as achieving better fault localization performance.

5.4.2 Results. As shown in Table 5 and Table 6, SMART consistently improves the utility of generated mutants for MBFL for both MUSE and Metallaxis. First, SMART achieves higher Top-k results than LLMut and LLMorpheus on both datasets. For instance, in terms of Top-1, SMART (Qwen-2.5-7B) localizes 134 bugs with MUSE and 119 bugs with Metallaxis on Defects4J, and 132 with MUSE and 115 with Metallaxis on ConDefects, outperforming LLMut and LLMorpheus. Second, SMART also reduces localization effort: on both datasets, SMART (DS-6.7B) achieves the lowest MAR and MFR values. These results suggest that SMART-generated mutants not only rank buggy statements higher, but also improve overall localization effectiveness across datasets.

Table 5. Statement-Level Mutation-Based Fault Localization on Defects4J (701 Bugs)

Approaches	MUSE					Metallaxis				
	Top-1	Top-3	Top-5	MAR	MFR	Top-1	Top-3	Top-5	MAR	MFR
LLMorpheus (GPT-4o)	100	143	148	157.61	157.52	93	144	148	157.58	157.52
LLMut (GPT-4o)	56	119	149	143.73	143.11	49	126	150	143.88	143.16
LLMut (DS-6.7B)	57	118	151	151.56	151.30	55	118	148	151.60	151.33
LLMut (LM3.1-8B)	69	111	127	162.25	162.16	68	111	127	162.21	162.15
LLMut (QW2.5-7B)	85	198	247	119.88	119.32	78	201	240	119.96	119.34
LLMut (QW2.5-14B)	104	252	308	94.56	93.62	97	259	309	94.58	93.59
LLMut (QW2.5-32B)	107	280	365	59.55	57.36	93	292	376	57.12	55.10
SMART$_{RC}$ (GPT-4o)	105	293	360	59.72	57.39	92	299	374	60.13	57.34
SMART (DS-6.7B)	116	306	397	52.64	50.91	97	316	399	52.93	50.94
SMART (LM3.1-8B)	78	251	327	76.40	75.29	76	260	332	76.51	75.16
SMART (QW2.5-7B)	134	314	392	59.68	58.27	119	314	390	59.87	58.27
SMART (QW2.5-14B)	130	303	378	56.79	55.09	111	310	383	57.12	55.10
SMART (QW2.5-32B)	129	309	377	56.52	54.78	116	305	386	56.77	54.77

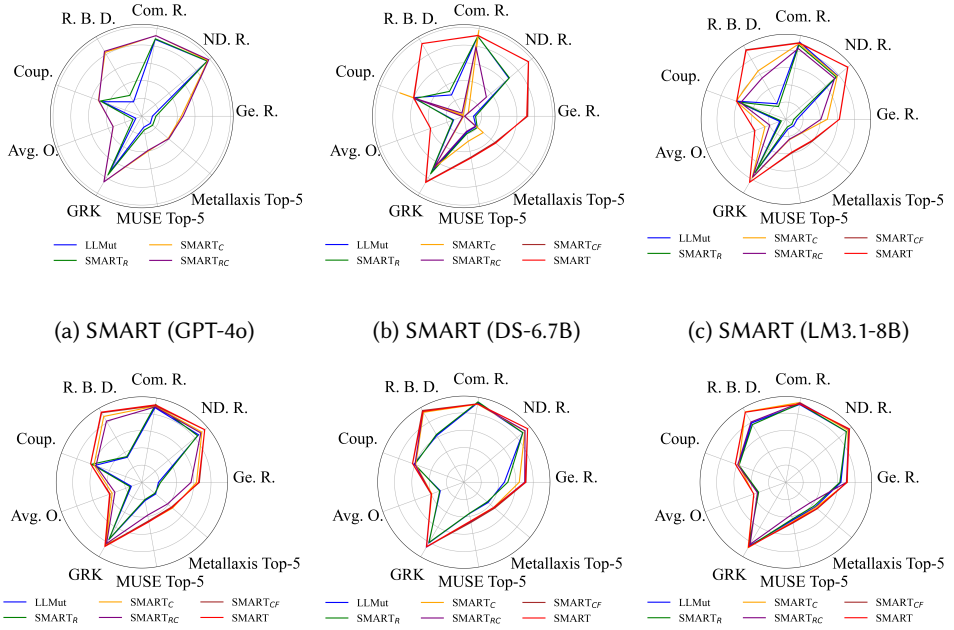
Table 6. Statement-Level Mutation-Based Fault Localization on ConDefects (1290 Bugs)

Approaches	MUSE					Metallaxis				
	Top-1	Top-3	Top-5	MAR	MFR	Top-1	Top-3	Top-5	MAR	MFR
LLMorpheus (GPT-4o)	11	16	16	197.53	197.53	12	16	16	197.53	197.53
LLMut (GPT-4o)	10	35	53	186.75	186.74	8	34	51	186.79	186.78
LLMut (DS-6.7B)	24	105	198	148.39	148.39	28	117	193	148.42	148.42
LLMut (LM3.1-8B)	14	45	72	176.29	176.28	15	47	69	176.35	176.34
LLMut (QW2.5-7B)	20	56	95	163.12	163.12	17	60	95	163.27	163.27
LLMut (QW2.5-14B)	103	263	383	88.09	88.08	79	246	373	88.41	88.39
LLMut (QW2.5-32B)	106	308	475	58.26	58.25	85	283	450	58.79	58.77
SMART$_{RC}$ (GPT-4o)	60	225	350	114.35	114.35	61	221	335	114.55	114.54
SMART (DS-6.7B)	105	349	499	70.16	69.96	102	340	479	70.54	70.35
SMART (LM3.1-8B)	120	287	398	115.08	115.02	101	297	392	115.17	115.09
SMART (QW2.5-7B)	132	337	474	101.36	101.28	115	336	454	101.48	101.39
SMART (QW2.5-14B)	143	349	476	90.03	89.89	120	340	464	90.31	90.16
SMART (QW2.5-32B)	132	367	516	85.33	85.18	109	354	499	85.58	85.39

Answer to RQ4: SMART improves MBFL effectiveness by achieving higher Top-k results than LLMut and LLMorpheus on both datasets, while also reducing localization effort. On Defects4J, SMART (QW2.5-7B) ranks 134 bugs in Top-1 under MUSE, compared with 107 for LLMut and 100 for LLMorpheus. Moreover, SMART (DS-6.7B) ranks 399 bugs in Top-5 under Metallaxis, compared with 376 for LLMut and 148 for LLMorpheus. On ConDefects, SMART remains consistently competitive and achieves the best MAR/MFR with QW2.5-32B. Overall, these results show that SMART-generated mutants better support fault localization by ranking buggy statements earlier.

5.5 RQ5: Component Ablation Study

Figure 4 summarizes representative metrics from the first four RQs, covering validity, effectiveness, MBTCP, and MBFL. We observe that enabling all three optimizations in SMART (RAG, chunking, and fine-tuning) achieves the best overall performance across different LLMs. Among them, chunking contributes the largest gains. RAG also provides substantial improvements when used alone (i.e., SMART$_R$ vs. LLMut), but its incremental benefit becomes smaller once chunking is enabled, suggesting that chunking already offers strong contextual guidance. Fine-tuning further adapts mid-scale open-source models (7B–14B) to mutation generation, enabling them to match or even surpass GPT-4o under the full SMART configuration.



(a) SMART (GPT-4o) (b) SMART (DS-6.7B) (c) SMART (LM3.1-8B)

(d) SMART (QW2.5-7B) (e) SMART (QW2.5-14B) (f) SMART (QW2.5-32B)

Fig. 4. Comparison among Different Variants of SMART

Table 7. Token Cost Statistics

	LLMut	SMART$_R$	SMART$_C$	SMART
Avg. Query Time per Mtd.	1	1	4.6	4.6
Avg Tokens per Query	716	778	950	979

Answer to RQ5: All three components of SMART contribute to improved mutant generation and downstream effectiveness, with code chunking consistently delivering the largest gains.

6 Discussion

6.1 Token Cost Analysis for Mutation Generation

While SMART improves the quality and effectiveness of generated mutants, it also introduces additional token and time costs due to its design choices. Both RAG and code chunking increase prompt size compared with LLMut. RAG adds retrieved bug–fix examples into the prompt, while chunking transforms a single query at the method level into multiple queries at the chunk level. As a result, chunking amplifies the total token consumption and query time.

Table 7 reports the average query time and token usage per method. Compared with LLMut, RAG alone (SMART$_R$) only increases average tokens per query from 716 to 778 due to the inclusion of few-shot examples. By contrast, chunking (SMART$_C$) substantially increases the number of LLM invocations, leading to an average of 4.6 queries per method and a token cost of 950 tokens per query. The full SMART has the same average of queries as chunking alone, while it costs 979 tokens

Table 8. RAG Cost in SMART

	RAG Building	RAG Retrieval (Mtd)	Avg. Mtd	RAG Retrieval (Chk)	Avg. Chk
Time (s)	613	5,468	2.7	12,961	6.5

Table 9. GPU Costs of Supervised Fine-Tuning in SMART

Model	Time (s)	FLOPs	GPU Mem.
DS-6.7B	17,552	1.78E+14	40GB
LM3.1-8B	16,267	1.48E+14	42GB
QW2.5-7B	14,066	1.89E+14	40GB
QW2.5-14B	30,025	4.76E+14	75GB
QW2.5-32B	74,493	1.05E+15	140GB

per query. In summary, SMART incurs higher token usage primarily due to chunking. Overall, the average token cost per query increases by about **36.7%** (from 716 in LLMut to 979 in SMART).

6.2 Cost Analysis of SMART

For the GPU-related steps of SMART, which dominate the overall cost, we report **wall-clock time** to capture end-to-end cost. For the SFT stage, the main hardware bottleneck of LLM-based approaches, we further report **peak GPU memory usage** and **floating-point operations (FLOPs)**. Peak memory reflects practical hardware requirements, while FLOPs provide a hardware-independent measure of training cost. To reduce randomness, we repeat each experiment three times and report the average.

Table 8 reports the wall-clock time for RAG building and retrieval. RAG building takes 613 seconds, which is a one-time offline cost for constructing the embedding index. For retrieval, method-level retrieval (i.e., *Mtd*, used for non-chunking prompts) takes 5,468 seconds in total, averaging 2.7 seconds per bug. Chunk-level retrieval (i.e., *Chk*, used for code-chunking prompts) requires 12,961 seconds in total, averaging 6.5 seconds per bug. The higher cost of chunk-level retrieval is mainly due to the increased number of retrieval queries triggered by fine-grained code chunks. Note that a bug may require multiple retrievals under chunking. Overall, the time overhead introduced by RAG remains moderate, especially considering that retrieval is performed once per bug and does not dominate the overall mutation generation process.

Table 9 reports the computational cost of supervised fine-tuning (SFT) for different base models. The SFT time ranges from 14,066 seconds (QW2.5-7B) to 74,493 seconds (QW2.5-32B), with corresponding FLOPs ranging from 1.48×10^{14} to 1.05×10^{15}. As expected, both wall-clock time and FLOPs increase substantially with model size. Peak GPU memory usage varies from 40GB to 42GB for 6-8B models to 75GB for the 14B model and 140GB for the 32B model, indicating that GPU memory becomes the primary hardware constraint for larger models. Overall, SFT is the most computationally intensive component of SMART and requires non-trivial GPU resources, especially for models beyond 14B parameters. However, this cost is incurred only once per model and can be amortized across multiple mutation campaigns.

6.3 The Influence of RAG Choices

To examine whether the choice of similarity metric affects SMART, we randomly sampled 100 real-world bugs (50 from Defects4J and 50 from ConDefects) and compared three commonly used similarity measures in RAG: cosine similarity, dot-product similarity, and Euclidean distance.

As shown in Table 10, Euclidean distance consistently achieves the best performance across mutation generation quality and downstream applications (MBTCP and MBFL). Specifically, compared with cosine and dot-product similarity, Euclidean distance improves the coupling rate and real-bug

Table 10. Performance of Different RAG Distance Metrics

Method	Gen. R.	ND. R.	Com. R.	R. B. D.	O.≥0.8	AOC	APFD-grk	MUSE Top-5	Meta. Top-5
Cos	65.7%	60.2%	48.5%	81	34	0.18	0.88	51	49
Dot-product	62.0%	57.0%	46.7%	85	34	0.19	0.86	53	52
Euclidean	63.2%	57.2%	47.5%	85	39	0.20	0.9	56	54

detection rate by approximately 5%, increases the number of high-Ochiai bugs by 14%, raises the average Ochiai score by about 10%, and boosts APFD by 3%-5%. It also improves MUSE-Top5 and Meta-Top5 by 9% and 10%, respectively. Overall, the results suggest that SMART is not overly sensitive to the choice of similarity metric, as all three metrics yield comparable performance trends. Nevertheless, Euclidean distance provides consistently better results and is therefore adopted as the default configuration.

6.4 Threats to Validity

Our study faces the following potential threats to validity.

First, the choice of programming language, datasets, mutation generation approaches, and the GPU resource requirements may limit the generalizability of our findings. We mitigate this by using Java (the most widely studied language), two datasets (Defects4J and ConDefects), and the state-of-the-art LLM-based baselines, along with diverse widely studied LLMs (e.g., GPT-4o, DeepSeek-Coder, Llama-3.1, and Qwen-2.5).

Second, the inherent randomness of LLMs (e.g., decoding strategies) and configuration choices could influence the results. To reduce this threat, we follow the default parameters of each model and evaluate multiple open-source LLMs of varying scales as well as GPT-4o. Another related factor is the number of retrieved few-shot examples used in RAG. Previous work [50] reports that varying the number of examples between 5 and 10 does not lead to statistically significant performance differences. Following this finding, we adopt a moderate and commonly used setting within this range. While different retrieval sizes may introduce minor variations, prior evidence suggests that such variations are unlikely to alter the overall conclusions. Nevertheless, we acknowledge that retrieval configuration remains a tunable factor.

Finally, although we carefully controlled the training and evaluation pipeline, potential data leakage remains a general concern in LLM-based studies. To mitigate this threat, we adopted multiple preventive measures. First, for the RAG retrieval database, we collected data exclusively from real-world GitHub repositories that do not appear in either Defects4J or ConDefects. We explicitly excluded all projects sharing the same names as those in the two benchmarks to ensure strict dataset separation. Second, for supervised fine-tuning (SFT), we partitioned projects into disjoint training and testing sets to prevent cross-project leakage. Specifically, the projects *Lang* and *Cli* were used only to provide training data and were never involved in evaluation. Third, beyond checking exact matches, we further quantified the similarity between the fine-tuning dataset and the evaluation dataset using N-gram analysis. The results indicate that the two sets are highly dissimilar: the 2-gram similarity is only 1.65%, and the similarity further drops to 0.019% for 10-grams, suggesting that long token sequences are rarely shared across datasets. These results provide stronger evidence that memorization-based leakage is unlikely to significantly affect our findings. Nevertheless, it is still possible that parts of Defects4J or ConDefects may have appeared in the pre-training data of large language models, which is beyond our control. To estimate its potential impact, we examined whether SMART produced exact matches (i.e., syntactically identical mutants) with real bugs. On Defects4J, the proportions of exact matches remain low across all models, ranging from 1.91% (GPT-4o) to 2.59% (DS-6.7B). On ConDefects, the rates are even lower,

all below 0.3% (e.g., 0.05% for GPT-4o). These small percentages suggest that, although data leakage cannot be completely ruled out, its influence on our results is limited.

7 Related Work

7.1 Mutation Testing

Mutation testing is essential for software testing and analysis research, which systematically introduces artificial faults into a program, used to evaluate test effectiveness [25, 38]. Moreover, mutation testing has been widely applied in software engineering, serving as a substitute for real bugs [2, 10, 28, 36], and supporting tasks such as fault localization [35, 39, 53], test case prioritization [44], and automated program repair [63, 64].

All downstream applications rely on valid and effective mutants. Prior studies suggest that effectiveness increases when mutants are syntactically or semantically close to real bugs [20, 22, 26]. For example, MBFL benefits from mutants that are coupled with real bugs [35, 39, 53].

7.2 Mutation Generation

Traditional mutation testing relies on *rule-based* operators, i.e., predefined program transformations such as replacing arithmetic or relational operators. Most tools, including PIT [8], Major [29], MuJava [34], AccMut [54], WinMut [52], and Milu [24], follow this paradigm, while MIN [67] further specializes it to method invocations.

With the rise of machine learning, *learning-based* methods emerged. Examples include DeepMutation [49] using seq2seq translation, LEAM [46] with learning-guided rule expansion, and μBERT [12], which predicts masked code tokens via BERT [15]. However, such approaches often struggle to ensure syntactic correctness and mutation diversity.

Recently, *LLM-based* methods have attracted growing attention [9, 11, 17, 23, 48, 50]. These studies show the feasibility of leveraging powerful generative models for mutation generation, though most focus on limited metrics such as mutation score or repair effectiveness. In contrast, our work systematically evaluates diverse LLMs and prompt strategies against rule-based and learning-based baselines, measuring both the validity and effectiveness of the generated mutants in multiple downstream tasks.

8 Conclusion

In this paper, we propose SMART, a novel LLM-based mutation generation framework that combines retrieval-augmented generation, code chunking, and supervised fine-tuning. Through extensive experiments on 1,991 real-world Java bugs from Defects4J and ConDefects, we demonstrate that SMART consistently generates more valid and effective mutants than state-of-the-art baselines. It significantly improves mutation validity (e.g., generation rate from 42.89% to 65.6%) and effectiveness (e.g., real bug detection rate of 92.61% vs. 57.86% for LLMut), while also boosting the performance of downstream applications such as test case prioritization and fault localization. Moreover, our results show that SMART enables 7B-scale open-source models to achieve performance competitive with, and in some cases surpassing, GPT-4o, highlighting its strong practical value.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant No. 2024YFE0202900; the National Natural Science Foundation of China under Grant No. 62202040; the ITEA projects GreenCode (Project No. 23016) and GENIUS (Project No. 23026); and the Luxembourg National Research Fund (FNR) through the CORE project MiCE (C23/IS/18182513).

Data Availability

The artifact of SMART is accessible at https://zenodo.org/records/19509595 [51].

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] 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

[3] 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

[4] 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

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374 [cs.LG] doi:10.48550/arXiv.2107.03374

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 654–674. doi:10.1145/3643756

[12] 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

[13] 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

[14] 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

[15] 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

[16] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752. doi:10.1109/TSE.2006.92

[17] 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

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] doi:10.48550/arXiv.2407.21783

[19] 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

[20] 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

[21] 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

[22] 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

[23] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2025. Challenging Bug Prediction and Repair Models with Synthetic Bugs. (2025), 133–144. doi:10.1109/SCAM67354.2025.00021

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2021. Retrieval-augmented generation for knowledge-intensive NLP tasks. arXiv:2005.11401 [cs.CL] doi:10.48550/arXiv.2005.11401

[33] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 46–57. doi:10.1109/ISSRE.2015.7381798

[34] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. (2006), 827–830. doi:10.1145/1134285.1134425

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188. doi:10.1109/ICSM.1999.792604

[42] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948. doi:10.1109/32.962562

[43] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2024. Code llama: Open foundation models for code. arXiv:2308.12950 [cs.CL] doi:10.48550/arXiv.2308.12950

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

[50] 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 Transactions on Software Engineering and Methodology (TOSEM)* (2026). doi:10.1145/3805038

[51] Bo Wang, Ming Deng, Mingda Chen, Chengran Yang, Youfang Lin, Mark Harman, Mike Papadakis, and M. Jie Zhang. 2025. The Artifact of TyMut. doi:10.5281/zenodo.19509595

[52] 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

[53] 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

[54] 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

[55] Guoqing Wang, Zeyu Sun, Sixiang Ye, Zhihao Gong, Yizhou Chen, Yifan Zhao, Qingyuan Liang, and Dan Hao. 2025. Do advanced language models eliminate the need for prompt engineering in software engineering? *ACM Transactions on Software Engineering and Methodology* (2025). doi:10.1145/3771933

[56] Guoqing Wang, Chengran Yang, Xiaoxuan Zhou, Zeyu Sun, Bo Wang, David Lo, and Dan Hao. 2026. TestDecision: Sequential Test Suite Generation via Greedy Optimization and Reinforcement Learning. arXiv:2604.01799 [cs.SE] doi:10.48550/arXiv.2604.01799

[57] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024). doi:10.1109/TSE.2024.3368208

[58] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088. doi:10.18653/v1/2023.emnlp-main.68

[59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

[60] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. 1258–1268. doi:10.1145/3691620.3695501

[61] Ming Wen, Zifan Xie, Kaixuan Luo, Xiao Chen, Yibiao Yang, and Hai Jin. 2022. Effective isolation of fault-correlated variables via statistical and mutation analysis. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2053–2068. doi:10.1109/TSE.2022.3209590

[62] 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

[63] 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

[64] 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

[65] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. arXiv:2505.09388 [cs.CL] doi:10.48550/arXiv.2505.09388

[66] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120. doi:10.1002/stvr.430

[67] 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* (2025). doi:10.1109/TSE.2025.3573751

[68] Yifan Zhao, Yizhou Chen, Zeyu Sun, Qingyuan Liang, Guoqing Wang, and Dan Hao. 2024. Spotting code mutation for predictive mutation testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1133–1145. doi:10.1145/3691620.3695491

[69] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. arXiv:2406.11931 [cs.SE] doi:10.48550/arXiv.2406.11931

[70] 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