Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches

Megan Maton
University of Sheffield
UK

Gregory M. Kapfhammer
Allegheny College
USA

Phil McMinn
University of Sheffield
UK

Abstract

Mutation testing tools normally rely on rule-based operators that mechanically swap and change source code tokens without understanding the code's purpose. Recently, Large Language Models (LLMs) have enabled mutant generators to consider more of the code's context and history. However, current LLM-based mutation testing methods have limited prompts that prevent them from unleashing the full power and creativity of the LLM to produce a diverse set of aggressive, yet realistic and productive, mutants. This paper's novel approach uses an LLM to interpret a method and re-implement it entirely with mutations guided by hazard analysis, analogous to a programmer misunderstanding a project's requirements or an aspect of an algorithm's implementation. To enable the empirical comparison of the new hazard-guided techniques with both prior LLM-based and traditional mutation testing tools, this paper also presents and applies a framework that integrates representative LLM-based methods. Using this framework and 279 bugs in 15 projects from the Defects4J dataset, the results show that the hazard-guided techniques can harness both local and cloud-based LLMs to generate compilable, diverse, and powerful mutants that help test cases to detect unique defects not found by other methods.

CCS Concepts

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

Keywords

Software Testing, Mutation Testing, LLM, Hazard Analysis

ACM Reference Format:

Megan Maton, Gregory M. Kapfhammer, and Phil McMinn. 2026. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Mutation testing is an effective technique for evaluating the quality of a test suite [10, 18]. It works by inserting synthetic faults, called mutants, into a program and checking whether the tests detect them. A test suite that distinguishes between a mutant and the original program is considered stronger than one that does not [24, 48]. Studies have shown that mutation adequacy correlates with real fault detection [4, 21], making it a valuable tool for companies like Google [36, 37] and open-source projects hosted on GitHub [38, 39].

Traditional mutation testing uses rule-based operators that make small syntactic changes to a program [23]. While these mutants correlate with real faults, Ahmed et al. found that commonly used mutation operators are "not representative of real-world faults" and "cannot be reliably used to reproduce these faults" [3]. The results from this study suggest that mutants lack, for instance, the introduction of new method calls and code blocks, which would make them more closely resemble defects in real programs. These findings highlight a need for mutation testing tools that make context-specific mutants reflecting real-world developer mistakes. Yet, the question remains: how can tools systematically make such mutants without creating too many that are otherwise unproductive?

The first methods developed to answer this question (such as, for example, DeepMutation) trained deep learning models on a dataset of real faults, and used it to generate mutants that mimic them [35, 41, 44]. Since they required a developer to train a model on large numbers of real defects, these methods were not widely adopted. Recent work has explored the use of pre-trained Large Language Models (LLMs) to generate mutants [45]. An LLM-based technique can produce context-specific mutants by leveraging their understanding of code semantics [6]. With that said, a key remaining challenge involves preventing the LLMs from making uncontrolled changes that result in uncompilable or unrealistic mutants.

To address this, researchers have developed techniques, such as "masking", in which specific code regions are hidden and the LLM predicts replacements, thereby generating a mutant [25, 43]. Since tools like LLMorpheus and μBERT focus on making changes to specific regions of code rather than entire methods, they cannot fully achieve the goals that Ahmed et al. identified for realistic mutants. While BugFarm generates mutants that differ from the program under test in multiple locations, its focus is on generating mutants that are "hard to detect" or "hard to repair" [16]—rather than making mutants that reflect realistic developer mistakes that might arise from misunderstanding of a program's specification.

For an automated LLM-based technique to generate realistic method-level defects, there is a need to "harness" the LLM so that it does not make uncontrolled changes that are uncompilable or unrealistic. The heretofore unaddressed challenge is to guide the LLMs to produce context-specific mutants that reflect real-world mistakes while maintaining characteristics like compilability. To address this challenge, this paper introduces the use of hazard analysis to guide LLM-based mutant generation. Developers of safety-critical systems use hazard analysis methods, like HAZOP (HAZard and OPerability Study) and STPA (System-Theoretic Process Analysis),

EASE 2026, Glasgow, Scotland, United Kingdom

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

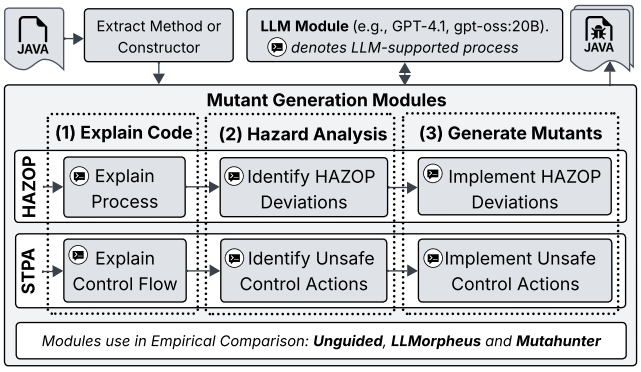


Figure 1: *multiplex*'s HAZOP and STPA implementations.

to identify potential risks by systematically analyzing deviations from intended behavior [27, 30]. Applying HAZOP and STPA to a program's workflow can guide an LLM to generate mutants that represent hazardous deviations from the original program's intent.

Figure 1 shows that this paper's hazard-guided mutation analysis technique operates in three phases. First, the LLM extracts the key algorithmic steps from the program's source code as a series of natural language steps (for HAZOP) or a control flow diagram (for STPA). Second, the method applies hazard analysis guide words or unsafe control actions to identify potential deviations. Third, the LLM generates mutants by turning these deviations back into source code. This approach leverages the LLM's semantic reasoning capabilities while using systematic hazard analysis to constrain its code changes. Since hazard analysis works at the method level—and not in specific code regions like LLMorpheus [43]—the LLM is empowered to make diverse and aggressive changes to the program.

Since the promise of pre-trained LLMs spurred the growth of LLM-based mutation testing tools—each with their own prompting strategies, support for programming languages, and use of different LLM backends—there is a pressing need for an integrated framework that enables the study of new techniques. To address this concern and support the empirical comparison of the presented hazard-guided techniques with prior LLM-based methods, we created a tool called *multiplex* (*mu*tation analysis with *l*arge language models for *t*esting, *i*n a *p*rompt-*l*evel *ex*perimentation framework). As shown in Figure 1, this paper makes **two key contributions** through *multiplex*: (1) a novel method that uses LLMs in the aforementioned three phases and (2) an extensible framework for the integration and study of prior LLM-based mutation testing methods.

Using 15 Java projects from the Defects4J benchmark, and both local (i.e., gpt-oss:20B) and cloud-based (i.e., GPT-4.1) LLMs, this paper's **third contribution** is the quantitative and qualitative comparison of mutant generators. The experiments that answer four research questions (**RQs**) reveal **for RQ1**: an unguided LLM produces trivial-to-detect mutants and, while hazard guidance is helpful, compilability issues persist across all LLM-based methods; **for RQ2**: (i) the volume of mutants may bias subsumption analysis and (ii) random sampling showed that hazard-guided mutants can effectively complement stronger rule-based mutants; **for RQ3**: (i) mutant volume may also bias the proportion of mutants triggering the same tests as real defects, and (ii) controlling for set size, HAZOP and STPA uniquely "test substitute" a significant proportion of defects; **for RQ4**: when using a local LLM instead of a cloud-based one, the methods perform similarly in terms of compilability, subsumption, and defect relationships. Furthermore, the hazard-guided methods qualitatively exceed the prior LLM-based ones by generating syntactically large, yet semantically subtle, mutants that tests don't detect, thereby offering useful assistance to developers.

2 Background

Hazard Analysis is a family of methods used to evaluate processes for potential risks, particularly in safety-critical industries [11], allowing engineers to scrutinize aspects of a task or procedure to identify the causes and consequences of things going wrong.

HAZOP (HAZard and OPerability study) identifies hazards as a result of specific deviations from a system's intended design [27]. It breaks a process down into individual elements, systematically applying "guide words" to modify their intent, including: (1) *No* (negation of the original intent); (2) *More* (quantitative increase); (3) *Less* (quantitative decrease); (4) *As Well As* (qualitative increase); (5) *Part Of* (qualitative decrease); (6) *Reverse* (logical opposite); (7) *Other Than* (complete substitution). An example of applying HAZOP to operating a train includes applying the guide word *More* to the parameter "Speed", showing the potential hazard of the driver accelerating beyond a safe limit; or *As Well As*, for the hazard of simultaneously applying both the brake and the accelerator.

STPA (System-Theoretic Process Analysis) is a hazard analysis approach that frames safety as a control problem [30]. STPA analyzes the effects of unsafe control actions issued by a "controller" (human or software) to a system—e.g., a driver triggering the opening of a train's doors while it is moving at full speed. Applying STPA involves examining each command the controller provides, modeling the effects of applying seven different "Unsafe Control Actions" (UCAs) to each, namely: (1) an unsafe command is *Provided*; (2) a control command required for safety is *Not Provided*; (3) it is given *Too Early*; or (4) it is given *Too Late*; (5) a control command is given *Out Of Order*; (6) *Stops Too Soon*; or (7) it is *Applied For Too Long*. The train driver's action falls into the first category, unsafe command *Provided*, resulting in a hazard if the doors were to actually open.

Mutation Analysis is the process of systematically producing variations of a program (i.e., "mutations"). Each modified version is called a "mutant". Mutants can be used to evaluate the fault-finding capability of a test suite [10], since the mutations themselves are designed to represent potential programmer mistakes (i.e., real bugs). Assuming that all tests pass on the original program, a mutant is said to be *killed* if any test fails when executed on it. The percentage of mutants killed represents the test suite's "mutation score" [4]. Any mutant that does not result in a test failing is said to *survive*, and may represent a testing "blind spot". However, a mutant may be semantically equivalent to the original program, and therefore impossible to kill [5]. The presence of *equivalent* mutants is one way in which a mutation score may not accurately reflect a test suite's fault detection capability. Another case arises when some mutants are *subsumed* by others. A mutant m_1 is *subsumed* by mutant m_2 if killing m_2 always kills m_1. The mutant m_1 is *redundant* and therefore can inflate the mutation score [28].

Until recently, mutation analysis has been limited to producing mutants by introducing small syntactical changes to a program intended to produce behavioral deviations. For this purpose, researchers proposed a variety of *rule-based* mutation operators to

Table 1: Applying HAZOP Guide Words

Examples of applying HAZOP to the LLM-extracted code explanation of `isAlphanumeric` (Figure 2). In the example deviations, added words appear in **bold**, while removed words appear in ~~strikeout~~.

HAZOP Guide Word	Code Explanation Snippet	Example Deviation
No (not, none)	Return true if all characters are alphanumeric	Return **no** true **value even** if all characters are alphanumeric
More (more of, higher)	Return false if the input is empty	Return false if the input **length** is ~~empty~~ **greater than a minimal threshold**
Less (less of, lower)	Iterate over each character index in the input	Iterate over ~~each~~ **fewer** character ~~index~~ **indices** in the input
Other Than (other)	Return false if the input is null	Returns ~~false~~ **true** if the input is null
Part Of	Starts a loop to examine each character	Starts a loop ~~to~~ **that** examine**s only some** ~~each~~ character**s**
Reverse	Return true if all characters are alphanumeric	Return true **only** if ~~all~~ characters are **not** alphanumeric
As Well As (more than)	Return true if all characters are alphanumeric	Returns true if all characters are alphanumeric **as well as at least one digit is present**

```
  public static boolean isAlphanumeric(CharSequence cs) {
      if (cs == null) {
          return false;
      }
      int sz = cs.length();
+H    boolean hasDigit = false;
      for (int i = 0; i < sz; i++) {
+S        char c = cs.charAt(i);
+S        if (c == '\_' || c == '-') continue;
          if (Character.isLetterOrDigit(cs.charAt(i)) == false) {
                return false;
          }
+H        if (Character.isDigit(c)) {
+H            hasDigit = true;
+H        }
        }
-H      return true;
+H      return hasDigit;
    }
```

This function from the `StringUtils` class shows the original code diffed against two mutations. Lines marked +/− indicate additions and removals; H/S denote *m*-HAZOP or *m*-STPA generated changes.

Figure 2: The `isAlphanumeric` method from commons-lang3.

describe transformation patterns that can be applied to the Abstract Syntax Tree (AST) of a program. They generate mutant versions by modifying suitable AST nodes specified by the operator. For example, a "Relational Operator Replacement (ROR)" operator replaces a relational operator with compatible alternatives, such as changing the operator in a condition, thereby replacing `a > b` with `a >= b`.

LLMs (Large Language Models) for Mutation Analysis. Both researchers and open-source software developers alike have applied LLMs to mutant generation. These efforts have resulted in two tools in particular: LLMorpheus [43] and Mutahunter [7].

LLMorpheus is a research prototype that uses a "mask-and-predict" strategy to generate mutants with LLMs [43]. It identifies locations in program code to mutate, replaces them with the marker text "`<PLACEHOLDER>`" and presents that code to the LLM along with the original fragment. It then asks the LLM to provide three different alternatives to the original (the "predict" phase). Typically, the LLM suggests changes such as switching operators, variables, and method calls. The tool requires the AST of the original code, using pre-determined AST node types to decide where the mutation should occur, similar to rule-based approaches, targeting (1) `if`, `while`, `do-while` and `switch` condition statements; (2) loop initializers, updaters and full headers and (3) `function` call receiver arguments and argument sequences. The original LLMorpheus used a modified version of StrykerJS, a state-of-the-art JavaScript mutation tool, to execute and evaluate the generated mutants. It has since been adapted to other languages, including Java [45].

Mutahunter is an open-source, language-agnostic, LLM-based tool for mutation testing [7]. Its prompt provides an overview of mutation testing and outlines guidelines for potential changes, such as syntax-based logic modifications, return type alterations, failure simulation, and data handling errors. It directs the LLM to target function blocks and "critical areas" while constraining mutants to a single line only. Instead of masking code like LLMorpheus, it prompts the LLM to modify existing lines directly, providing only the original code and, optionally, the program's AST for context.

Table 2: Applying STPA Unsafe Control Actions (UCAs)

Examples applying STPA guide words to the LLM-extracted code explanation of `isAlphanumeric` (Figure 2). In the table, `charCheck`, `loopInit`, and `loopCheck` are specific nodes of the DOT graph extracted from the source code of the method and named by the LLM. In this table, "Character API" is the name the LLM used for the method-under-test.

UCA Guide Word	Example UCA
Provided	Return true provides output even if some chars were not checked due to loop error.
Not Provided	`charCheck` does not provide correct validation when Character API misclassifies characters.
Too Early	Return false (non-alphanumeric char) triggered before all chars are checked.
Too Late	Return false (non-alphanumeric char) triggered after invalid data processed.
Out Of Order	Compute `sz = cs.length()` executed after `loopInit` may misalign loop bounds.
Stopped Too Soon	Loop terminates before all characters are checked.
Applied For Too Long	Loop continues beyond string length due to increment or `loopCheck` error.

Although both LLMorpheus and Mutahunter provide *context* through a complete method or class, these approaches direct the LLM to mutate either specific AST nodes (LLMorpheus), or single code lines (Mutahunter). They do not leverage the LLM's semantic reasoning capabilities for adding and removing code. To achieve this, we now present an approach that mutates higher-level code summaries using the aforementioned hazard analysis techniques.

3 Approach

We introduce an approach that generates code mutants by applying hazard analysis to high-level descriptions of the code's operation using an LLM. We implemented this into our tool called *multiplex* (*mu*tation analysis with *l*arge language models for *t*esting, *i*n a *p*rompt-*l*evel *ex*perimentation framework). Its modular nature supports interchangeable LLM-based mutant generation approaches, thereby creating a platform for replicating the other LLM mutation tools used in our empirical evaluation, as described in Section 4.1. Our approach, which uses either HAZOP or STPA as the underlying hazard analysis technique, can be broken down into the three common steps shown in Figure 1: The initial step, **(1) Explain Code**, inserts the source code of a specified Java constructor or method into a prompt that asks an LLM to extract and explain the method's underlying algorithm. The next step, **(2) Apply Hazard Analysis**, asks the LLM to produce perturbations to the extracted algorithm by applying hazard analysis, modeling the effects of HAZOP guide words

and STPA unsafe control actions. Finally, in **(3) Generate Mutants**, our approach instructs the LLM to write out each modified algorithm as Java source code, producing a final set of mutants. Using the `isAlphanumeric` method in Figure 2 as a worked example, the following subsections detail these steps for HAZOP and STPA.

HAZOP explores hazards by analyzing the potential effects of the deviations from the operation or parameters of a system's intended function. We refer to our application and implementation of HAZOP in *multiplex* as "*m*-HAZOP". The first step of applying HAZOP in *m*-HAZOP involves exposing the key parts of the algorithm of a method that our approach can perturb to produce mutants.
(1) Explain Code. *m*-HAZOP begins by instructing the LLM to dissect the code of a method in a sequence of numbered natural language steps. It specifically prompts the LLM to "explain what the code does, not how it is written", thereby gaining an algorithmic and semantic insight into the method's operation, rather than an exclusively syntactic description. For instance, rather than describing Line 17 in Figure 2 in isolation, the LLM may characterize it as "Return true if all characters are alphanumeric", reflecting its role within the entire method and not merely the logic of the individual line.
(2) Apply Hazard Analysis. *m*-HAZOP then prompts the LLM to identify deviations for the line-by-line descriptions, using HAZOP guide words. For the example description "Return true if all characters are alphanumeric", the LLM applies the guide word *As Well As* and returns the deviation "Returns true if all characters are alphanumeric as well as at least one digit is present", as shown in Table 1.
(3) Generate Mutants. *m*-HAZOP then asks the LLM to interpret each deviation in terms of the code, to produce mutants. As shown in Figure 2, for "Returns true if all characters are alphanumeric as well as at least one digit is present", the LLM adds the `hasDigit` variable on Line 6 and the check on Lines 13–15, before finally returning the `hasDigit` value on Line 18 instead of `true`, meaning that the sequence must contain a digit to be considered alphanumeric.

STPA explores potential hazards by modeling the control of a system. In the context of our approach, this maps to a control-based description of a method to which it can apply unsafe control actions. We refer to our implementation of STPA in *multiplex* as "*m*-STPA".
(1) Explain Code. The first step of *m*-STPA involves prompting an LLM with the source code of the method and asking it to extract a control flow diagram augmented with English descriptions of code behavior at each node. To standardize the graphs it produces, *m*-STPA prompts the LLM to generate graphs in DOT form [12].
(2) Apply Hazard Analysis. Given the control flow description in DOT form, *m*-STPA prompts the LLM to identify potential Unsafe Control Actions (UCAs) for the method-under-test. For instance, with the `isAlphanumeric` example, *m*-STPA applies the *Does Not Provide* guide word leading to the UCA: "charCheck does not provide correct validation…" (Table 2 gives further UCAs for this example).
(3) Generate Mutants. As with HAZOP, our tool prompts an LLM to translate each UCA change into code, thereby producing mutants. To implement the UCA of incorrect validation in Figure 2, we can see that Lines 8 and 9 are added to skip validating specific characters.

Note that both *m*-HAZOP and *m*-STPA modify multiple lines and AST nodes, and, therefore, extend beyond the bounds of traditional rule-based approaches, such as MAJOR, and more recent LLM-based mutant generation tools like LLMORPHEUS and MUTAHUNTER.

4 Evaluation

To evaluate our hazard analysis approach and compare it to existing approaches, including LLMORPHEUS, MUTAHUNTER, and MAJOR, we designed an empirical study centering on four key questions:

RQ1 (Mutant Properties): How do the mutants generated by our hazard-guided LLM approach compare to those of other techniques with respect to compilability and killability rates, and how similar are they to the program under test's original source code?

RQ2 (Subsumption Analysis): To what extent does our approach produce redundant, subsumed mutants as opposed to non-subsumed mutants that subsume those of other approaches (and vice versa)?

RQ3 (Defect Analysis): To what degree do the mutants generated by the hazard-guided LLM mutation techniques serve as "test substitutes" [13] that mirror defects detected by developer tests?

RQ4 (Multi-LLM Comparison): How do the results from RQ1 through RQ3 vary when the LLM-driven mutant generators use a locally-deployed LLM instead of a cloud-based one?

4.1 Tools

As outlined in Section 3, we implemented our hazard analysis techniques into our "*multiplex*" tool, as the *m*-HAZOP and *m*-STPA modules. To enable a fair comparison in terms of other aspects of tool operation, we instantiated LLMORPHEUS's and MUTAHUNTER's prompting and Java AST parsing strategies as further modules in *multiplex*, which we refer to as *m*-LLMORPHEUS and *m*-MUTAHUNTER, respectively. As a baseline comparison, we also implemented an "uninformed" *multiplex* module, *m*-UNGUIDED, which prompts the LLM to create mutants without any instructions involving hazard analysis processes, prescribed code changes, or scope restrictions. Finally, to compare against rule-based mutation, we use MAJOR for Java [19, 22]. MAJOR is a common baseline in mutation studies that is integrated into the DEFECTS4J framework [20], from which we also derive our subjects, as further explained in Section 4.2.

To conduct our evaluation, we used each *multiplex* module with both a cloud-based LLM (OpenAI's GPT-4.1; training cut-off June 2024, released April 2025), and a locally deployed model (OpenAI's open-weight gpt-oss:20B model; training cut-off June 2024, released August 2025). Since OpenAI reports that gpt-oss:20B "delivers similar results to OpenAI GPT-o3-mini on common benchmarks" [33], our goal for using this local model was not to facilitate a direct comparison between the two LLMs, but to ensure the reproducibility of the results and the viability of *multiplex* when run on developer workstations. The *multiplex* tool, with all modes, prompts, and code is available in this papers's replication package [1].

4.2 Subjects

We used the DEFECTS4J dataset (v3.01) as subjects for mutant generation and evaluation against real-world tests and faults. We did not consider classes with failing tests, or classes where MAJOR failed to execute, thus excluding Closure and JacksonDatabind. Since this study evaluates hazard analysis of processes, we restricted the scope

Table 3: Defects4J Subjects used in the Empirical Study

Project	Bugs Eval.	Bugs Total	Project	Bugs Eval.	Bugs Total
Chart	24	26	JacksonXml	5	110
Cli	27	39	Jsoup	26	93
Codec	17	18	JxPath	18	22
Collections	23	28	Lang	12	61
Compress	33	47	Math	24	106
Csv	15	16	Mockito	5	38
Gson	14	18	Time	15	26
JacksonCore	21	26			

of the evaluation to bugs in constructors and methods only (excluding, for example, bugs in class fields). For the remaining projects, we attempted all bugs, except for *jsoup* and *math*. We attempted only the first 30 bugs for these two projects to prevent them from dominating the dataset, due to their relatively large number of versions (93 and 106 bugs, respectively). These constraints reduced the overall evaluation set to 322 bugs. We then attempted to generate mutants for one fixed method in each bug, retrying a second time in cases where mutant generation failed due to non-deterministic aspects of interfacing with and using LLMs (e.g., timeouts and rate limits). This resulted in each approach completing for 279 bugs and thus being used in our experiments, as shown in Table 3.

4.3 Method

For RQ1 (Mutant Properties), we evaluate the number of mutants generated by each approach implemented in *multiplex*, as well as Major, the rule-based technique. We count mutants that are generated and compilable (i.e., do not cause a build failure), and how many are killed by the original developer test suites for each subject. We further compare mutants' *mutation sizes* for each technique, by measuring the Levenshtein distance between each mutant and its original method, ignoring spacing and comments (since LLMs often add superfluous comments and/or modify code formatting). We chose Levenshtein distance, also called edit distance, over AST metrics, since it is more sensitive to fine-grained modifications—such as method call changes—applied by LLMorpheus and Mutahunter.

For RQ2 (Subsumption Analysis), we wrote a script to calculate the *minimal set of mutants M* for each technique from its set of mutants killed by developer tests. (Note that since subsumption is assessed using killing tests, we cannot analyze subsumption for surviving mutants without writing further tests. We leave this as an item for future work.) For each mutant $m_1 \in M$, there is no other mutant $m_2 \in M$ where m_2 subsumes m_1; i.e., where m_1 is killed by a subset of tests that kill m_2. By computing these sets, we can both identify (a) the strongest sets of mutants for a technique [28], and (b) obtain an idea of the amount of redundancy in mutant generation. In other words, if a minimal set for a technique is very small, it means that it generates a lot of redundant mutants. We then create a superset of the mutants from all approaches, and calculate the uniquely subsuming set of mutants across all approaches. That is, the set of mutants that are not subsumed by any other mutant, and are not killed by an identical set of tests to any other mutant.

For RQ3 (Defect Analysis), we analyzed the developer tests that killed mutants generated by each *multiplex* configuration, as well as Major. We designed a script to identify the mutants that fail on the same subset of test cases that "trigger" the original defect, i.e., "test substitutions" [13]. Since we do not evaluate the reason a test fails, this set includes "strong substitutions" which fail for the exact same reason as the original test cases (cf. [13, 21]). We analyzed these mutant-defect pairs to find which techniques exclusively generated "test substitute" mutants for each real-world program defect.

Table 4: Mutant Statistics

"Mutants" is the number of mutants generated for each *multiplex* mode; "Compilable" reflects mutants that are syntactically valid; "Killed" represents mutants killed by developer test suites (where the associated percentage score reflects "kill rate" of compilable mutants).

	Mutants	Compilable		Killed	
	#	#	(%)	#	(%)
m-hazop	3589	3202	(89.2)	2435	(76.0)
m-stpa	2490	2354	(94.5)	1530	(65.0)
m-unguided	2681	2560	(95.5)	2402	(93.8)
m-llmorpheus	26268	18071	(68.8)	11014	(60.9)
m-mutahunter	1259	1179	(93.6)	856	(72.6)
Major	24325	24325	(100.0)	15215	(62.5)

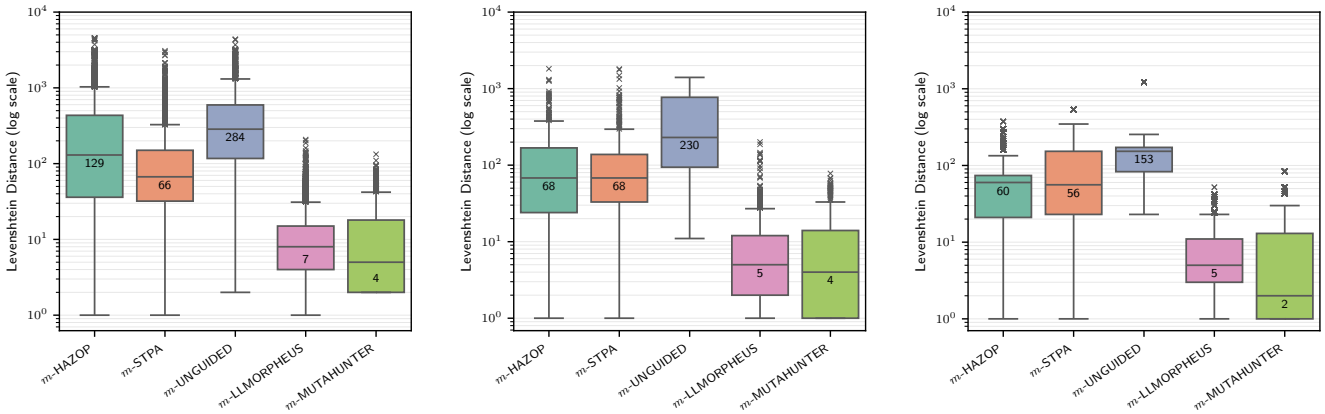
The number of mutants a method generates can vary significantly depending on the technique. Our preliminary results for RQs 2 and 3 suggested these counts can influence the final results. To control for this, we employed what we refer to in this paper as *smallest set downsampling*, where for each method-under-test (MUT), we identified the smallest mutant set size $n_{\min}$ across all approaches, and randomly downsampled larger sets of mutants generated for other techniques to match this size. To address the potential bias of the smallest set remaining unchanged, we also performed *half smallest set downsampling* by sampling all techniques to $\lceil n_{\min}/2 \rceil$ mutants. For this, we excluded any MUT where $n_{\min} = 1$, as maintaining a set size of one would fail to mitigate the aforementioned bias. We repeated the random sampling 1000 times for both the smallest set downsampling and half smallest set downsampling methods.

For RQs 1–3, we employ a cloud-based LLM, GPT-4.1. **For RQ4 (Multi-LLM Comparison)**, we re-performed the method that we used for RQ1–3, but this time ran *multiplex* with a locally-deployed model (i.e., gpt-oss:20B) and then compared the obtained results.

5 Results

RQ1: Mutant Properties. Table 4 shows the number of mutants produced, along with the numbers and percentages of those that are compilable and killed for each LLM-driven technique provided by *multiplex* and the rule-based Major. While Major works at the Java bytecode level and is designed not to produce uncompilable mutants, LLMs can produce uncompilable code. This is particularly noticeable for *m*-llmorpheus, which tries several replacements at each AST node, some of which are syntactically invalid. The compilability rates of other LLM-based mutation techniques, including *m*-hazop and *m*-stpa, are approximately 90% or higher.

Figure 3a shows the distribution of mutation sizes produced by each technique, measured using the Levenshtein edit distance from their original methods on a log scale. We do not show a box for Major as it mutates bytecode as opposed to source code. Furthermore, its mutation sizes are very small by construction, due to its application of rule-based operators. Our hazard-based LLM approaches (*m*-hazop and *m*-stpa) produce some of the largest mutant sizes, along with *m*-unguided. *m*-llmorpheus and *m*-mutahunter are



(a) All mutants edit distance from original method to mutant method

(b) Uniquely subsuming mutants (with smallest set downsampling)

(c) Mutants exclusively matching defects (with smallest set downsampling)

Figure 3: Distributions of mutation sizes, measured using Levenshtein edit distance from their original methods on a log scale.

limited to individual lines of code, and so are restricted from making significant code changes. The large changes do not necessarily result in trivial mutants, however, as evidenced by the kill rates recorded in Table 4. The kill rates for *m*-HAZOP (76%) and *m*-STPA (65%) are not vastly dissimilar from those obtained for MAJOR (62.5%) and *m*-LLMORPHEUS, which performed best on this metric (60.9%). However, these data values combined show the importance of *guiding* the LLM towards useful mutants. This is evidenced by *m*-UNGUIDED, which produces large mutations and whose 93.8% kill rate suggests the production of many trivial mutants.

Conclusion for RQ1: While our hazard-based LLM techniques produce uncompilable mutants, like the other LLM-based approaches, they also achieve kill rates comparable to these approaches and MAJOR. Crucially, hazard guidance mitigates the production of trivial, easy-to-kill mutants observed in the *m*-UNGUIDED variant.

RQ2: Subsumption Analysis. Table 5 shows the results of the subsumption analysis of mutants that were killed by developer tests. The minimal sets produced by our analysis indicate the potential redundancy levels in each technique's mutant set. If the minimal set forms only a small percentage of the overall set, then many mutants are subsumed and potentially wasteful. In this regard, for "no sampling", we see that the high-mutant producers (*m*-LLMORPHEUS and MAJOR) have the highest levels of redundancy (smallest minimal set proportions) while the reverse is true for the low mutant producers (*m*-MUTAHUNTER). This changes as we apply sampling to ensure the same original set sizes: for both sampling methods, each technique falls into a range no wider than 15 percentage points, with *m*-UNGUIDED at the bottom-end and MAJOR at the top.

The unique sets reveal which techniques have the ability to produce the most subsuming mutants overall. Again with "no sampling", the high-mutant producers dominate. Applying sampling to control for this, we see that *m*-UNGUIDED produces a lot of "weak" mutants, subsumed by other techniques in the final, unique set of mutants, which pits every mutant from each technique against each other. Conversely, the rule-based MAJOR tool produces the highest proportion of the strongest, unsubsumed mutants. Of the remaining LLM-based techniques, *m*-MUTAHUNTER and *m*-STPA are the strongest performers, depending on the sampling method applied.

Table 5: Subsumption Analysis

"#" (total) / μ (mean sampled) set sizes for: (i) Minimal Set (non-subsumed mutants per technique) and (ii) Unique Set (technique contribution to global subsuming set, shown as count and %).

	Total	Killed	Minimal Set		Unique Set	
	# / μ	# / μ	# / μ	% kill.	# / μ	% uniq.
No Sampling (raw counts)						
m-HAZOP	3589	2435	533	21.89	42	3.79
m-STPA	2490	1530	504	32.94	18	1.63
m-UNGUIDED	2681	2402	391	16.28	7	0.63
m-LLMORPHEUS	26268	11014	821	7.46	211	19.06
m-MUTAHUNTER	1259	856	430	50.23	11	0.99
MAJOR	24325	15215	1441	9.47	818	73.89
Smallest Set Downsampling (mean averages)						
m-HAZOP	730.00	627.26	313.65	50.01	56.29	12.63
m-STPA	730.00	685.80	357.05	52.06	79.26	17.78
m-UNGUIDED	730.00	699.80	286.58	40.95	8.45	1.90
m-LLMORPHEUS	730.00	472.24	281.79	59.69	65.01	14.58
m-MUTAHUNTER	730.00	669.54	358.37	53.52	84.27	18.91
MAJOR	730.00	542.35	345.88	63.78	152.41	34.20
Half Smallest Set Downsampling (mean averages)						
m-HAZOP	370.00	318.39	221.78	69.66	46.70	12.94
m-STPA	370.00	347.01	250.47	72.18	72.46	20.08
m-UNGUIDED	370.00	355.90	222.61	62.55	11.44	3.17
m-LLMORPHEUS	370.00	240.25	186.54	77.66	53.90	14.93
m-MUTAHUNTER	370.00	338.15	244.23	72.23	72.01	19.96
MAJOR	370.00	275.35	219.50	79.73	104.40	28.93

Figure 3b shows the mutant sizes for smallest set downsampling. The trends are the same as for no sampling and half smallest set downsampling (latter plot not shown for space reasons): the hazard-based *m*-HAZOP and *m*-STPA, along with *m*-UNGUIDED produce the largest mutant sizes. However, the mutants for *m*-HAZOP and *m*-STPA form a significant proportion in the final unique sets, showing that larger mutation sizes also have a role in producing strong, subsuming and potentially valuable mutants.

Conclusion for RQ2: A technique's volume of generated mutants can bias the subsumption analysis results. When we apply sampling we see that the mutants produced by *m*-UNGUIDED are the weakest while MAJOR produces the strongest. However, the LLM techniques also supply strong, complementary mutants, with *m*-HAZOP and *m*-STPA producing significantly larger subsuming mutants sizes, fulfilling a distinct role in mutation testing.

Table 6: Defect Test Substitution

"Total" is the number of mutants generated, "Killed" is the number of those mutants killed by developer tests, "Matched" is the number of defects killed by exactly the same tests as at least one generated mutant (test substitution), and "Unique" is the number of defects matched exclusively by a technique.

	Total	Killed	Matched	Unique
No Sampling (raw counts)				
m-HAZOP	3589	2435	101	1
m-STPA	2490	1530	106	1
m-UNGUIDED	2681	2402	52	1
m-LLMORPHEUS	26268	11014	144	16
m-MUTAHUNTER	1259	856	86	2
MAJOR	24325	15215	146	11
Smallest Set Downsampling (mean averages)				
m-HAZOP	1029.00	719.10	50.85	4.98
m-STPA	1029.00	655.05	62.80	7.09
m-UNGUIDED	1029.00	948.25	30.46	1.45
m-LLMORPHEUS	1029.00	515.88	47.67	7.15
m-MUTAHUNTER	1029.00	696.64	68.68	8.53
MAJOR	1029.00	714.48	60.62	10.42
Half Smallest Set Downsampling (mean averages)				
m-HAZOP	503.00	354.80	31.00	5.60
m-STPA	503.00	318.60	38.80	7.80
m-UNGUIDED	503.00	459.80	22.40	1.80
m-LLMORPHEUS	503.00	236.40	26.80	5.80
m-MUTAHUNTER	503.00	339.80	39.20	10.40
MAJOR	503.00	344.40	35.80	9.80

RQ3: Defect Analysis. Table 6 shows the results of mutants that "test substituted" real developer defects in the DEFECTS4J dataset by triggering the tests that match those that trigger real defects. Each technique was capable of test substituting at least one defect that the others did not, however the high-volume mutant producers (*m*-LLMORPHEUS and MAJOR) matched the most. The results for sampling show that this is indeed because they had the most opportunities to do so—with smallest set downsampling, more defects are uniquely matched by the other techniques, a trend consistent with half smallest set downsampling. The only exception is *m*-UNGUIDED, which continues to perform relatively poorly at matching defects.

Figure 3c shows the distribution of mutation sizes for uniquely matched defects with smallest set downsampling. The trend is the same as for previous box plots. In particular, the mutation sizes for *m*-HAZOP and *m*-STPA are among the largest; yet, both methods competitively match defects, showing that LLM-guided large-scale mutation plays a complementary role in finding real defects.

Conclusion for RQ3: Unless sampling is used, the results of defect analysis can be biased by techniques that simply produce the largest number of mutants. Controlling for size, *m*-UNGUIDED performs poorly, but *m*-HAZOP and *m*-STPA uniquely "test substitute" a significant proportion of real defects. Given that both hazard-guided techniques yield large mutation sizes, this finding provides further evidence to suggest that the LLM-guided production of larger mutations is useful for real-world mutation analysis.

RQ4: Multi-LLM Comparison. For RQ4, we encountered significant difficulties with LLMORPHEUS's prompt with gpt-oss:20B, which exceeded its context window on multiple occasions. This meant we could only obtain a complete set of results for 146 bugs of the DEFECTS4J dataset, and so these are the bugs we used to compare all of the techniques in this paper for this RQ. Table 7 shows the mutant statistics and results of mutants test substituting or "matching" defects. We found that with gpt-oss:20B, *m*-HAZOP tended to produce more mutants and *m*-STPA fewer. As STPA is based on control flow, the number of mutants is dependent on the richness of the control flow diagram. We found that gpt-oss:20B is less expressive in producing DOT descriptions of control flow, which in turn resulted in it creating fewer mutants for *m*-STPA. Yet, we note that the overall trends for mutants compiled and killed remained similar across models, when comparing with Table 4. Mutation sizes also follow similar trends. *m*-HAZOP and *m*-STPA produce the largest mutation sizes (median Levenshtein distances of 101 and 62 with gpt-oss:20B; 129 and 66 with GPT-4.1) compared to *m*-LLMORPHEUS and *m*-MUTAHUNTER (11 and 2 with gpt-oss:20B, 7 and 4 with GPT-4.1). These similarities persist in other aspects also. For example, defect test substitution (without sampling applied) follows similar trends to the first segment of Table 6.

Table 7: Multi-LLM Comparison using gpt-oss:20B

The GPT-4.1 Mutant Statistics and Defect Test Substitution results replicated using gpt-oss:20B.

	Mutants	Compilable		Killed		Defects	
	#	#	(%)	#	(%)	Match.	Uniq.
m-HAZOP	4030	3400	(84.4)	2559	(75.3)	67	2
m-STPA	612	558	(91.2)	292	(52.3)	39	0
m-UNGUIDED	1457	1340	(92.0)	1116	(83.3)	57	0
m-LLMORPHEUS	13617	6382	(46.9)	3447	(54.0)	77	9
m-MUTAHUNTER	682	627	(91.9)	446	(71.1)	45	1
MAJOR	9643	9643	(100)	5210	(54.0)	77	4

Conclusion for RQ4: Applying a different, local model (i.e., gpt-oss:20B) yields differing results in the volume of mutants generated (*m*-HAZOP and *m*-STPA), or their capacity to do so (*m*-LLMORPHEUS). However, the trends in terms of compilability, kill rate, subsumption and defects all follow similar patterns regardless of which model is used (i.e., gpt-oss:20B or GPT-4.1).

6 Discussion and Further Observations

Since the mutants generated by *m*-HAZOP and *m*-STPA have large mutation sizes, and therefore have the potential to "look" quite different from small, traditional rule-based mutations, including those generated by LLMORPHEUS and MUTAHUNTER—which are also small in size (Figure 3a)—we devote this section to discussing hazard-guided mutants as part of a manual analysis that we performed. We did not study mutants generated by *m*-UNGUIDED, since although they are also relatively large in mutational size, the experiments showed them to be relatively trivial to kill and subsume, and thus of arguable less use to developers [37]. Since equivalence is undecidable in general [29], thus mandating a manual analysis, we begin by studying these mutants for equivalence. We then study those that, as part of RQ3, uniquely test substituted DEFECTS4J bugs. Finally, we examine a selection of mutants in both the killed and unkilled, yet non-equivalent, categories. In particular, the latter group of non-equivalent mutants are arguably the most helpful to developers as they are helpful guides during test improvement [15].

Equivalent and Unproductive Mutants. We randomly selected *surviving* (i.e., not killed) mutants generated by *m*-HAZOP and *m*-STPA from six DEFECTS4J projects, chosen for variance over domains—namely *Chart*, *Cli*, *Lang*, *Math*, *Mockito*, and *Time*—to result in 30 mutants for each technique in total, and 60 overall. Two authors then individually assessed these mutants, meeting to resolve differences and reach a consensus. They found that 5 of the 30 mutants

generated for *m*-HAZOP were equivalent to their original counterparts, while the same was true of 15 of the 30 mutants for *m*-STPA. Table 4 shows that 2,435 of 3,589 *m*-HAZOP mutants and 1,530 of 2,490 *m*-STPA mutants were killed by developer tests, yielding survival rates of 21.4% and 33.1%, respectively. Given that in our sample, 16.7% of surviving mutants are equivalent for *m*-HAZOP and 50% for *m*-STPA, we can therefore estimate that the equivalent mutant rate (EMR) of each technique is 3.57% and 16.5%, respectively.

In comparison, Wang et al. [45] manually analyzed similar sample sizes of mutants, estimating MAJOR to have an EMR of 2.1%, while LLMORPHEUS's EMR was 3.1–10.6% depending on the model used. Our analysis therefore confirms a common finding: LLM-based mutation techniques have a higher EMR than their rule-based counterparts. While the EMR for *m*-HAZOP falls toward the end of the range of estimates for LLMORPHEUS, *m*-STPA's EMR is considerably higher at 16.5%. Yet, even here, the number of non-equivalent mutants still significantly outweighs the number of equivalent mutants overall—and *m*-STPA was one of the most effective LLM approaches in the subsumption (RQ2) and defect test substituting (RQ3) analyses when we applied downsampling to the mutant sets.

Finally, we further deemed one non-equivalent mutant from this set to be "unproductive", generated by *m*-HAZOP for *Chart*. This mutant added an additional code line that appended a log entry to `System.err`. Such a mutant would be very hard for a developer to kill with a test, and would not be a useful addition to a test suite. We found no other unproductive mutants, highlighting the strength of *m*-HAZOP and *m*-STPA. Future work should study adjusting prompts to avoid these situations and the creation of these mutant types.

Observation 1: LLM-based mutation techniques have significantly higher equivalent mutant rates (EMRs) than rule-based approaches, and hazard-guided approaches introduced by this paper are no different in this regard. Unproductive mutants are rare; we only found one example, which, with future work, could likely be avoided altogether through improved LLM prompting.

Mutants Test Substituting DEFECTS4J Bugs. We now examine mutants generated by *m*-HAZOP and *m*-STPA that exclusively triggered the matching tests to DEFECTS4J bugs in RQ3 (without sampling) and investigate the reasons why this occurs.

Cli-40. In the `createValue` method of the `TypeHandler` class, this bug returns `null` rather than throwing an exception, which is exclusively replicated by a *m*-STPA mutant. *m*-STPA identifies "Throw ParseException" in its code explanation and applies *Does Not Provide*, writing the mutant so that the exception is not thrown, adding `return null` so the method still compiles. In theory, this action could be replicated by a rule-based operator, but it is not one implemented by MAJOR or PIT, hence why MAJOR did not reveal it. It is also an "aggressive" change with a Levenshtein edit distance of 109, the size of which falls out of the normal mutational range for both LLMORPHEUS and MUTAHUNTER, as shown in Figure 3a.

Gson-9. With this bug, the method `value` in the `JsonWriter` class should return `null` when the variable passed to the method is `null`. Instead, the buggy method always returns the current `this` instance. *m*-HAZOP exclusively finds this bug by mutating step two of its extracted code description "Checks if the given value is null". It applies the *Reverse* guide word, so that the step is "Checks if the given value is not null". It then writes out the Java code by changing the `null` return value to `this`, replicating the behavior of the bug. This behavior could be replicated by a rule-based operator, but its implementation is challenging, since the return type of the new variable must match the old, else the tool will make many uncompilable mutants. As such, this operator is not implemented by MAJOR or PIT. It is not detected by either of the prior LLM approaches. LLMORPHEUS does not target `return` nodes, while MUTAHUNTER appears to only change return *types* rather than mutating return values to `null`, showing the novelty of hazard-guided methods.

Observation 2: Hazard-based analysis can guide LLMs to generate nuanced and challenging mutants that mimic developer mistakes in ways that other rule-based and LLM-based techniques cannot.

Differences Between Hazard-Guided LLM Generated Mutants and Those of Other Techniques. We took a deeper look at the 39 non-equivalent/productive mutants selected for equivalence analysis, plus an additional 5 *non-surviving* mutants for each technique and project that were killed by developer tests (i.e., 99 mutants in total). We classified bugs into one of four categories: *1) Producible by a rule-based operator*: the mutant could have been produced by a tool for mutating Java code, such as MAJOR or PIT; *2) Producible by a combination of rule-based operators:* the mutant could have been produced by a combination of rules considered valid for the first category (i.e., these mutants could be categorized as "higher-order" mutants (HOMs) [17]); *3) Plausibly rule-based:* the mutant could have been produced by a rule, although seldom implemented into tools, and not implemented into MAJOR or PIT; *4) Unique to hazard-guided LLM approaches:* The mutant follows no particular pattern adhering to any of the prior categories or follows a context-specific pattern corresponding to a hazard-guided LLM mutation of a code description. Each mutant was assessed by two authors, who met to reach a consensus on the categorization. From the 99 mutants, we determined that 21 belonged to category 1—rule-based, while a further 6 were "higher-order" combinations of rule-based operators and fitted into the second category, overall accounting for approximately 27% combined of the overall set. We now discuss those in the third category, indicating numbers of mutants in brackets:

We define the first subcategory as *aggressive versions of rule-based operators*, noting that these mutants generated were exaggerated versions of those produced by rule-based operators. For example, adding larger constants rather than just traditional off-by-one amounts (2 mutants), appending a new string literal (1 mutant), or completely rewriting the conditions leading to an exception (1 mutant). The latter mutant changed a condition involving a type check of two arrays to one comparing their length instead.

The second subcategory are mutants that *could* be produced by operators that are *not typically* implemented in mutation systems. These include changing a method call to another method with a compatible type signature (1 mutant), changing a method parameter to another variable of compatible type (1 mutant), changing a class used in conjunction with the `synchronize` keyword (1 mutant). Furthermore, we encountered a `while` keyword changed to an `if` (1 mutant), deleting the condition of an `if` statement and "promoting" the code in the true or false branch up a nesting level (7 mutants), reversing the order of a loop iteration (2 mutants) and swapping

the invocation target and parameter in a method call (1 mutant). Operators capable of producing these mutants may be challenging to implement in practice, because code analysis (e.g., type checking) may be needed to avoid making many uncompilable mutants.

The final category of mutants introduced relatively aggressive changes to code, in line with an instruction to mutate code explanations derived from hazard analysis. These involved: *Adding New Code* (5 mutants); *Deleting Code* (6); *Reordering Code Statements* (13); or involving a *Substantial Re-write* of sub-sections of code across the method involving all three categories (30 mutants). While rule-based systems have deletion operators, they tend not to delete more than individual lines or blocks, as in these cases. Nor do they introduce completely new code statements or reorder them. The latter case in particular would have scope for a very large number of mutants, many of which may be uncompilable. These mutants appeared in similar quantities across both the *killed* and *survived* categories of mutants, implying that while they may involve relatively large code changes/mutational code sizes, they might aid developers because they did not always introduce trivially detectable behavioral differences found by the existing tests.

We speculate that prior LLM-based approaches—for instance, those studied in this paper, namely LLMorpheus and Mutahunter—are not able to generate these large, aggressive mutants because they focus on specific parts of the code. Interestingly, this makes these LLM-based methods more like the rule-based approach in Major. In contrast, it seems that both *m*-hazop and *m*-stpa consider both the method at large and the wider operations that it performs.

Observation 3: Hazard-based LLM approaches generate mutants with large mutational sizes that the rule-based and other LLM-based approaches do not. These larger mutations do not imply large behavioral differences that are trivial to detect, with many of these mutants remaining unkilled by the developer tests.

The Usefulness of Surviving Mutants Generated by Hazard-Based LLM Approaches in Improving Developer Test Suites. We also took a deeper look at four of the examples of mutants in the *survived* category, and now describe them and highlight their potential in helping the developer write a more robust test suite.

(1) For *Chart*, in the `AbstractCategoryItemRenderer` class and the `getLegendItems` method, *m*-hazop deletes a `null` check, replacing it with an exception handler. This `null` check code could be dead or pseudo-tested code [32]. (2) For *Math*, *m*-hazop produced a mutant for the `sample` method in the `DiscreteDistribution` class that rewrote a section of code so that instead of filling all the elements of a newly initialized array with a sample value, the first element is instead re-assigned a value in each iteration. The test suite could therefore be improved to check further values of the array. (3) For *Lang*, *m*-stpa produced a mutant for the `addAll` method in the `ArrayUtils` class that rewrote a standard library call to `System.arraycopy` to a `for` loop that managed the copy itself. This introduces subtle behavioral differences, since `System.arraycopy` may throw exceptions before copying anything due to array-level type rules, while a loop-based approach would only throw an exception due to an actual element that cannot be stored. This is another potentially useful additional angle for the test suite to consider. (4) Finally, for the `getInstance` method in *Time*'s `GJChronology` class, *m*-stpa rewrote a check related to when the calendar switches from Julian to Gregorian, changing it from using a local calendar date in a chosen time zone to comparing a raw UTC timestamp against a fixed cutoff. Because of this, the two versions of the code can disagree near midnight in some time zones, introducing a very subtle difference in behavior not considered by any of the developer tests.

Table 8: Levenshtein Distances between GPT-4.1-generated Mutants and Defects in Defects4J.

		Levenshtein Distances from Defects				
Approach	Total	Min.	Median	Max.	Std.	# at Min (%)
m-hazop	3589	0	194.00	12216	666.56	26 (0.72%)
m-stpa	2490	0	112.00	12051	527.04	27 (1.08%)
m-unguided	2681	0	203.00	4349	533.47	14 (0.52%)
m-llmorpheus	26268	0	85.00	12176	4057.73	14 (0.05%)
m-mutahunter	1259	0	69.00	12076	1063.01	4 (0.32%)

Observation 4: Hazard-based LLM approaches generate mutants that have useful qualities which can help developers improve their tests to confirm that the program does not contain subtle defects.

7 Threats To Validity

Internal. Since Large Language Model (LLM) choice can influence *multiplex*'s generation of mutants, it is a validity threat. We mitigated it by using two distinct models—one that was cloud-based (i.e., GPT-4.1) and one that was locally deployed (i.e., gpt-oss:20B)—with the results suggesting that, although differences exist, both the two hazard-guided methods and the three existing LLM-based techniques work with both models. LLMs are known to both exhibit non-deterministic behavior and to be influenced by configuration parameters like the temperature [34]. As such, a validity threat for this paper's study is that we did not systematically vary these parameters. Yet, we used the default temperature of both LLMs, as did Wang et al. [45]. Future studies should investigate how varying such LLM parameters influences mutant quality and diversity. It is also possible that there are defects in the *multiplex* implementation, its interaction with an LLM, or the platform hosting the LLM. We mitigated these concerns by testing our tool with small-scale subjects and building into *multiplex* a retry mechanism for cases where it failed to generate mutants due to, for instance, timeouts or rate limits. Finally, since using different prompts or source code context could influence the results, we provide *multiplex*, its prompts and LLM configurations in a publicly available replication package [1].

External. Our use of gpt-oss:20B ensures that researchers can easily replicate this paper's results without a cloud-based LLM. Yet, since these LLMs have a training data cut-off date of June 2024 and may not feature the most recent advancements, future work should explore more models to further validate the generalizability of all methods in *multiplex*. Another validity threat is that the LLMs have likely seen Defects4J in their training data. While prior research used ConDefects to address this concern [46], it is noteworthy that this defect dataset was released in 2024 and thus is also likely in the training data of both LLMs. To address this concern, we have calculated the Levenshtein distance from each mutant to the original method defect, as presented in Table 8. While all approaches produced mutants with a Levenshtein distance of 0, Table 8 shows that at most this was 1.08% of an approach's mutants and, therefore, does not have a substantial impact on the generalizability of these

results. Finally, this paper does not use other datasets, such as Bears [31] and Bugs.jar [2], as they contain fewer bugs and do not offer an integrated method for reproducing the study with Major.

Construct. The hazard-guided approaches restrict mutant generation to constructors and methods, excluding possible defects in class fields or other elements of Java programs. This limitation stems from the way in which hazard analysis processes map to method-level control flow. While this may prevent replication of certain fault types, it aligns with the focus of the two hazard-guided approaches and covers a substantial portion of real-world defects. Future work could extend hazard analysis concepts to other Java program elements. This paper's experiments empirically compared *multiplex*'s use of HAZOP and STPA to (i) the *m*-unguided method that prompts the LLM without instructions guided by hazard analysis; (ii) two other LLM-based tools, Mutahunter and LLMorpheus; (iii) and a rule-based tool, Major. Of the LLM-based tools, we picked Mutahunter as an open-source tool from GitHub and LLMorpheus as a recent research-based offering [43]. We also chose Major for its integration with Defects4J, thereby enabling comparisons between defects and mutants. Although this paper focused on five representative mutation methods, future work should extend the experiments to incorporate other rule-based and LLM-driven ones.

8 Related Work

Hazard Analysis in Mutation. This paper is not the first one to apply hazard analysis to the task of mutant generation. Kim et al. applied the HAZOP safety technique to systematically propose a comprehensive set of mutation operators for Java programs [26]. However, this work remained at the conceptual level, with no realized implementation or follow-up research appearing in subsequent papers. Zhang et al. also used HAZOP for mutation of use cases in the Restricted Use Case Modelling (RUCM) methodology [47]. Like the aforementioned paper by Kim et al., this work restricted its use of hazard analysis to defining the mutation operators. Finally, while Gurbuz et al. use hazard analysis to pick traditional mutation operators for testing safety-critical systems [14], several parts of their method are not automated and need an expert's domain knowledge. Since they automate both the identification of conceptual hazards and the generation of context-aware program mutants, this paper's hazard-guided methods differ from these prior schemes.

LLMs in Mutation Analysis. Given the capability of LLMs in code generation [6], using LLMs to generate mutants has been explored by several prior works. However, the first approaches trained a deep learning model on program defects and used it to generate mutants that mimic them [35, 41, 44]. Since these approaches, exemplified by tools like DeepMutation, required a developer to train a deep learning model on real defects, they were not widely adopted. Tip et al.'s tool for the mutation testing of JavaScript programs, called LLMorpheus, instructs pre-trained LLMs to overwrite specific regions of code marked as "placeholders" [43]. Similar to LLMorpheus, the μBERT tool [9, 25] masks specific nodes in a program's AST and uses an LLM fine-tuned on source code to fill in the masked regions.

Like μBERT, LEAM also uses the program's AST when it generates mutants by learning to apply grammar-based rules for mutation testing [41]. While these tools focused on making changes to a specific region of the program's AST, the BugFarm tool uses the LLM's attention mechanism to focus on the AST's least-attended regions, yielding mutants that differ from the original program in multiple locations [16]. Wang et al. also studied LLMorpheus, showing that it generates more diverse mutants than rule-based methods [45]. Finally, Mutahunter is an open-source LLM-based mutation testing tool for Java programs that arose independently of research in this area [7]. Since LLMorpheus and Mutahunter are representative of LLM-based methods that can run on commodity hardware, *multiplex* integrates them both for empirical comparison with *m*-hazop and *m*-stpa, the two novel hazard-guided methods.

LLMs have not only been used to generate code mutants. Tian et al. use LLMs to automatically identify equivalent mutants, finding they can outperform traditional deterministic and machine learning methods [42]. Similarly, Straubinger et al. use LLMs for equivalent mutant detection through an iterative debugging process that generates test cases to expose behavioral differences between mutants and original programs [40]. While these approaches use LLMs to analyze mutants after they are generated, this paper's methods use hazard analysis to guide the LLM when generating mutants.

Mutation Testing for Test Generation. Several researchers have explored using mutation testing in conjunction with LLMs to generate test cases. Dakhel et al. [8] show that pre-trained LLMs can generate effective test cases when guided by mutation testing feedback. Harman et al. [15] present an industrial application of mutation-guided test generation at Meta, demonstrating that LLMs can generate test cases that kill mutants produced by rule-based mutation operators. These works focus on using mutation testing to improve LLM-based test generation, whereas this paper focuses on using LLMs to improve mutation testing itself by generating more realistic and diverse mutants than rule-based approaches can produce.

9 Conclusions and Future Work

This paper presents *multiplex*, an integrated suite of three prior LLM-based mutation testing methods and two novel techniques, *m*-hazop and *m*-stpa, that leverage hazard analysis. The paper's experiments applied Major and *multiplex* to 15 Java projects from Defects4J. Although the results show the complementary nature of the studied mutant creation techniques, there is also evidence that the large, aggressive mutants generated by the hazard-guided approaches are both hard for developer-written tests to detect and beneficially similar to developer mistakes. Since this paper's results highlight the distinct roles that rule-based and LLM-based mutation testing methods play in the creation of a high-quality test suite, future studies should explore both their trade-offs and profitable combinations. This paper's promising results further suggest that, along with extending *multiplex* to work for more programming languages and constructs, future work should ameliorate the challenges arising from uncompilable or equivalent mutants while capitalizing on the potential to produce the diverse and aggressive mutants that are poised to best help developers improve their tests. Ultimately, combining this paper's method with the proposed future work will yield a mutation analysis technique that generates diverse mutants that can ensure tests will detect real-world defects.

Acknowledgments. We thank Rimsha Chaudhry for extracting method locations. Megan Maton is funded by the EPSRC Doctoral Training Partnership with the University of Sheffield, grant EP/W524360/1. Phil McMinn is supported, in part, by the EPSRC grant "Test FLARE" (EP/X024539/1).

References

[1] [n. d.]. Replication Package (https://github.com/LLM-Mutation/ease-2026-replication-package).

[2] 2018. Bugs.jar: A Large-scale, Diverse Dataset of Bugs for Java Program Repair (https://github.com/bugs-dot-jar/bugs-dot-jar).

[3] Z. Ahmed, E. Schwass, S. Herbold, F. Trautsch, and J. Grabowski. 2024. A New Perspective on the Competent Programmer Hypothesis Through the Reproduction of Real Faults with Repeated Mutations. *Software Testing, Verification and Reliability* 34, 3 (2024), e1874.

[4] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006).

[5] T.A Budd and D. Angluin. 1982. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica* 18 (1982).

[6] M. Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[7] CodeIntegrity. 2025. Mutahunter (https://github.com/codeintegrity-ai/mutahunter).

[8] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais. 2024. Effective Test Generation using Pre-trained Large Language Models and Mutation Testing. *Information and Software Technology* 171 (2024).

[9] R. Degiovanni and M. Papadakis. 2022. μBert: Mutation Testing using Pre-Trained Language Models. In *Proceedings of International Conference on Software Testing, Verification and Validation Workshops*.

[10] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978).

[11] C. A. Ericson. 2015. *Hazard Analysis Techniques for System Safety* (2nd ed.). John Wiley & Sons.

[12] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. 1993. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering* 19, 3 (1993).

[13] G. Gay and A. Salahirad. 2023. How Closely are Common Mutation Operators Coupled to Real Faults?. In *Proceedings of the International Conference on Software Testing, Verification and Validation*.

[14] H. G. Gurbuz, B. Tekinerdogan, C. Catal, and N. P. Er. 2024. Test Suite Assessment of Safety-critical Systems using Safety Tactics and Fault-based Mutation Testing. *Cluster Computing* 27, 4 (2024).

[15] M. Harman, J. Ritchey, I. Harper, S. Sengupta, K. Mao, A. Gulati, C. Foster, and H. Robert. 2025. Mutation-Guided LLM-Based Test Generation at Meta. In *Proceedings of the International Conference on the Foundations of Software Engineering*.

[16] A. R. Ibrahimzada, Y. Chen, R. Rong, and R. Jabbarvand. 2025. Challenging Bug Prediction and Repair Models with Synthetic Bugs. In *Proceedings of the International Conference on Source Code Analysis and Manipulation*.

[17] Y. Jia and M. Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (2009).

[18] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011).

[19] R. Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of International Symposium on Software Testing and Analysis*.

[20] R. Just, D. Jalali, and M. D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis*.

[21] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the International Symposium on Foundations of Software Engineering*.

[22] R. Just, G. M. Kapfhammer, and F. Schweiggert. 2011. MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. In *Proceedings of the International Conference on Automated Software Engineering*.

[23] R. Just, G. M. Kapfhammer, and F. Schweiggert. 2011. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis. In *Proceedings of the International Workshop on Automation of Software Test*.

[24] Gregory M. Kapfhammer. 2004. Software testing. In *The Computer Science Handbook*. CRC Press.

[25] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. Le Traon. 2023. Efficient Mutation Testing via Pre-Trained Language Models. *arXiv preprint arXiv:2301.03543* (2023).

[26] S. Kim, J. A. Clark, and J. A. McDermid. 1999. *The Rigorous Generation of Java Mutation Operators Using HAZOP*. Technical Report 2/8/99. Department of Computer Science, University of York, York, UK.

[27] T. A. Kletz. 1999. *HAZOP and HAZAN: Identifying and Assessing Process Industry Hazards* (4th ed.). Institution of Chemical Engineers, Rugby, UK.

[28] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng. 2014. Mutant Subsumption Graphs. In *Proceedings of the International Workshop on Mutation Analysis*.

[29] B. Kushigian, A. Rawat, and R. Just. 2019. Medusa: Mutant Equivalence Detection Using Satisfiability Analysis. In *Proceedings of the International Workshop on Mutation Analysis*.

[30] N. G. Leveson. 2011. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press.

[31] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*.

[32] M. Maton, G. M. Kapfhammer, and P. McMinn. 2024. Exploring Pseudo-Testedness: Empirically Evaluating Extreme Mutation Testing at the Statement Level. In *Proceedings of the International Conference on Software Maintenance and Evolution*.

[33] OpenAI. 2021. Introducing GPT-OSS. *https://openai.com/index/introducing-gpt-oss/* (2021).

[34] S. Ouyang, J. M. Zhang, M. Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2, Article 42 (2025).

[35] J. Patra and M. Pradel. 2021. Semantic Bug Seeding: A Learning-based Approach for Creating Realistic Bugs. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[36] G. Petrovic and M. Ivankovic. 2018. State of Mutation Testing at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*.

[37] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just. 2022. Practical Mutation Testing at Scale: A View from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022).

[38] A. B Sánchez, P. Delgado-Pérez, I Medina-Bulo, and S. Segura. 2022. Mutation Testing in the Wild: Findings from GitHub. *Empirical Software Engineering* 27, 6 (2022).

[39] A. B. Sánchez, J. A. Parejo, S. Segura, A. Durán, and M. Papadakis. 2024. Mutation Testing in Practice: Insights From Open-Source Software Developers. *IEEE Transactiions on Software Engineering* 50, 5 (2024).

[40] P. Straubinger, M. Kreis, S. Lukasczyk, and G. Fraser. 2025. Mutation Testing via Iterative Large Language Model-Driven Scientific Debugging. In *Proceedings of International Workshop on Mutation Analysis*.

[41] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang. 2023. Learning to Construct Better Mutation Faults. In *Proceedings of the International Conference on Automated Software Engineering*.

[42] Z. Tian, H. Shu, D. Wang, X. Cao, Y. Kamei, and J. Chen. 2024. Large Language Models for Equivalent Mutant Detection: How Far Are We?. In *Proceedings of the International Symposium on Software Testing and Analysis*.

[43] F. Tip, J. Bell, and M. Schaefer. 2025. LLMorpheus: Mutation Testing using Large Language Models. *IEEE Transactions on Software Engineering* 51 (2025).

[44] M. Tufano, J. Kimko, S. Wang, C. Watson, G. Bavota, M. Di Penta, and D. Poshyvanyk. 2020. DeepMutation: A Neural Mutation Tool. In *Companion Proceedings of the International Conference on Software Engineering*.

[45] B. Wang, M. Chen, M. Deng, Y. Lin, M. Harman, M. Papadakis, and J. M. Zhang. 2025. A Comprehensive Study on Large Language Models for Mutation Testing. *arXiv preprint arXiv:2406.09843* (2025).

[46] Y. Wu, Z. Li, J. M. Zhang, and Y. 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*.

[47] H. Zhang, T. Yue, S. Ali, and C Liu. 2016. Towards Mutation Analysis for Use Cases. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*.

[48] H. Zhu, P. A. V. Hall, and J. H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Compututing Surveys* 29, 4 (1997).